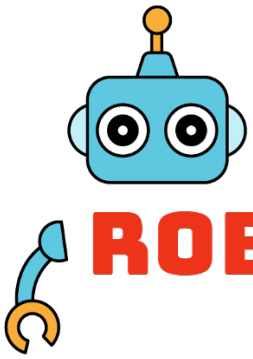


ROBOSAPIENS

D1.1

RoboSAPIENS architecture and semantics for verification and uncertainty quantification

WP1

Public Document

Grant Agreement	101133807
Project	RoboSapiens
Deliverable Number	D1.1
Version	2
Due Month	18
Date	July 2025

http://robosapiens-eu.tech/

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or HADEA. Neither the European Union nor the granting authority can be held responsible for them.

Contributors:

Shaukat Ali, SRL
Ziggy Attala, UoY
James Baxter, UoY
Ana Cavalcanti, UoY
Claudio Gomes, AU
Chengjie Lu, SRL
Hassan Sartaj, SRL
Jim Woodcock, UoY
Fang Yan, UoY

Editors:

Ana Cavalcanti, UoY

Reviewers:

Shaukat Ali, SRL
Peter Gorm Larsen, AU
Jim Woodcock, AU

Consortium:

Aarhus University	**AU**	University of Antwerp	**UA**
Aristotle University of Thessaloniki	**AUTH**	Norwegian University of Science and Technology	**NTNU**
Danish Technological Institute	**DTI**	PAL Robotics	**PAL**
Fraunhofer IFF	**IFF**	ISDI Accelerator	**ISDI**
University of York	**UoY**	Simula Research Lab	**SRL**

Funded by
the European Union

Document Revision History:

Ver	Date	Author	Description
0.1	22-06-2024	Ana Cavalcanti	Initial document version
0.2	28-10-2024	Ziggy Attala	Chapter 3 draft completed
0.3	17-05-2025	James Baxter	Merged material on architecture modelling into Chapter 2
0.4	22-05-2025	Ziggy Attala	Merged material from NFM paper into Chapter 3
0.5	22-05-2025	James Baxter	Added RoboArch translation rules appendix
0.6	28-05-2025	Fang Yan	Added deadlock-freedom verification section
1	30-05-2025	Ana Cavalcanti	First full draft
1.1	10-06-2025	Jim Woodcock	Expanded Appendix D
1.2	11-06-2025	Ana Cavalcanti	Addressed all comments
1.3	25-06-2025	Fang	Added an example of deadlock-freedom verification
2	26-06-2025	Ana Cavalcant	Final checks

Funded by
the European Union

Abstract

This report presents foundations for modelling and verifying self-adaptive intelligent systems. First, we describe an abstract but formal view of the RoboSAPIENS architecture for adaptation: MAPLE-K. For that, we extend a formal language called RoboArch for describing architectural designs of robotic applications. We describe the RoboArch metamodel, well-formedness conditions for the MAPLE-K architecture, and its formal semantics. The formalisation is described in RoboChart, a domain-specific diagrammatic notation for platform-independent, timed, reactive behavioural modelling of control software, with a process algebraic semantics described in a variant of CSP (Communicating Sequential processes).

Our second contribution here is an extension and mechanisation of RoboChart to support the description of components implemented by a neural network. We describe the metamodel and well-formedness conditions of RoboChart with neural networks and its implementation as part of RoboTool: the Eclipse-based tool that supports modelling and verification using RoboChart.

We give a semantics for our extension of RoboChart using *Circus*, a variant of CSP and Z. We define and mechanise the semantics, obtaining a tool for the automatic generation of *Circus* models that use the theorem prover, Isabelle. We also show how Isabelle can be used to reason about *Circus*, and therefore, MAPLE-K models. Guided by the needs of industrial partners, our focus at this stage is automated proof of deadlock freedom, without concerns about model size, but the underlying technology and compositional-proof approach are general.

Our third contribution is the definition of a conformance relation $conf(\epsilon)$ for intelligent software that accommodates the loss of precision derived from using a neural network and the associated argument for compositional reasoning. We define $conf(\epsilon)$ for an arbitrary *Circus* process and prove compositionality results. These translate to compositionality results for RoboChart and RoboArch.

The use of neural networks may also introduce uncertainty in the software. Our final contribution is a technique based on Monte Carlo dropout to quantify uncertainty that may arise during the design and training of DL models. Based on the model predictions, two types of novel quantification metrics are being used for classification and regression. The method and metrics have been applied to the DTI use case. We are also investigating the applicability of the method and metrics to other use cases, such as the NTNU use case.

We also present our investigation into the potential of using large language models (LLMs) to identify uncertainties in robotic software. This will serve as the basis for developing LLM-based uncertainty identification tools for our use cases.

We explain how all these results underpin our vision for the RoboSAPIENS framework, and use examples inspired by RoboSAPIENS case studies to illustrate and evaluate these results. This complements the work presented in D1.2.

Funded by
the European Union

Contents

Funded by
the European Union

Funded by
the European Union

1 Introduction

First of all, this chapter presents the motivation for our work in Section 1.1, where we highlight the contribution of WP1 to the RoboSAPIENS workflow (Figure 1), and justify our choices of notations and techniques. Finally, in Section 1.2 we provide a description of the structure of this report as a whole.

1.1 Motivation

Together with Deliverable 1.2, which describes our work on requirements, this deliverable describes our progress towards realising the conceptual framework of RoboSAPIENS (left-hand side of Figure 1). Our goal is to provide notations and techniques that can describe the conceptual artefacts of a RoboSAPIENS system and provide the mathematical underpinning to demonstrably connect them (via transformation or integration) to the design and realisation results.

This requires mathematical theories that can be unified to deal with software models and operational requirements. So, we opt for the RoboStar framework [CBB+21] due to its process-algebraic semantics, underpinned by Hoare and He's Unifying Theories of Programming [HJ98a] (UTP). With that, we can carry out both system and component-level reasoning, benefitting from stepwise and compositional refinement-based techniques. Moreover, RoboStar provides a baseline of notations accessible to practitioners: diagrammatic and controlled natural languages.

The process algebra used is *CyPhyCircus* [FCC+20, MSF20]. It combines the state-rich process algebra for refinement *Circus* [CSW03], itself combining CSP [Ros11] and Z [WD96a] for data modelling, with Differential Dynamic Logic [Pla10]. Both *Circus* and *CyPhyCircus* have a UTP semantics, with a mechanisation in the Isabelle [NWP02] theorem prover called Isabelle/UTP [FCC+20, WCF+23]. Encoding of *Circus* in standard CSP enables reasoning using the refinement model checker FDR [GRABR14]. Isabelle/UTP provides complementary support via theorem proving to address the state explosion that hinders the use of model checking. This is particularly the case when dealing with the *CyPhyCircus* hybrid models.

The RoboStar notation for architectural description is RoboArch [BCM22]. It has been developed to allow the description of layered designs for platform-independent robotic control software, as well as design patterns for each layer. Most architectures used in robotics are described informally in the literature, with variations often described by different authors. At best, patterns are realised in an implementation or programming language. Such descriptions mix the core concepts of the architecture with those of the application or programming language.

RoboArch's existing metamodel and well-formedness conditions specify a notion of layered architecture and the reactive-skills architectural pattern for control layers. Formalisation of these architectures is achieved through transformation into a RoboChart model. RoboChart [MRL+19] is a domain-specific language with a formal semantics described in CSP. The transformation to RoboChart accurately captures the meaning of each architectural element defined in the RoboArch metamodel. RoboArch's model-transformation approach, based on 50 rules, is automated to generate a RoboChart model from a RoboArch architectural design.

Funded by
the European Union

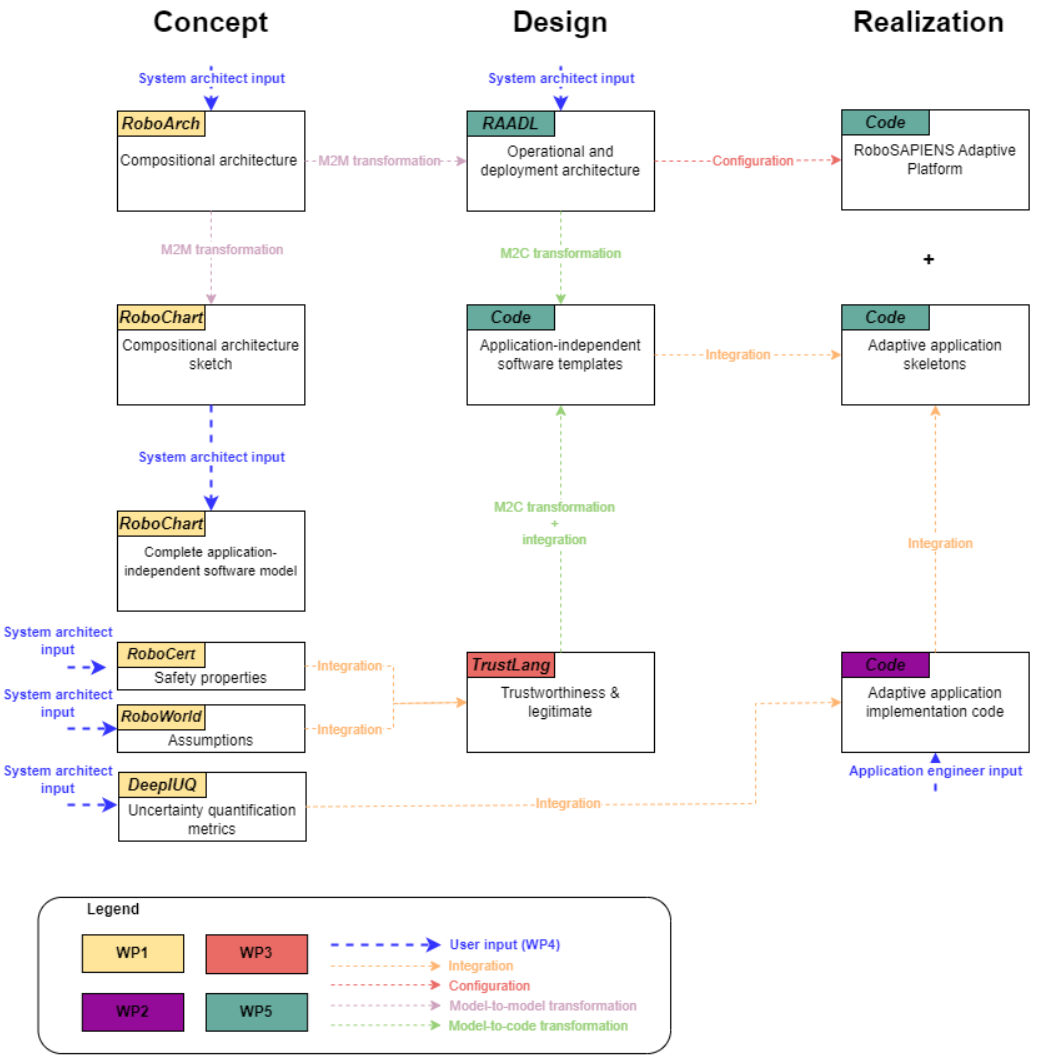


Figure 1: RoboSAPIENS big-picture high-level workflow.

In this report we present our extension of RoboArch to define a RoboSAPIENS architecture: MAPLE-K. We provide a metamodel, well-formedness conditions, and transformation rules. In doing so, we formalise the MAPLE-K architecture and its variations. There are options to omit and coalesce the M, A, P, L, E, and K components, as well as to execute the control loop sequentially or in parallel.

For intelligent RoboSAPIENS systems, RoboChart is insufficient. RoboChart provides a diagrammatic notation to describe a simple component model and behaviour using state machines. A RoboChart model for a control software is given by a module block, containing a robotic platform block and one or more controller blocks. A robotic platform describes variables, events, and operations that capture an abstraction of the services of an actual robot that is used by the software.

The behaviour of a controller is specified by one or more state machines representing parallel threads of control. A controller or a state machine is an independent modelling component whose behaviour is described using a well-defined context

Funded by the European Union

of variables, events, and operations. The observable behaviour is defined in terms of the changes of values of shared variables, occurrence of events (input or outputs), and calls to operations (of the platform).

Following the suggestion in [ACW23], we present an extension of RoboChart to allow the description of a controller via the hyperparameters of a trained, fully connected feedforward artificial neural network (ANN). We give the metamodel, well-formedness conditions, and semantics for this extension of RoboChart.

Refinement requires precision. A component or system A is refined by another one B if the behaviours of B are allowed by A. For example, if B (after a particular trace of interactions) can produce an output of value v over a channel *out*, that same output, with the same value v, must be allowed by A. This is not appropriate for ANNs, because an ANN will not have that level of precision, so we need to characterise and accept results that are close enough.

For this reason, we formalise here a notion of conformance to replace refinement when dealing with RoboChart models with ANN controllers. The new notion is parametrised by a tolerance ϵ that bounds the acceptable loss of precision. Moreover, we present theorems that establish properties of the new notion of conformance needed to support compositional verification. Compositionality is now predicated on the context, that is, other controllers, not requiring or being affected by loss of precision as long as it is bounded by ϵ.

The RoboSAPIENS framework (see Figure 1) uses two additional RoboStar notations: RoboCert [WC22, YCF+22a] to capture properties, and RoboWorld to describe operational requirements. The latter is presented in Deliverable D1.2. Work on the case studies will inform future extensions to these notations.

Use of an ANN may also lead to uncertainties. Here, we require the definition of metrics as input for a technique for uncertainty quantification (UQ). Uncertainty may arise during the design and training of deep learning (DL) models. A UQ method based on Monte Carlo (MC)-Dropout is being developed. MC-Dropout quantifies uncertainty in the model predictions. Based on the model predictions, two types of UQ metrics are being developed for classification and regression, which include not only standard metrics (for example, entropy) but also some novel metrics (for example, prediction surface). Additionally, novel uncertainty-based robustness metrics are being developed to assess the robustness of deep learning (DL) models. Finally, a benchmark dataset construction method based on VLM and an adversarial generation technique is being developed.

This deliverable describes the application of the UQ method and metrics to the DTI use case, in particular to DL-based sticker detectors for the laptop refurbishment process. We have evaluated the accuracy, uncertainty, and robustness of various sticker detectors and provided guidelines for selecting detectors from different perspectives. The benchmark datasets used for evaluation have been constructed using two visual-language models (DALL-E and Stable Diffusion) and an adversarial-generation technique called DAG. We are also investigating the applicability of the UQ method and metrics to other RoboSapiens use cases, such as the NTNU use case, that is, DL-based digital twins for ships.

Moreover, to systematically identify uncertainties in robotic software, as part of

Funded by
the European Union

building a holistic uncertainty identification and quantification solution, we present an initial investigation into using Large Language Models (LLMs) for this purpose. The results are based on the four RoboSAPIENS use cases. The results will be used to develop LLM-based tools for identifying uncertainty in robotic software.

Finally, we present prototype tools that enable the evaluation of our notations and techniques mentioned above, as well as their application to RoboSAPIENS case studies. The first tool we describe is a mechanisation of RoboArch to support modelling and validation based on the metamodel and well-formedness conditions described here. More importantly, the transformation of RoboSAPIENS architectures, as described in RoboArch, to RoboChart models is automated.

Further work is based on Isabelle. We report on our mechanisation of *CyPhyCircus* in Isabelle/UTP. It is the basis for mechanising a model-to-model transformation that connects the output of RoboTool to an Isabelle theory, allowing the use of theorem proving to reason about RoboChart models. We cover standard RoboChart models and RoboChart models with ANN controllers.

We have also initial results on the mechanisation of legitimisation and trustworthiness checking in Isabelle. We focus here on checks of deadlock freedom based on RoboChart models. The approach and infrastructure of proof are, however, more general. We will work on additional properties and even test generation using that infrastructure. The vision is that replanning may and should lead to changes to the RoboChart model, which can be used by a legitimiser or trustworthiness checker to ensure the changes do not introduce problems.

In summary, we focus on this deliverable on the design and verification notations and techniques in Figure 1. We present RoboArch, its transformation into RoboChart, the enrichment of RoboChart with ANNs, associated verification techniques for RoboChart, to support legitimisation, and uncertainty quantification metrics. In the complementary Deliverable 1.2, we present RoboCert and RoboWorld, which address software and system requirements. System requirements are assumptions that complement the specification of the control software in RoboChart. The software requirements are properties that a RoboChart model must satisfy.

1.2 Structure of the deliverable

In this report, Chapter 2 presents the precise characterisation of the RoboSapiens architecture. As said, this is given by defining a metamodel, well-formedness conditions, and a behavioural semantics.

Chapter 3 describes the support for reasoning about AI in RoboSAPIENS. We provide the metamodel, well-formedness conditions, and semantics of a diagrammatic notation that can be used to specify AI-enabled software. A technique for compositional verification based on this notation and its semantics is in Chapter 4.

Uncertainty arising from the use of AI is explicitly addressed in Chapter 5. It also describes an LLM-based approach to identifying uncertainties, to develop a comprehensive solution for uncertainty identification and quantification.

Tools for supporting the notations and techniques of Chapters 2, 3, and 4 are described in Chapter 6. They are extensions of RoboTool and Isabelle.

We conclude in Chapter 7. We summarise our results, and, importantly, we also present detailed plans for the next 18 months of RoboSAPIENS, and beyond.

Funded by
the European Union

2 RoboSAPIENS architecture for verification

This chapter defines the MAPLE-K architecture. In the next section, we present the baseline technology: RoboArch [BCM22] via a very simple example. In Section 2.2, we give an example of a description of an instance of the MAPLE-K architecture using the extension of RoboArch we define here. Our example is the architecture of the navigation example defined in Deliverable D5.1, deployed in a TurtleBot.

The actual definition of the architecture comes in Section 2.3, which defines an extension of the RoboArch metamodel to include the MAPLE-K architecture, and in Section 2.4, which defines a model-to-model transformation to map RoboArch architectures into RoboChart behavioural models.

Afterwards, we present a brief account of related work in Section 2.5. We conclude this chapter in Section 2.6, where we summarise our contributions.

A summary of this work has already been published [BvAK+25].

2.1 RoboArch

As noted, RoboArch architectures are layered, a structure widely used in robotics. Each layer can have a pattern that describes 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`.

RoboArch uses the type system of the formal modelling notation Z [WD96a]. 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` and `Sense`, each with a single 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 robotic firmware and API.

The declaration of each layer may give it a pre-defined type: either `ControlLayer`, `ExecutiveLayer` or `PlanningLayer`, each with its 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**, which are events that may have a type. A `ControlLayer` additionally **uses** or **requires** the same interfaces as the robotic platform, since it is intended to coordinate communication with the robot.

Funded by
the European Union

```
system ObstacleAvoidance

datatype Velocities {
   linear:real
   angular:real
}

interface Motors {
   event move: Velocities
}

interface Sense {
   event proximity: int
}

robotic platform PuckRobot {
   uses Motors
   uses Sense
}

layer Application {
   inputs= eventReply:Events, …;
   outputs= activate:Skills, deactivate:Skills, …;
} ;
```

```
layer MoveAndSense: ControlLayer {
   uses Motors
   uses Sense

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

connections=
Executive on activate to Control on activate,
Executive on deactivate to Control on deactivate,
…
Control on eventReply to Executive on eventReply,
Control on activeSkills to Executive on activeSkills,
…
Control on move to PuckRobot on move,
PuckRobot on proximity to Control on proximity;
```

Figure 2: An example of a RoboArch model

A layer can have a **pattern**. In our example, `MoveAndSense` uses the `ReactiveSkills` pattern [BFG+97]. 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 the robotic platform, ensuring a strict layering discipline is maintained.

RoboArch architectures can be translated to a RoboChart model. RoboChart has similarities to RoboArch in its abstraction of the robotic platform and its type system. A RoboChart model is a module, containing a robotic platform, declaring variables, operations, and events, as well as 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. Here, we describe the features of RoboChart used in models generated from RoboArch as needed. A full account of RoboChart can be found in [MRL+19].

2.2 Overview and example

The MAPE-K and MAPLE-K architectures are described in Deliverable 5.1. Here, we provide a formal account of that architecture. Our formalisation enables the definition of instances of a MAPLE-K architecture as illustrated in this section.

We have extended RoboArch to support MAPE-K and MAPLE-K by defining a new `MAPLE-K` pattern type for a layer. An example of a layer with this pattern can be seen in Fig. 3, which displays the MAPLE-K layer for the navigation example in Deliverable 5.1, along with the types it utilises. The example of the DTI case study is considered in Section 6.1, where we present our RoboArch tool.

Fig. 3 declares a layer named `Adaptation`, which is a `PlanningLayer`, since MAPLE-K loops are intended to be the highest layers of a system, above the layers of the managed system. As with other layers, it declares **inputs** and **outputs**, but its **pattern** is set to the new `MAPLE-K` pattern. The **pattern** declaration is followed by blocks declaring further information about the MAPLE-K components: **monitor**, **analyse**, **plan**, **legitimate** and **execute**. Some of these blocks may be omitted to

Funded by
the European Union

```
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 = spinCommand : SpinConfig;
    pattern = MAPLE-K;
    monitor {
        inputs = lidarData;
        processed_data_type = LidarRange;
        recorded_data = lidarScans : Seq(LidarRange);
    }
    analyse {
        analysis_results = probLidarMasks : Seq(ProbLidarMask);
        analysis_results = boolLidarMasks : Seq(BoolLidarMask);
    }
    plan {
        plan_data = directions: Seq(SpinCommand);
    }
    legitimate {
        // legitimate just signals success or failure
    }
    execute {
        outputs = spinCommand;
    }
};
```

Figure 3: An example of a MAPLE-K pattern in a RoboArch model

define variations on the pattern (for example, omitting **legitimate** for the traditional MAPE-K), but **monitor** and **execute** must always be present to communicate the inputs and outputs of the layer.

The first component, **monitor**, defines a list of **inputs**, which indicate which of the **inputs** of the layer are inputs to the MAPLE-K loop. This distinguishes them from other input events that may be used to communicate with the layer, such as events for requesting information from the knowledge base or for allowing further adaptation by a higher layer. In our example, the only input is `lidarRange` of type `LidarRange`, representing the data read from the TurtleBot's lidar. The **monitor** component may record these inputs in the knowledge base and transform them to a processed form convenient for further analysis in the **analyse** component.

The **processed_data_type** records the data type passed to the **analyse** component. Here, it is `LidarRange`, the type of the input, since the input is not intended to be changed by the **monitor** component. In other systems, the **processed_data_type** could represent the output of an error correction technique, collect multiple inputs, or add metadata such as timestamps to the input data.

The data recorded in the knowledge base by the **monitor** component is indicated by **recorded_data**, which is a list of variable declarations that represent data to be stored in the knowledge base. In Fig. 3, this list contains only a single variable, `lidarScans`, which is a sequence of `LidarRange` values to which the input values are written. In other systems, inputs may be written to variables in the knowledge base in different ways. For example, several inputs may be combined into a single variable in the knowledge base. Alternatively, an input may be recorded in multiple variables depending on its value. For example, we can categorise readings above and below a threshold for further checking in the analysis step by recording the inputs in different variables.

The **analyse** component analyses the data received from the **monitor** component, recording the analysis results in the knowledge base and determining whether an anomaly has occurred. The results of the analysis are recorded in the variables indicated by **analysis_results**, similar to **recorded_data** in **monitor**. In our example,

Funded by
the European Union

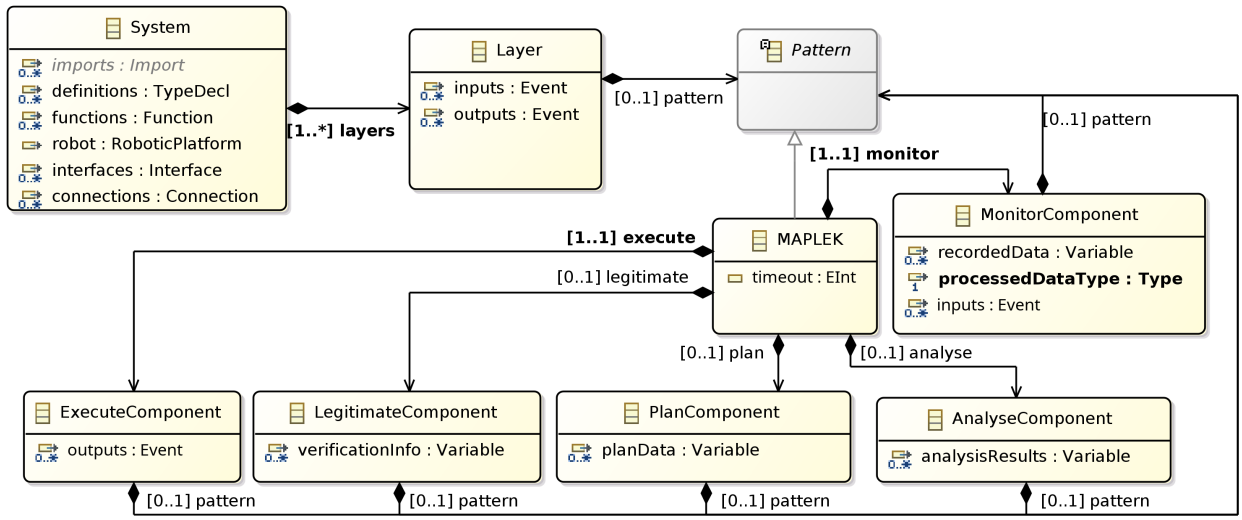


Figure 4: Part of the RoboArch metamodel, showing the MAPLE-K pattern

analysis_results declares two variables: `probLidarMasks` and `boolLidarMasks`. The `probLidarMasks` variable is a sequence of `ProbLidarMask` values, recording for each `LidarRange` the probability that a lidar occlusion exists in each segment of the lidar scan. Similarly, `boolLidarMasks` records a sequence of `BoolLidarMask` values with a boolean determination (computed from `probLidarMasks`) as to whether a lidar occlusion exists in each segment of the lidar scan.

The **plan** component formulates a plan to mitigate anomalies detected by the **analyse** component and records the plan in the knowledge base using variables indicated by **plan_data**, similar to **recorded_data** and **analysis_results**. In our example, this is a sequence of `SpinCommand` values, indicating periodic rotations that the robot should make to mitigate against the occlusion of part of its field of view.

The **legitimate** component verifies and validates the plan formulated by the **plan** component, possibly recording additional information in knowledge base variables indicated by **verification_info**. For our example, we do not require additional information to be produced by the legitimate component, so **verification_info** is omitted. The presence of an empty **legitimate** component still ensures that the component is generated as part of the implementation.

Finally, the **execute** component executes the plan by using outputs of the layer, possibly translating or spacing commands as needed. Similarly to the **inputs** in the **monitor** component, the **execute** component records a list to indicate which **outputs** of the layer are outputs of the MAPLE-K loop. In our example, this is the `spinCommand` output, which sends a single `SpinCommand` to instruct the robot to spin.

2.3 MAPLE-K: Metamodel and well-formedness conditions

The syntax of RoboArch has an underlying metamodel. We have extended it to include the MAPLE-K pattern as shown in Fig. 4.

MK1 A *Layer* that has a pattern of type *MAPLEK* must be a *GenericLayer* or *PlanningLayer*.

MK2 If an instance of *MAPLEK* has a *legitimate* component, then it must have *plan* component.

Figure 5: The well-formedness conditions for *MAPLEK*

The top level of a RoboArch architecture is an instance of the class *System* in Fig. 4. It has one or more *layers*: instances of *Layer*. A layer may have a *pattern* represented by a subclass of *Pattern*. *MAPLEK* is the subclass for our new pattern.

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 correspond to the **`monitor`**, **`analyse`**, **`plan`**, **`legitimate`** and **`execute`** blocks shown in Fig. 3. As mentioned, all these components are optional, except for *monitor* and *execute*, allowing for variations on the pattern.

Each component has its own parameters, corresponding to the items within its block. Each component can also have its own pattern, allowing further variations of MAPLE-K or standard structures for the components to be defined. The definition of additional patterns for each of the components is ongoing.

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 in [BCM22].

The first well-formedness condition, **MK1**, restricts the types of layers that can use a *MAPLEK* pattern to just *PlanningLayer*s and *GenericLayer*s. This is because a MAPLE-K loop is intended to go above the layers of the managed system, and *PlanningLayer*s are the topmost layer types. By allowing for a *GenericLayer* to use *MAPLEK* as well, however, we cater for its use in variations on the usual layered architecture. This can include, for instance, an architecture where we just separate the MAPLE-K loop and the managed system, with no additional layers, or where a MAPLE-K loop communicates directly with the robotic platform.

The second well-formedness condition, **MK2**, restricts the components that can be omitted. The metamodel itself already ensures that a layer with a *MAPLEK* pattern always has a *MonitorComponent* and an *ExecuteComponent*. However, the presence of a *LegitimateComponent* can only be needed if there is a *PlanComponent* with *planData* variables containing a plan that might need to be legitimated. Thus, **MK2** requires a *plan* component to be defined whenever *legitimate* is.

2.4 Semantics: overview and rules

As noted, a RoboArch model can be automatically translated to a sketch of a RoboChart model, with each layer translated to a RoboChart controller. This translation to RoboChart gives semantics to the structure represented by the metamodel and well-formedness conditions. In this section, we discuss the RoboChart

Funded by
the European Union

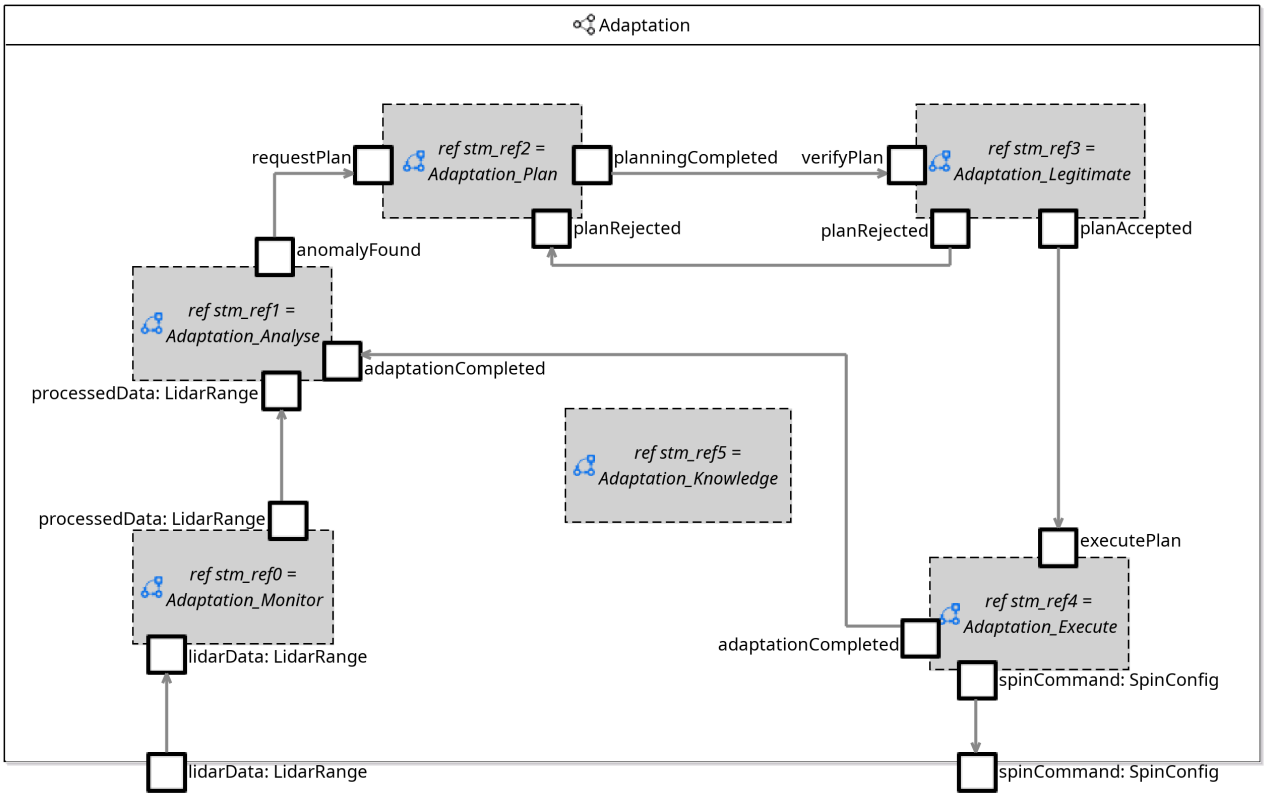


Figure 6: The RoboChart controller generated for the `Adaptation` layer in Fig. 3, showing the events driving the MAPLE-K loop

corresponding to the *MAPLEK* pattern presented in the previous section, first presenting an overview of the RoboChart for our example in Section 2.4.1, then discussing the translation from RoboArch to RoboChart in Section 2.4.2.

2.4.1 Overview

Fig. 6 shows the RoboChart controller for the *MAPLEK* layer `Adaptation` in Fig. 3, as an example of how we capture the behaviour of a MAPLE-K architecture, particularly when all components are present. As shown, each component of the MAPLE-K loop is represented by a state machine, with an additional state machine, *Knowledge*, representing the knowledge base. Each of the state machines is included in the controller by reference and linked by event connections. These event connections can be divided into two kinds: signals that drive the MAPLE-K event loop, and events that communicate with the knowledge base to get or set data. The first kind of events can be seen in Fig. 6, with connections linking the state machines, but we omit the second kind of events in Fig. 6 due to their large number, so that the flow of control can be seen more clearly. The omitted events are shown later, in Fig. 7, where we hide the first kind of events for clarity of presentation.

The six state machines of the *Adaptation* controller all have their names prefixed with *Adaptation_*, to ensure the names are unique when there are multiple *MAPLEK* layers. The state machine named *Adaptation_Monitor* corresponds to the **`monitor`** block in RoboArch. It receives inputs from the managed system and sends on

Funded by the European Union

processed data via an event *processedData*, which has the type of the **`monitor`** component's **`processed_data_type`**. The inputs are those declared in the RoboArch model, in this case *lidarData* of type *LidarRange*, which represents data from a lidar device detecting objects in the surroundings. The *processedData* event also has the type *LidarRange*, corresponding to the **`processed_data_type`** provided in Fig. 3.

The *Adaptation_Analyse* machine is then triggered to start analysis by the receipt of the *processedData* from the *Adaptation_Monitor* state machine. When analysis is completed, it signals via an *anomalyFound* event if an anomaly is found. This event is connected to the *requestPlan* event of the *Adaptation_Plan* state machine.

The *Adaptation_Plan* state machine creates a plan to adapt to the anomaly after it is signalled by *requestPlan*. It signals to the *Adaptation_Legitimate* state machine on the *planningCompleted* event when a plan has been formulated. *Adaptation_Legitimate* receives the *planningCompleted* event as the *verifyPlan* event, signalling it to perform verification and validation on the plan. If the plan is rejected, *planRejected* is signalled back to the *Adaptation_Plan* state machine to indicate that it should formulate a revised plan. If the plan is accepted, *planAccepted* is signalled to the *Adaptation_Execute* machine to execute the plan.

Adaptation_Execute receives *planAccepted* via the *executePlan* event and communicates with the managed system via output events. It signals back to *Adaptation_Analyse* via *adaptationCompleted* when it has finished, allowing a new adaptation cycle to begin. As with the inputs, the output events are those listed in the **`execute`** block. In this case, the only output is *spinConfig* of type *SpinConfig*, representing instructions for the robot to rotate as it moves to mitigate occlusions.

The knowledge base is represented by the machine *Adaptation_Knowledge*, which contains variables derived from **`recorded_data`**, **`analysis_results`**, **`plan_data`** and **`verification_info`** in RoboArch. As mentioned, the other components access these variables via events to get and set their values, as shown in Fig. 7 (omitting the events already included in Fig. 6). Each variable corresponds to several events:

1. for getting the value of a variable, an event named by prefixing the variable name with *get_* is provided to request the variable value, and the value itself is then sent on an event named with the name of the variable itself,
2. for setting the value of a variable, an event named by prefixing the variable name with *set_* is provided, with a parameter type the same as the variable's type,
3. for components that use variables declared by other component, events are provided as in 1 but suffixed with an underscore and the name of the component using the value (for example, *get_directions_Legitimate* and *directions_Legitimate* are provided to allow *Adaptation_Legitimate* to use the variable **`directions`** from **`plan_data`**),
4. for components external to the MAPLE-K controller to get values of variables, events are provided as in 1 but suffixed with *_ext* (such events are omitted in Fig. 7, since they are unused, because the **`Adaptation`** RoboArch layer declares no inputs or outputs corresponding to those events).

Each state machine is connected to the getter and setter events for the variables

Funded by
the European Union

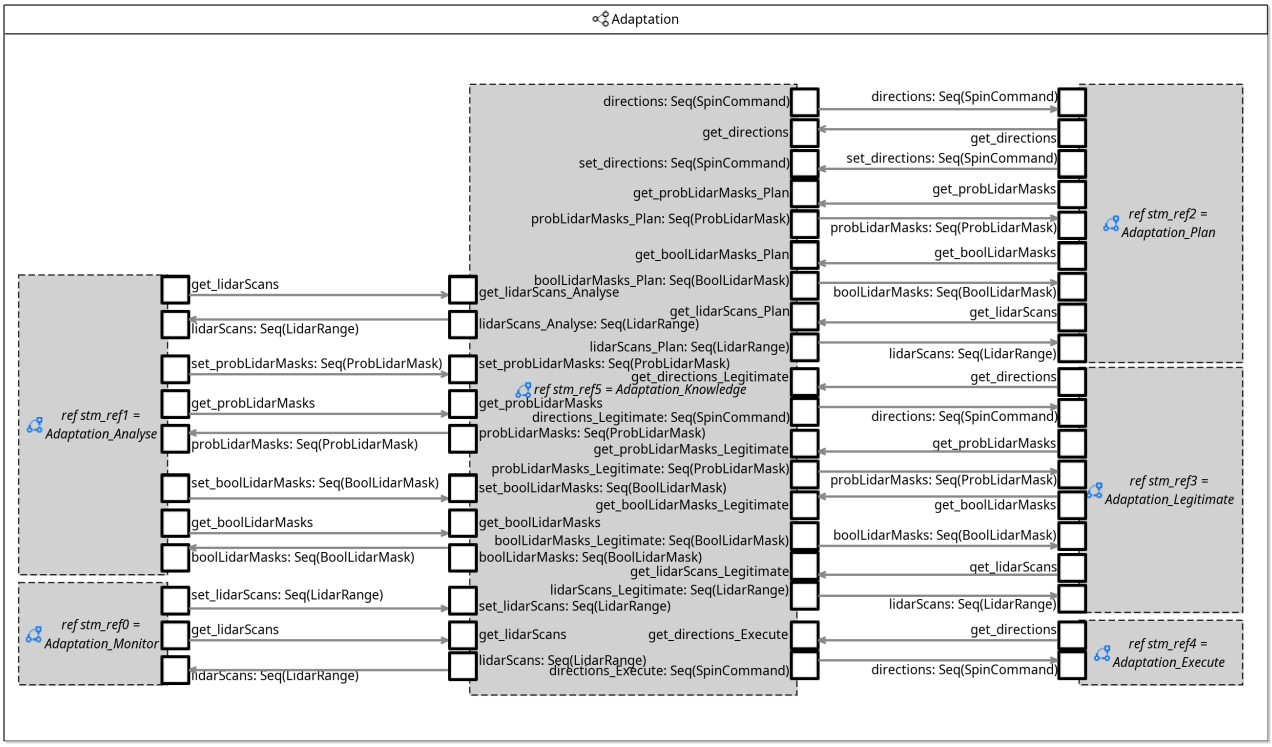


Figure 7: The RoboChart controller generated for the `Adaptation` layer in Fig. 3, showing the events communicating data to and from the *Knowledge* component

declared in the corresponding RoboArch block. It also uses the getter events for any other variables it should have access to, connected to the component-specific events in the *Knowledge* state machine. Note that only the component that defines a variable in RoboArch can use the setter events to write to it, thus ensuring the knowledge will not be subject to arbitrary changes.

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* enables application-specific logic to determine how to handle new data received 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.

Each of the state machines has a standard form, dependent upon the components present in layer's controller (and hence the communications required between them). We discuss them as part of presenting the translation rules next.

2.4.2 Translation from RoboArch to RoboChart

The translation from RoboArch to RoboChart is defined by a series of translation rules that map elements of the RoboArch metamodel to RoboChart constructs that define their semantics. Existing translation rules define the top level of the translation, mapping RoboArch types and interfaces to equivalent RoboChart layers and interfaces, and each layer onto a controller, defining event connections between

Funded by
the European Union

Rule 1. Interfaces for a `MAPLEK` Layer (Rule 1 in Appendix A)

MAPLEKLayerToInterfaces(ammkl : Layer) : Set(Interface) =

(ammkl.name_Inputs
for ev **in** inputs
⚡ ev.name: ev.type
end for

ammkl.name_Outputs
for ev **in** outputs
⚡ ev.name: ev.type
end for

ammkl.name_InputVars
for ev **in** inputs
✗ ev.name_data: ev.type
end for

ammkl.name_InputFlags
for ev **in** inputs
✗ ev.name_flag: boolean
end for)

∪ RecordedDataInterfaces(ammkl.pattern.monitor.recordedData, ammkl.pattern)
∪ ⋃{analyse : ammkl.pattern.analyse •
AnalysisResultsInterfaces(analyse.analysisResults, ammkl.pattern, ammkl.name)}
∪ ⋃{plan : ammkl.pattern.plan •
PlanDataInterfaces(plan.planData, ammkl.pattern, ammkl.name)}
∪ ⋃{legitimate : ammkl.pattern.legitimate •
VerificationInfoInterfaces(legitimate.verificationInfo, ammkl.pattern)}

where
inputs = ammkl.pattern.monitor.inputs
outputs = ammkl.pattern.execute.outputs

provided
ammkl.pattern **instanceof** MAPLEK

controllers that enforce the layering policy. Further types and interfaces are defined for each layer, along with state machines and connections within a controller, depending on the pattern of the layer, if present.

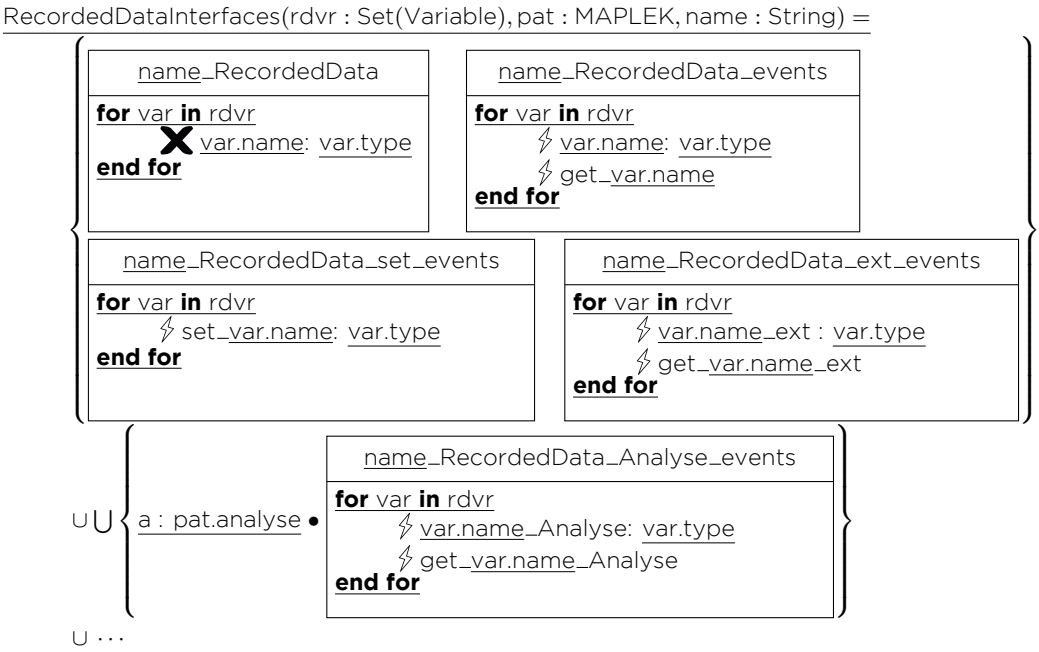
For the *MAPLEK* pattern, there are no additional types defined beyond those defined explicitly in the RoboArch model. The interfaces for the *MAPLEK* pattern are, therefore, defined by Rule 1, which takes a *Layer* as its parameter and returns a set of interfaces to be included in the corresponding RoboChart model.

The rules present RoboChart elements in a graphical format, along with meta-notation in grey and underlined. This meta-notation includes **for** loops to specify repeated elements, and **if** statements to specify conditional elements. Abbreviations of model elements are specified in a **where** clause, and conditions restricting the application of the rule are stated in a **provided** clause. Here, we focus on the rules defining the main RoboChart components corresponding to the *MAPLEK* pattern. The full rules, including those here, can be found in Appendix Atoo.

Rule 1 applies only to layers where the *pattern* is *MAPLEK*, and begins with defining interfaces for the *inputs* of its *monitor* component and the *outputs* of its *execute* component. The interfaces (as well as the state machines, defined in later rules) are named using the name of the layer with a suffix to ensure there are no conflicts if multiple *MAPLEK* layers exist. For brevity, we refer to the interfaces by their name after the layer name prefix. The interface *_Inputs* records the events of *inputs* themselves, and, similarly, the *_Outputs* interface records the events of *outputs*. These interfaces are used to include events in the state machines for the components.

Funded by
the European Union

Rule 2. Interfaces for Recorded Data Variables (Rule 2 in Appendix A)



The interface *_InputVars* declares variables for holding the values communicated by the *inputs*, named with the name of the event appended with *_data*. Similarly, the *_InputFlags* interface declares variables of *boolean* type for each of the *inputs*, named named with the name of the event appended with *_flag*. These are used in the *_Monitor* state machine to collect the values communicated by the *inputs* and record that a communication has occurred on each.

Rule 1 extends these interfaces via set union, with various interfaces for the knowledge base variables declared by each component. The declarations of these interfaces are defined by functions defined in further rules. Each of these functions receives the list of variables for the interface, the *pattern* for the layer (to check for the presence of optional components), and the name of the layer (to include it in interface names). For the variables originating from optional components, we use a set comprehension syntax to use the function only if that component exists, and a unary union operator to flatten the resulting set of sets.

As an example of one of these functions, we show the definition of *RecordedDataInterfaces* in Rule 2. This defines the events and variables corresponding to the *recordedData* variables declared in a *MonitorComponent*. The *_RecordedData* interface contains the variables themselves, which are used in defining the knowledge base. The *_RecordedData_events* interface defines the events for getting the value of the variables: one with the same name and type as the variable for communicating the value, and one with the name of the variable prefixed with *get_* and no type for signalling a request for the value. The corresponding events for setting the value of the variable are declared in the *_RecordedData_set_events* interface, with the name of the variable prefixed with *set_* and the same type as the variable. Having these in a separate interface allows them to be omitted from components that are not permitted to set the value of the variable.

Funded by
the European Union

Rule 3. Machines and Connections for a `MAPLEK` layer (Rule 6 in Appendix A)

```
MAPLEKPatternToMachinesAndConnections(ammkl : Layer) : (Set(StateMachine), Set(Connection)) =

    if # ammkl.pattern.legitimate = 0 then
        if # ammkl.pattern.plan = 0 then
            if # ammkl.pattern.analyse = 0 then MEKMachinesAndConnections(ammkl)
            else MAEKMachinesAndConnections(ammkl)
            end if
        else
            if # ammkl.pattern.analyse = 0 then MEPKMachinesAndConnections(ammkl)
            else MAPEKMachinesAndConnections(ammkl)
            end if
        end if
    else
        if # ammkl.pattern.analyse = 0 then MPLEKMachinesAndConnections(ammkl)
        else MAPLEKMachinesAndConnections(ammkl)
        end if
    end if
provided
    ammkl.pattern instanceof MAPLEK
```

The other interfaces declared in Rule 2 define distinguished events to allow components other than *_Monitor* to get the value of the variables. The events are similar to those in *_RecordedData_events*, but with a suffix to distinguish them. Those in *_RecordedData_ext_events* are suffixed with *_ext* and used by components external to the layer to get the variable values. The *_RecordedData_Analyse_events* interface is defined in a set comprehension and union, so that it is only defined when the *analyse* component is present. It defines events suffixed with *_Analyse*, which are used by the knowledge base to allow the *_Analyse* machine to get the values of the variables. Interfaces are defined in a similar way for *_Plan* and *_Legitimate* to get the values of the variables, but we omit these in Rule 2 for brevity.

The functions defining the remaining interfaces (*AnalysisResultsInterfaces*, *PlanDataInterfaces* and *VerificationInfoInterfaces*) are declared in further rules, which we omit here but present in full in Appendix A. They are similar to Rule 2, declaring interfaces with events for accessing the variables they consider, including interfaces for any other components that need to use those variables.

The RoboChart controller corresponding to the RoboArch layer is created by an existing rule, which adds the *inputs* and *outputs* of the *Layer* to the controller and invokes a further rule defining the state machines for the *Layer*'s *pattern* and the connections between them. The rule defining the state machines and connections for the *MAPLEK* pattern is Rule 3. It takes the *Layer* as input and outputs a pair of sets of *StateMachines* and *Connections*. It is divided into several different cases using nested if statements to handle the different cases of presence or absence of the optional components. In these if statements, the optional components are treated as sets, which are empty when the component is not present; we then check the cardinality (#) of the set. The three optional MAPLE-K components, plus the

Funded by
the European Union

MK2 well-formedness condition, result in six cases that must be considered, each of which is handled by a separate function, to which the *Layer* is passed.

As an example of one of these six cases, we show in Rule 4 the definition of the function *MAPLEKMachinesAndConnections*, which defines the state machines and connections when all the components of the MAPLE-K loop are present. It generates six state-machine references, which point to machine definitions specified by further rules. The connections between them are defined as shown in Fig. 6, with the boundary of the controller (defined outside of the rule) denoted by a dashed line. Additional connections, linking components to the knowledge base, are added in further rules, joined to the sets with a union operator. These additional rules are reused for the structure of connections with fewer components in other rules, retaining only those rules relevant for the components included in the controller. Since these rules for connections to the knowledge base are quite large and the connections they generate are shown in Fig. 7, we omit them here.

Each of the rules for generating the state machines takes in the corresponding RoboArch component (except for *KnowledgeStateMachine*, which just uses the previously defined interfaces), the *MAPLEK* pattern itself (to check for the presence of other components), and the name of the layer (to prefix names). Rule 5 defines the *MonitorStateMachine* function, which defines the *_Monitor* state machine. It uses the *Inputs*, *InputVariables*, *InputFlags*, *RecordedData_events* and *RecordedData_events* interfaces defined previously. This allows it to use the events from those interfaces and includes local variables from those interfaces for storing values. Additionally, *_Monitor* defines two further variables: *outputData*, which has the type of the *processedDataType* and is used to store the data to be output, and *dataToSend*, a *boolean* variable indicating whether there is data to be output. The *processedData* event is also defined, with the *processedDataType* as its type.

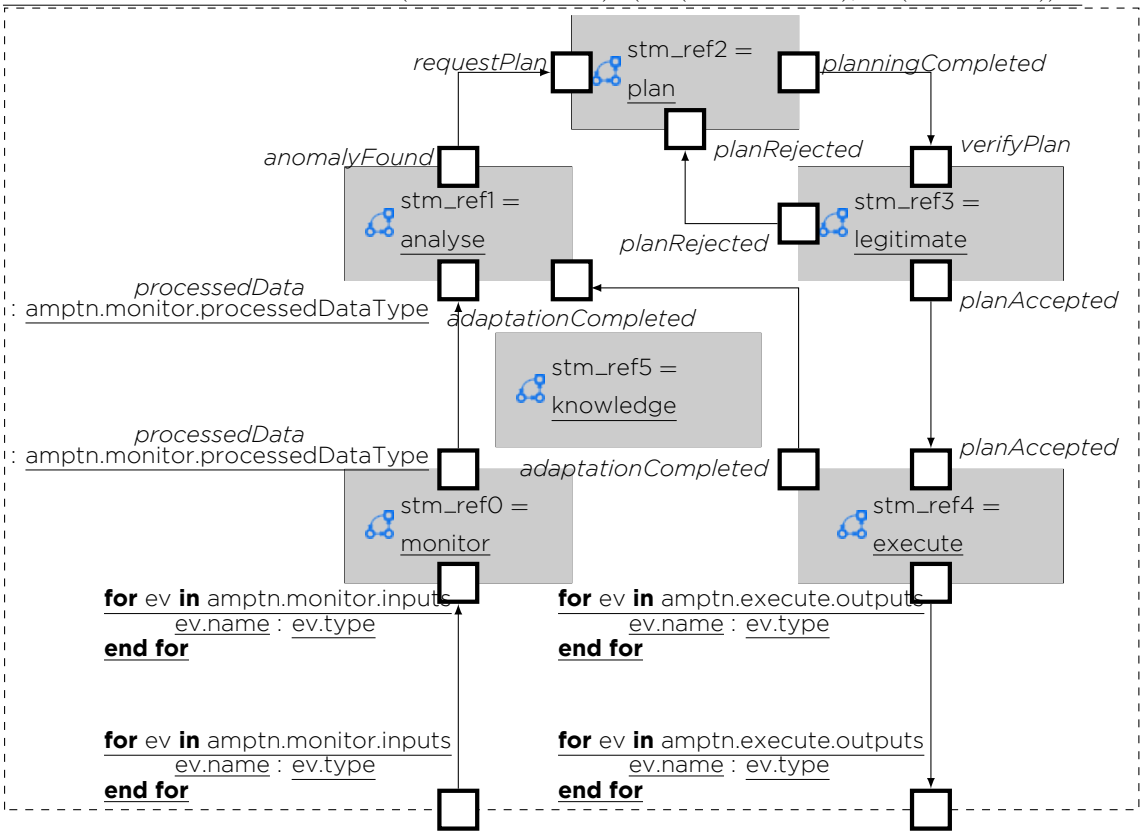
The body of *_Monitor* contains states and transitions to define its behaviour; the states are intended to be extended with application-specific behaviour. The first state is *Initialise*, which allows for application-specific initialisation to be added. A transition goes from *Initialise* to *ReadInput*, where the machine waits until input is available. An application-specific delay or waiting condition can be added here to control how frequently the monitoring occurs. The transitions out of *ReadInput* are triggered by the input events from the *_Inputs* interface (with one transition per input), storing the parameter of the event to the corresponding variable from *_InputFlags*. After the transition is triggered, the corresponding variable from the *_InputFlags* interface is set. This allows the machine to handle multiple inputs, checking which have been received and resetting the *_InputFlags* variables.

After one of the input transitions has been triggered, the *_Monitor* state machine enters the *ProcessData* state, where application-specific processing of the data in the *_InputVariables* variables may be performed, followed by the *RecordData* state, where data can be written to the knowledge base in an application-specific way. Allowing for these functions to be separated across two states means they can be presented more clearly. During the processing or recording of the data, a determination should be made as to whether the data can be sent to the *_Analyse* state machine, and this should be indicated via the *dataToSend* variable. This determination could be based on factors such as whether sufficient data has already been

Funded by
the European Union

Rule 4. Machines and Connections for a `MAPLEK` layer with all components (Rule 7 in Appendix A)

MAPLEKMachinesAndConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =



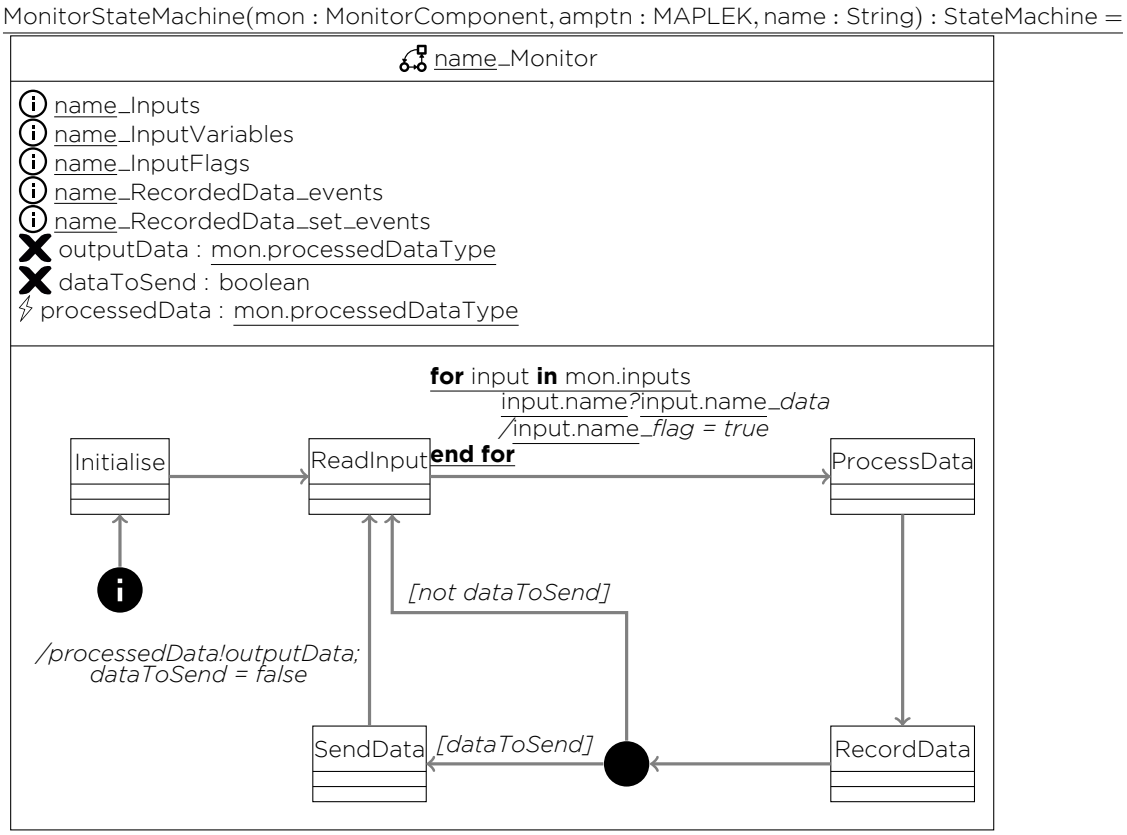
∪ MAPLEKKnowledgeExternalConnections(amptn)
∪ MAPLEKMonitorKnowledgeConnections(amptn)
∪ MAPLEKAnalyseKnowledgeConnections(amptn)
∪ MAPLEKPlanKnowledgeConnections(amptn)
∪ MAPLEKLegitimateKnowledgeConnections(amptn)
∪ MAPLEKExecuteKnowledgeConnections(amptn)

where

monitor = MonitorStateMachine(amlyr.pattern.monitor, amlyr.pattern, amlyr.name)
analyse = AnalyseStateMachine(amlyr.pattern.analyse, amlyr.pattern, amlyr.name)
plan = PlanStateMachine(amlyr.pattern.plan, amlyr.pattern, amlyr.name)
legitimate = LegitimateStateMachine(amlyr.pattern.legitimate, amlyr.pattern, amlyr.name)
execute = ExecuteStateMachine(amlyr.pattern.execute, amlyr.pattern, amlyr.name)
knowledge = KnowledgeStateMachine(amlyr.pattern, amlyr.name)

Funded by the European Union

Rule 5. _Monitor State Machine (Rule 15 in Appendix A)



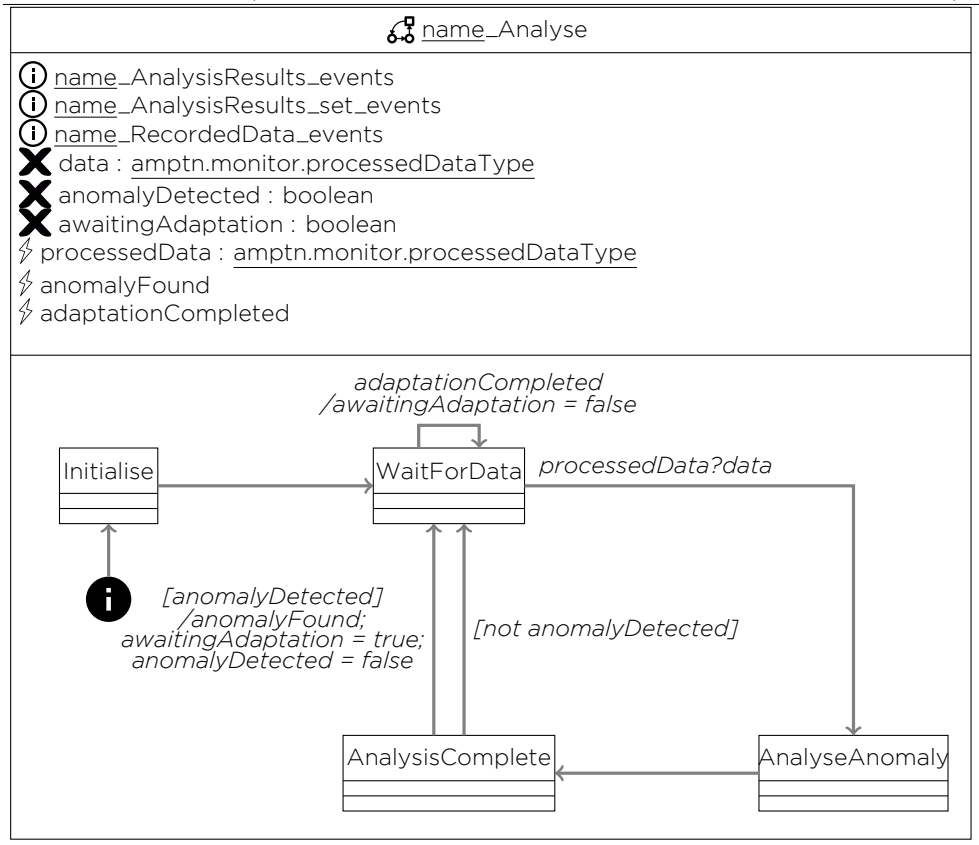
communicated for an error correction technique to be applied. If there is data to send, it should be placed into the *outputData* variable. The guards on the transitions coming out of a junction after the *RecordData* state check the value of the *dataToSend* variable. If *dataToSend* is false, then a transition back to *ReadInput* is taken to allow more data to be obtained. Suppose *dataToSend* is true. In that case, a transition is taken to *SendData*, where further preparation for sending the data may be made (including calculating *outputData*, if not calculated earlier), before a transition back to *ReadInput* is taken. The transition communicates the *outputData* to *_Analyse* via the *processedData* event and then sets *dataToSend* to *false* to ensure the same data is not mistakenly re-sent.

The *_Analyse* machine is defined by the *AnalyseStateMachine* function in Rule 6. It uses the *_AnalysisResults_events* and *_AnalysisResults_set_events* interfaces to allow it to write the results of the analysis. It also uses the *_RecordedData_events* interface, so that it can read (but not write) the data recorded in the *_Monitor* state machine to use in its analysis. It also declares local variables: *data*, to receive the *processedData* coming from the *_Monitor* state machine, *anomalyDetected*, to record whether the analysis detected an anomaly that needs to be signalled, and *awaitingAdaptation*, to record whether an anomaly has been signalled and adaptation in the later MAPLE-K components is being awaited. The *_Analyse* machine use three events to communicate with the other MAPLE-K components: *processedData*, from the *_Monitor* machine, *anomalyFound*, signalled to the *_Plan* machine (or to *_Execute* if *plan* is not present in the pattern), and *adaptationCompleted*,

Funded by
the European Union

Rule 6. _Analyse State Machine (Rule 16 in Appendix A)

AnalyseStateMachine(analyse : AnalyseComponent, amptn : MAPLEK, name : String) : StateMachine =



Funded by
the European Union

which is received from *_Execute* when the adaptation has been performed.

The body of the *_Analyse* state machine begins with the *Initialise* state, just as the *_Monitor* state machine does. After transitioning out of that state, it enters the *WaitForData* state, where it waits for data to be sent from the *_Monitor* state machine via the *processedData* event and stored in the *data* variable. When the data has been received, the state machine enters the *AnalyseAnomaly* state, where application-specific analysis can be performed, and then the *AnalysisComplete* state, from which an anomaly may be signalled. During the analysis, the presence of an anomaly is recorded via the *anomalyDetected* variable, and *AnalysisComplete* has two transitions out of it depending on its value. When *anomalyDetected* is false, the state machine just transitions back to *WaitForData* without any signalling or updates of variables. When *anomalyDetected* is true, the anomaly is signalled via the *anomalyDetected* event, the *awaitingAdaptation* variable is set to *true* to signal that adaptation is in progress. The *anomalyDetected* variable is reset to *false* so that the anomaly is not signalled a second time. It is left to the implementation to determine how the *awaitingAdaptation* variable should be handled in the analysis, whether that is ignoring anomalies when an existing adaptation is in progress, setting higher thresholds for subsequent anomalies, or ignoring *awaitingAdaptation* altogether.

Whether or not *anomalyDetected* is true, the *_Analyse* machine returns to the *WaitForData* state. While there, *_Analyse* may receive an *adaptationCompleted* event from *_Execute*, signalling that the adaptation has finished. This causes the *awaitingAdaptation* variable to be reset to *false*, allowing a new adaptation to begin.

Rule 7 defines the *_Plan* state machine, derived from the *plan* component of the *MAPLEK* pattern. This rule is split into two cases based on whether or not there is an *legitimate* component in the pattern, since its presence means that the possibility of the plan being rejected must be handled. Here, we omit the case where there is no *legitimate* component, since it is similar, and note in our explanation where it differs from the case with a *legitimate* component. It uses the interfaces *_PlanData_events* and *_PlanData_set_events* to allow the plan to be written to the knowledge base, but also *_RecordedData_events*, to provide input for planning, and *_VerificationInfo_events*, to provide input for replanning after a plan is rejected. The state machine also declares events *planningCompleted*, to signal when a plan has been formulated, and *planRejected*, to receive a signal that the plan has been rejected. If the *legitimate* component is not present, the *_VerificationInfo_events* interface and *planRejected* event are not included, since they are involved in communication with the *_Legitimate* state machine. The event used to initiate planning depends on whether or not the pattern has an *analyse* component. If *analyse* is present, then the *_AnalysisResults_events* interface is used in addition to the other interfaces, and the *requestPlan* event is used to receive the signal from the *_Analyse* state machine. If *analyse* is not present, then the signal to begin planning is the receipt of *processedData* directly from the *_Monitor* state machine. Therefore, an event is declared along with a *data* variable to store its parameter.

The body of the *_Plan* state machine begins with an *Initialise* state, as for the other state machines. It then transitions to a *WaitForSignal* state, where it waits for a signal on *requestPlan* or *processedData*, depending on whether the *analyse* com-

Funded by
the European Union

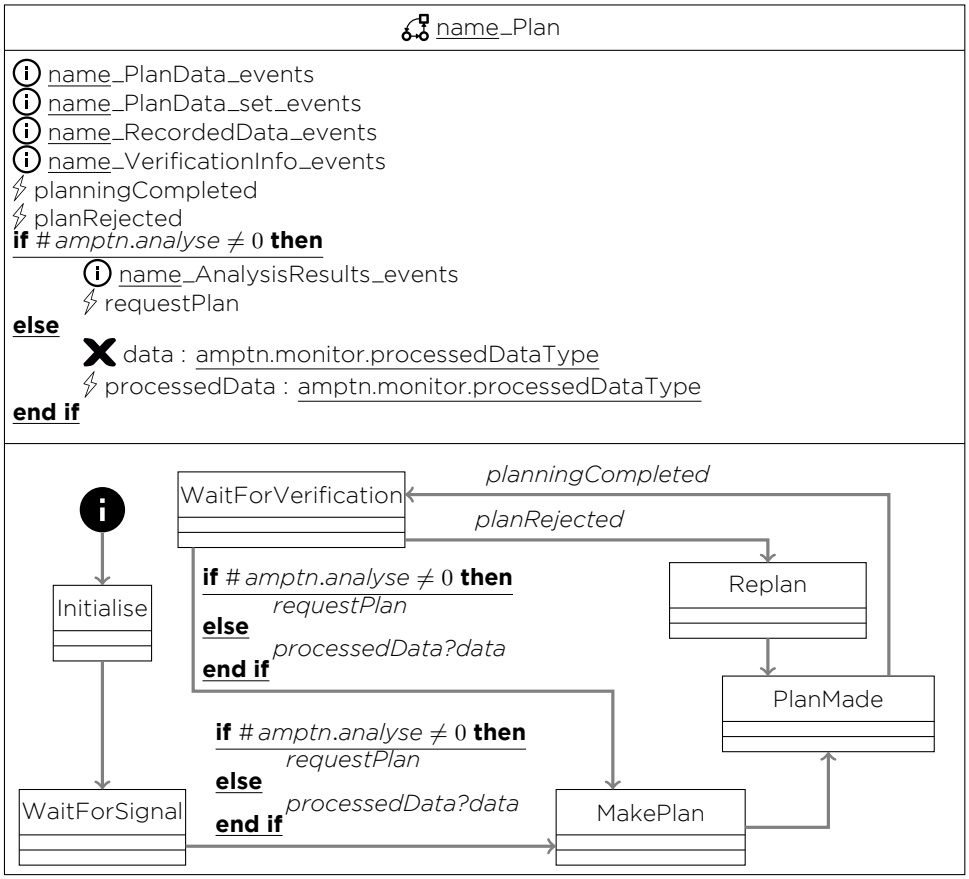
Rule 7. _Plan State Machine (Rule 17 in Appendix A)

PlanStateMachine(plan : PlanComponent, amptn : MAPLEK, name : String) : StateMachine =

if # amptn.legitimate = 0 **then**

...

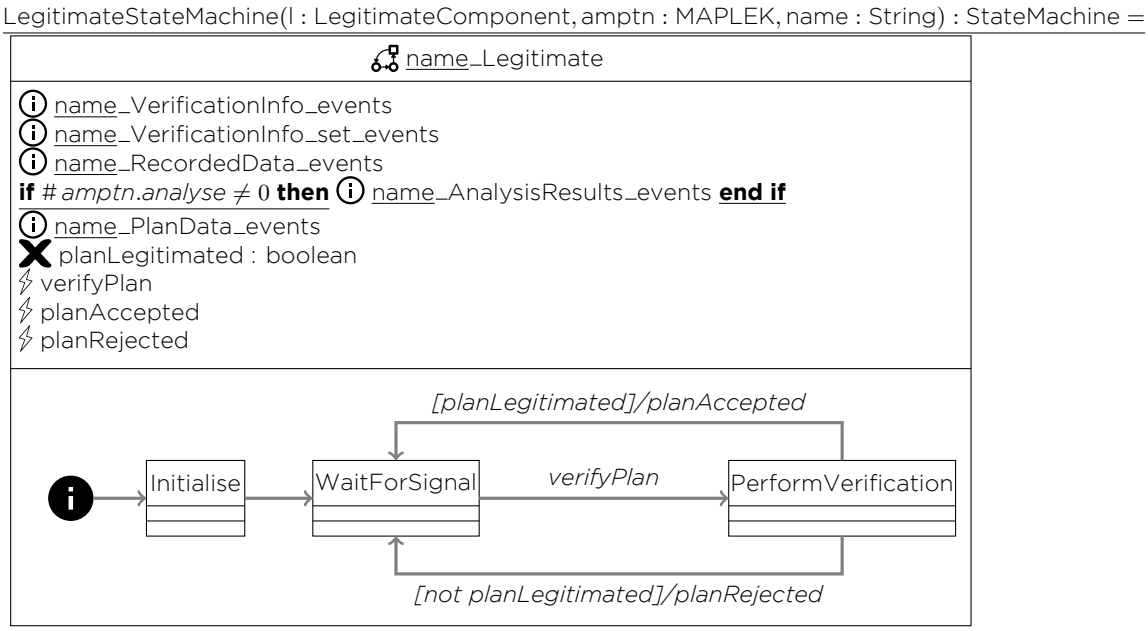
else



end if

Funded by
the European Union

Rule 8. _Legitimate State Machine (Rule 18 in Appendix A)



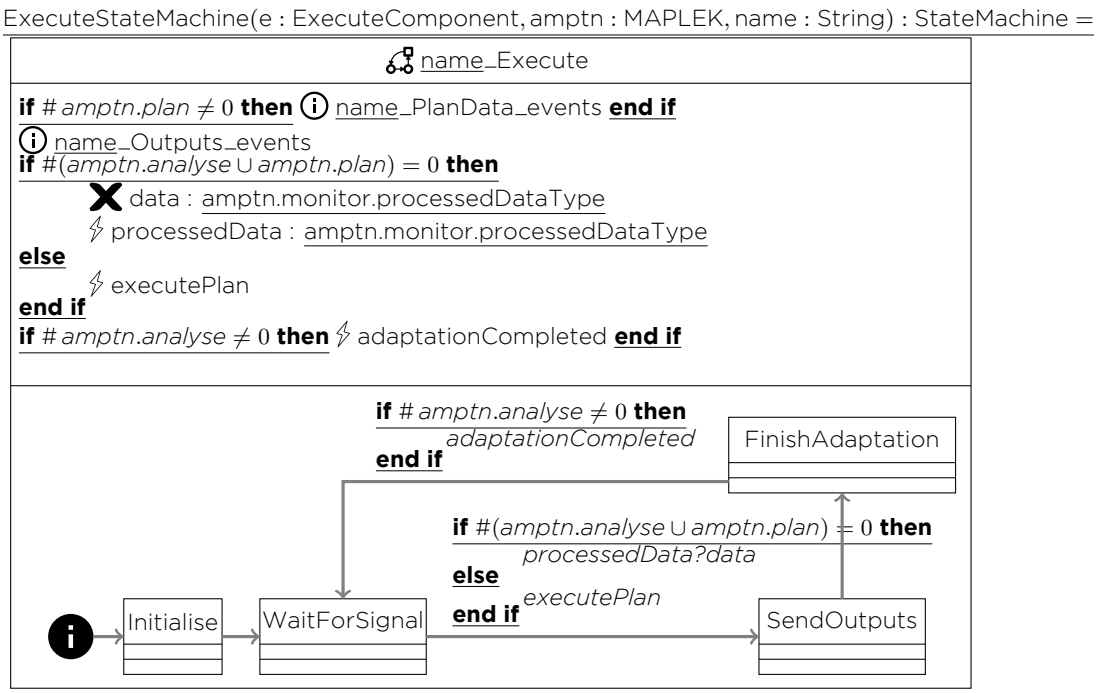
ponent is present or not, before entering the *MakePlan* state. Implementation-specific planning is performed in the *MakePlan* state, before transitioning to the *PlanMade* state, where any post-processing of the plan can be performed. In particular, *PlanMade* is entered after both planning and replanning, so it can include post-processing common to both. When the plan has been made, it is signalled via *planningCompleted* on the transition out of *PlanMade*, and a *WaitForVerification* state is entered. In *WaitForVerification*, the state machine waits for either a signal that replanning is needed via *planRejected*, whereupon it enters *Replan* followed by *PlanMade* to formulate a new plan, or a signal that a plan should be made in a new adaptation loop via *requestPlan* or *processedData*, whereupon it enters *MakePlan* again. If *legitimate* is not present, the *WaitForVerification* and *Replan* states are omitted, and the transition from *PlanMade* instead goes to *WaitForSignal*.

The *_Legitimate* state machine is defined by Rule 8. It uses *_VerificationInfo_events* and *_VerificationInfo_set_events*, to allow information about the verdict given by the legitimation to be recorded in the knowledge base. It also uses the interfaces for the other knowledge base variables to assist in legitimation, although *_AnalysisResults_events* is only included if the *analyse* component is present (*plan* must be present per the well-formedness conditions). A *boolean* variable, *planLegitimated*, is used to record the outcome of legitimation. The *verfiyPlan* event is declared, to signal legitimation should begin, and *planAccepted* and *planRejected* to signal the outcome to the other components.

As with the other state machines, it begins with an *Initialise* state followed by a *WaitForSignal* state. In the *WaitForSignal* state, it waits for a signal via *verifyPlan*, then transitions to *PerformVerification*, where application-specific verification and validation can be performed, with the result recorded in *planLegitimated*. There are two transitions out of *PerformVerification*: if *planLegitimated* is true, then *planAccepted* is signalled, and if it is false, *planRejected* is signalled. Both transitions then

Funded by
the European Union

Rule 9. _Execute State Machine (Rule 19 in Appendix A)



return to *WaitForSignal* to wait for the next request to verify the plan.

The *_Execute* machine is defined by Rule 9. It uses the *_PlanData_events* interface to read the plan, if a *plan* component is present, and the *_Outputs_events* to send the outputs to the managed system. In the absence of a plan, *_Execute* may perform a standard adaptation, or perform some simple planning. It also declares an event to receive the signal to begin executing the plan, which is *processedData*, along with a *data* variable to receive its parameter, if no other components are present to receive that from the *_Monitor* state machine, or *executePlan* otherwise. If an *analyse* component is present, it must be signalled when adaptation is completed so that the *_Analyse* state machine can allow other adaptations to begin, and an *adaptationCompleted* event is included for this purpose.

As with other state machines, *_Execute* begins with *Initialise* and *WaitForSignal* states. In the *WaitForSignal* state, *_Execute* waits for a signal to begin execution of the plan, either via *processedData* (storing its parameter into *data*) if there are no other components to receive it, or via *executePlan* if another component has already received *processedData*. After a signal has been received, *_Execute* enters the *SendOutputs* state where the plan is sent to the outputs in an application-specific fashion, before transitioning to *FinishAdaptation*, where any finalisation of the execution can be added. Afterwards, *_Execute* transitions back to *WaitForSignal*, signalling *adaptationCompleted* if the *analyse* component is present.

The *_Knowledge* machine, which stores the knowledge base, is defined by Rule 10. It uses the interfaces for the knowledge base variables (*RecordedData*, *AnalysisResults*, *PlanData* and *VerificationInfo*), with those derived from optional components conditional on their presence. *_Knowledge* also uses the interfaces containing the events to access these variables, such as *_RecordedData_events* and

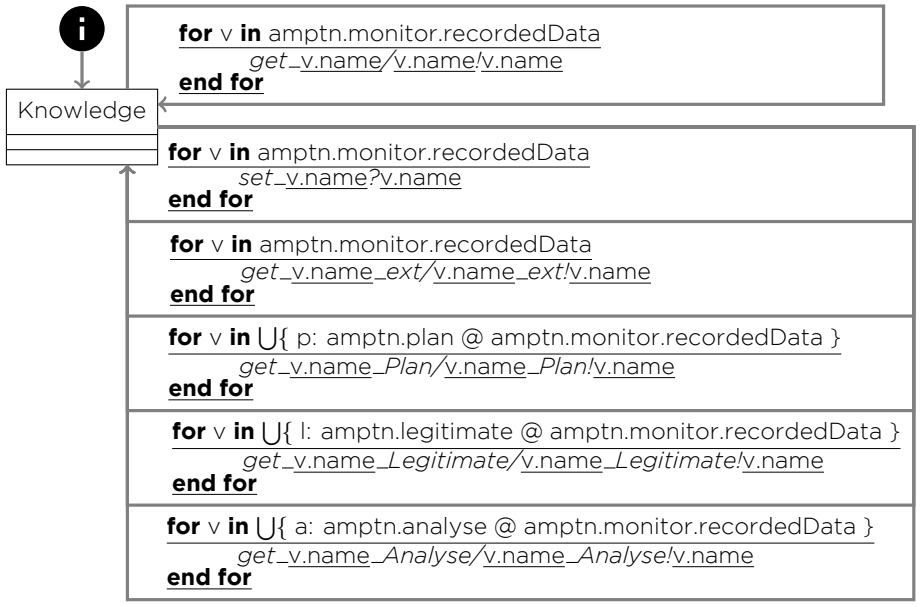
Funded by
the European Union

Rule 10. _Knowledge State Machine (Rule 20 in Appendix A)

KnowledgeStateMachine(amptn : MAPLEK, name : String) : StateMachine =

name_Knowledge

name_RecordedData
if # *amptn.analyse* ≠ 0 **then** name_AnalysisResults **end if**
if # *amptn.plan* ≠ 0 **then** name_PlanData **end if**
if # *amptn.legitimate* ≠ 0 **then** name_VerificationInfo **end if**
name_RecordedData_events
name_RecordedData_set_events
name_RecordedData_ext_events
if # *amptn.analyse* ≠ 0 **then** name_RecordedData_Analyse_events **end if**
...
if # *amptn.analyse* ≠ 0 **then**
    name_AnalysisResults_events
    ...
end if
if # *amptn.plan* ≠ 0 **then**
    ...
end if
if # *amptn.legitimate* ≠ 0 **then**
    ...
end if



if # *amptn.analyse* ≠ 0 **then** KnowledgeAnalysisResultsTransitions(amptn, name) **end if**
if # *amptn.plan* ≠ 0 **then** KnowledgePlanDataTransitions(amptn, name) **end if**
if # *amptn.legitimate* ≠ 0 **then** KnowledgeVerificationInfoTransitions(amptn, name) **end if**

Funded by
the European Union

_RecordedData_set_events. The interfaces for the state machine-specific versions of these are also used, if the corresponding component is present. For example, *RecordedData_Analyse_events* is included when *analyse* is present. Due to the large number of interfaces used, we omit most of them for brevity.

The body of *_Knowledge* has only a single state, *Knowledge*, with transitions for getting and setting each variable included as looping transitions on this state. The transitions for getting the value of the variable are triggered by the event named with *get_* followed by the name of the variable, after which *_Knowledge* outputs the value of the variable via the event with the name of the variable. The transitions for setting the value of the variable are triggered by the event named with *set_* followed by the name of the variable, with its parameter stored into the variable. This asymmetry is because RoboChart does not allow a transition to be triggered by an output. Separate transitions are provided for other components getting the value of a variable, but those for optional components are included by iteration over a flattened set comprehension, so they are not included when the corresponding component is not present. The rule presented here includes the transitions for the *recordedData* variables, and the transitions for other variables are included via functions with the same parameters defined in other rules. We omit these other rules as the transitions defined are similar to those for *recordedData*.

In Appendix A, we present the full set of rules described here.

2.5 Related work

Over the years, several variations and applications of MAPE-K have been proposed [ARS15, PDPB14, BCG+12, BSML18]. For instance, [PDPB14] proposes a dual-layer MAPE-K approach to ensure both localised and global self-management capabilities in middleware for wireless-sensor networks that can dynamically adapt to changing contexts (that is, fluctuating network topologies, limited energy resources, and hostile conditions). The architecture employs two levels of autonomic management: the Sensor MAPE-K Layer manages self-adaptation at the node level (individual sensors); and the Network MAPE-K Layer (NML) oversees the entire network, handling collective adaptation decisions.

The authors in [BCG+12] propose a structural approach to identify the key characteristics of adaptive systems, going beyond traditional behavioural definitions. They emphasise the fundamental role of control data in enabling adaptive behaviour. Here, adaptation is framed as the runtime modification of control data, distinguishing it from static or pre-defined behaviour changes in non-adaptive systems. Beyond the MAPE-K loop, the paper identifies other adaptive patterns, such as internal and external control loops (depending on whether the control logic is embedded within the system or operates externally), reactive adaptation (that is, systems that directly respond to environmental changes without a complex control loop), and towers of adaptation (that is, hierarchical adaptation, where adaptive components manage other adaptive components).

The authors also describe adaptivity in various computational paradigms: Context-Oriented Programming, where adaptation occurs through dynamic context changes affecting code execution; Declarative Programming, where adaptivity is achieved

Funded by
the European Union

by modifying rules or logic clauses at runtime; Rule-Based Systems, where adaptation happens via dynamic rule changes, often supported by reflection or metaprogramming; Concurrency Models (for example, Petri Nets and process algebras), where adaptation is by changes in control tokens or communication patterns; and Reflective Systems, which can modify their own structure and behaviour through meta-level computations.

As far as we know, however, our extension to deal with legitimisation is not considered elsewhere. In addition, our work is unique in that we create a pathway from high-level, albeit formal, descriptions of architectures based on MAPLE-K to deployment architectures and code. 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. In what follows, we review some of the works that have taken advantage of formal modelling and reasoning in conjunction with MAPE-K applications and frameworks.

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. [ARS15] introduces a formal framework for modelling, validating, and verifying self-adaptive systems based on MAPE-K. The authors leverage Abstract State Machines (ASMs), particularly multi-agent ASMs, to specify decentralised and concurrent control mechanisms for adaptation. The framework models the classic MAPE-K loop, explicitly representing each phase as ASM transition rules. The focus is on ensuring functional correctness of self-adaptive behaviours, especially when multiple feedback loops operate simultaneously. The approach enables validation through the simulation of adaptation scenarios using the ASMETA toolset [asm], identifying inconsistencies early in the design phase, and verification via formal model-checking techniques to verify adaptation properties and detect potential interferences or conflicts between feedback loops.

A similar approach is proposed in Camilli et al. [CBC18], where they 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. The framework architecture consists of two key layers. The Base-Level Layer models the managed system and its environment using Petri Nets, capturing the system's static structure and nominal behaviour, including resource allocation and service interactions. The High-Level Layer leverages High-Level Petri Nets (HLPNs) to implement MAPE-K loops for various adaptation concerns. This layer includes an emulator to encode and manipulate the Base-Level Layer, an API for defining read/write operations (sensors and actuators) that interact with it, and a Managing Subsystem that runs multiple MAPE-K loops, each addressing specific adaptation goals, such as, energy efficiency and performance optimization. A key novelty is the use of HLPNs to represent structural changes and dynamic reconfiguration in self-adaptive systems, which enables the detection of undesired behaviours, such as conflicting adaptations and inconsistent system states.

Finally, Weyns et al. [WCM+23a] introduces ActivFORMS (Active FORmal Models

Funded by
the European Union

for Self-adaptation), an end-to-end, formally grounded, model-driven approach to engineering self-adaptive systems. ActivFORMS addresses key challenges in ensuring correctness, efficiency, and flexibility, especially under runtime uncertainties, such as fluctuating workloads and resource availability. The approach spans design, deployment, runtime adaptation, and evolution, providing formal guarantees at each stage.

The authors also present ActivFORMS-ta, a tool-supported instance of ActivFORMS that leverages timed automata and statistical model checking to maintain system correctness at runtime, enabling efficient adaptation through statistical model checking (SMC), and real-time verification without exhaustive state exploration. The authors validate ActivFORMS through DeltaIoT, an Internet of Things (IoT) application for smart-building security monitoring, deployed at KU Leuven. The system consists of 15 IoT motes strategically placed for tasks such as access control, temperature monitoring, and occupancy detection. The key adaptation goals of the system include maintaining packet loss below 10%, minimizing energy consumption, and dynamically adding new goals during operation, such as reducing latency to improve system responsiveness.

The use of legitimisation is relevant in the context of all these works, for enhanced autonomy and safety. We provide here a precise account of the MAPLE-K architecture, including its formal semantics at both conceptual and deployment levels. This paves the way for integrating our results within the frameworks just described.

2.6 Final considerations

We have presented in this chapter a formal conceptual account of the RoboSAPIENS MAPLE-K architecture and its variations. This is a precise description of the architecture that, at the same time, enables the definition of adaptive systems that adopt that architecture. With these definitions, we enable formal verification via RoboChart and its semantics, as well as a connection to AADL, for the definition of realisation details of an implementation. Verification is also discussed in this deliverable. Connection to AADL is presented in [BvAK+25] and in Deliverable 5.2.

Funded by
the European Union

3 RoboChart with neural networks

This chapter presents our extension of RoboChart to support components whose behaviour is defined by a neural network. These ANN components allow the neural network model of computation to be part of a system of interacting components. Inclusion of these components in RoboChart allow reasoning about systems with neural networks rather than focusing on a single network.

We provide an overview and an example RoboChart model with an ANN component in Section 3.1. Next, we describe our extensions to RoboChart's metamodel and well-formedness conditions for ANN components in Section 3.2. We then provide a denotational semantics for ANN components in Section 3.3. Related work is mentioned in Section 3.4. Section 3.5 concludes with a summary.

This work has been published recently in [AYF+25].

3.1 Overview and example

In our work, we extend RoboChart with a new form of controller defined by an ANN block. In Fig. 8, we present a RoboChart module for a Segway robot that includes an ANN component AnglePIDANN. This module, called Segway, contains a robotic platform SegwayRP, a standard controller SegwayController, and an ANN controller AnglePIDANN. SegwayRP has events representing data provided by the segway sensors and operations to control movement via the segway motors. SegwayController describes the behaviour of this system, defined through three state machines: BalanceSTM, RotationPID and SpeedPID.

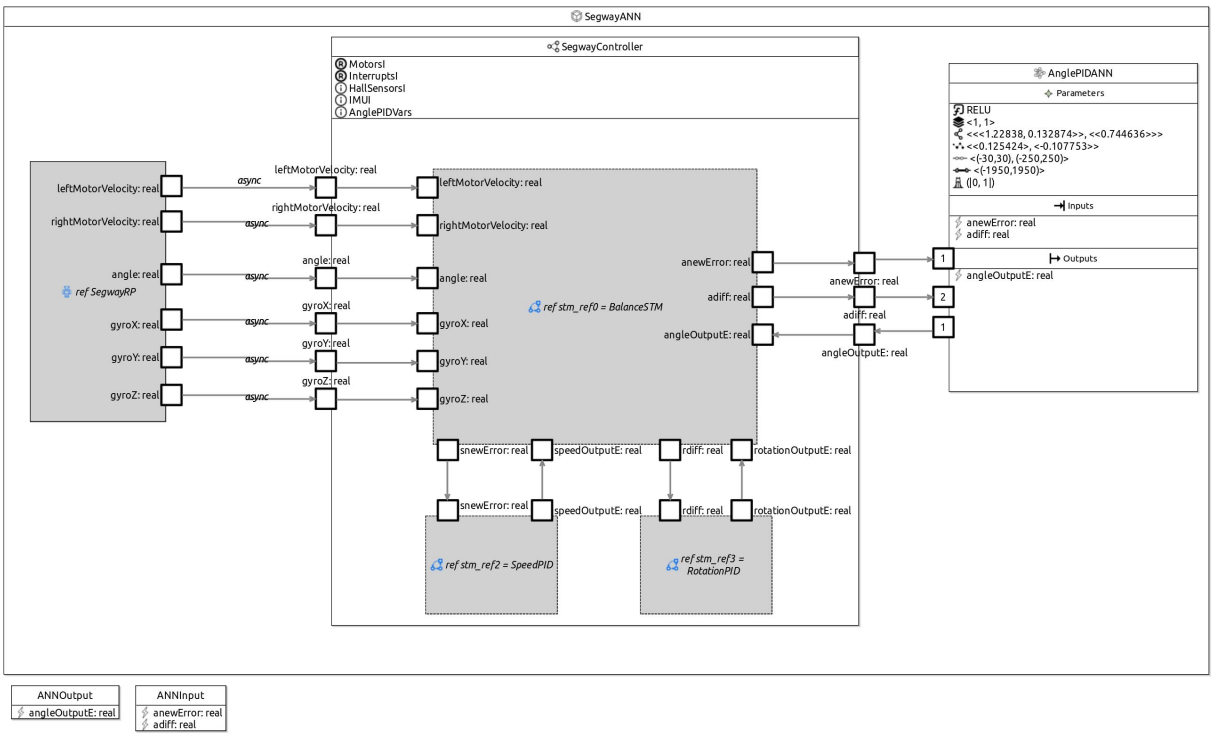


Figure 8: An example RoboChart module containing an ANN component.

As shown in Fig. 8, the block SegwayController comprises three blocks that represent references to its state machines, as described in [BC19]. SegwayController exhibits a cyclic behaviour defined by BalanceSTM, which updates the motor speeds using the outputs of the PID machines and of the AnglePIDANN controller to maintain the segway's upright position. In the original version of this example, there is a third state machine, AnglePID, which specifies a standard PID to control the angle of the segway. In our version here, we have an ANN instead, with the same interface. Just like AnglePID, the ANN component AnglePIDANN accepts as input the events anewError and adiff and communicates its output through the event angleOutputE.

The block for an ANN component (marked using the symbol ⚛) has its behaviour defined by some parameters defined in three blocks. First, in the *Parameters* block, we define the activation function (*ReLU*, in the example), the layer structure, the weights as a triple nested sequence, and the biases as a double nested sequence.

Next, in the *Inputs* block, we define the events, or interfaces containing events, that the *ANNController* treats as inputs. In this example, we use a single defined interface, *ANNInput*, that contains two events, *adiff* and *anewError*, defined in a block outside the *SegwayANN* module. Lastly, we define an *Outputs* block that defines the events the *ANNController* uses as outputs.

We define an *ANNController* as a cyclic controller: it waits to receive communication from all input events in any order, then, when it has received them, it engages in its output events in an arbitrary order, then repeats. This controller style can never choose to terminate itself, but it will terminate when the system does.

We can model the RoboSAPIENS case studies involving ANN components. In Figure 9 we show an in-development RoboChart model representing the screen unscrewing aspect of the DTI case study, with an ANN component to replace a particular service of the example. The case study involves a robot arm designed to unscrew laptop screens.

The module for our RoboChart model is *DTISoftware*. As shown in Figure 9, it contains three blocks: *IKANN*, *UnscrewingController*, and *RobotArmRP*. Here, *UnscrewingController* is an incomplete (as shown with the ellipsis) controller designed to perform the unscrewing task. It has a single state machine reference (*UnscrewingSTM*) and requires a single interface (*JointsI*) representing operations to adjust all six joints of the robot arm. The robotic platform (*RobotArmRP*) provides the required interface *JointsI* to the system.

IKANN models an ANN component that is trained to replace an inverse kinematics service. Given Cartesian coordinates (captured by the events *x*, *y*, and *z*) of the desired position of the end effector of the robot arm, it computes the angle that each joint must be in to ensure that the end effector reaches the given coordinates. The output events of this ANN are *j1* to *j6*, representing the angle in radians that the joints should be in to reach the desired coordinates. The ANN has two hidden layers, each with 128 nodes, and both use the ReLU activation function; we define these parameters through a given file, `ikann_params`, because defining the weights in the component itself would be too verbose.

Next, we present the metamodel and well-formedness conditions of ANNControllers.

Funded by
the European Union

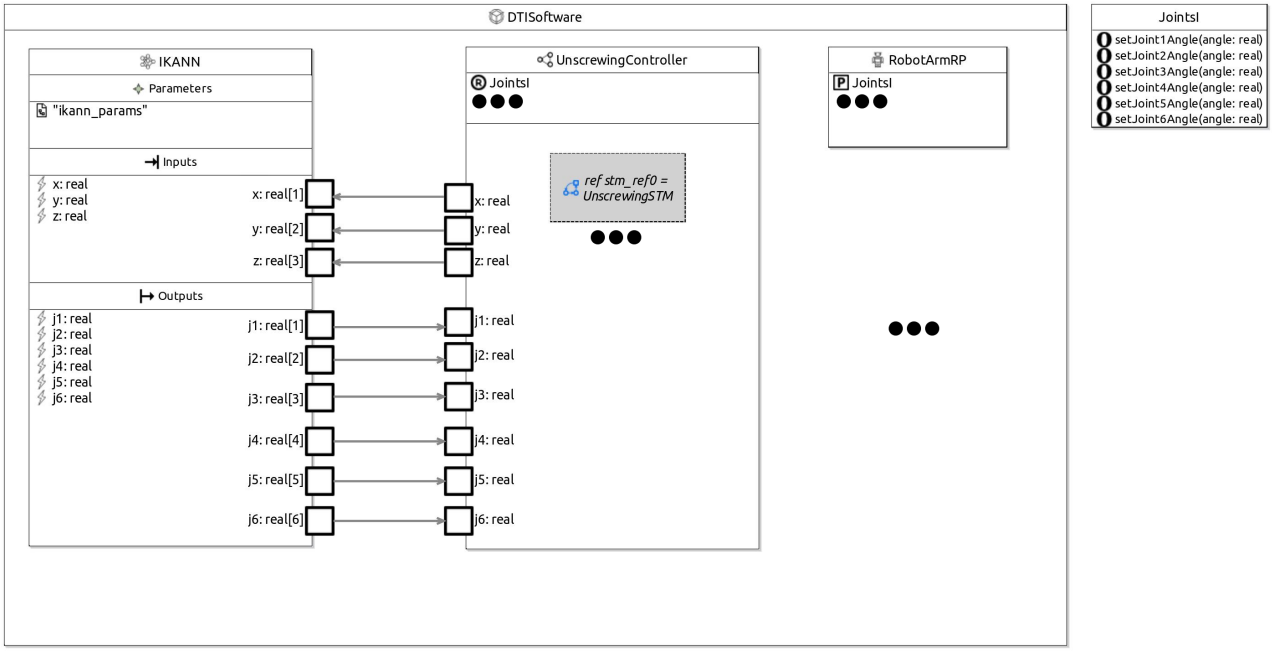


Figure 9: A sketch of the RoboChart module for the DTI case study. Here, the ellipsis means that the model is incomplete. The parameters of th ANN (*IKANN*) are defined in a file named `ikann_params`.

3.2 Metamodel and well-formedness conditions

Our extension of RoboChart adds a few classes to its metamodel, shown in Figure 10. They introduce the concept of an *ANN* as an abstract class. Instances of *ANN* can be a RoboChart controller—our focus here—or an operation. Figure 10 shows *ANNController*, which inherits from another class, *GeneralController*, omitted in Figure 10. A controller in RoboChart is typically used to represent functionality allocated to a computational unit or a self-contained architectural component. Our extension allows a controller to be defined by an ANN.

An *ANN* includes *ANNParameters* to define the hyperparameters and the trained parameters of an ANN component, as shown in Figure 10. (The complete metamodel is in Appendix B.) The activation function is given using an enumerated type. Here, we give semantics only to an ANN that uses ReLU, which is suitable for using Marabou in reasoning. The use of our semantics in conjunction with other functions is straightforward. Moreover, given the nature of process algebra, explicitly devised to model networks (of processes), our overall approach is well suited for exploration of any ANN structure. This will be part of future work.

We capture the input layer, including its size, through an *inputContext* and the output layer by an *outputContext*. Such a context can be used to define, possibly via interfaces, input and output events to connect the *ANN* to other RoboChart components. *Context* is an existing RoboChart concept used to define the interface of every component. Each *Event* in an *inputContext* or *outputContext* must be of a new class, *OrderedEvent*, which adds an integer to an *Event* definition to specify an order for the inputs and outputs of an ANN component.

Funded by
the European Union

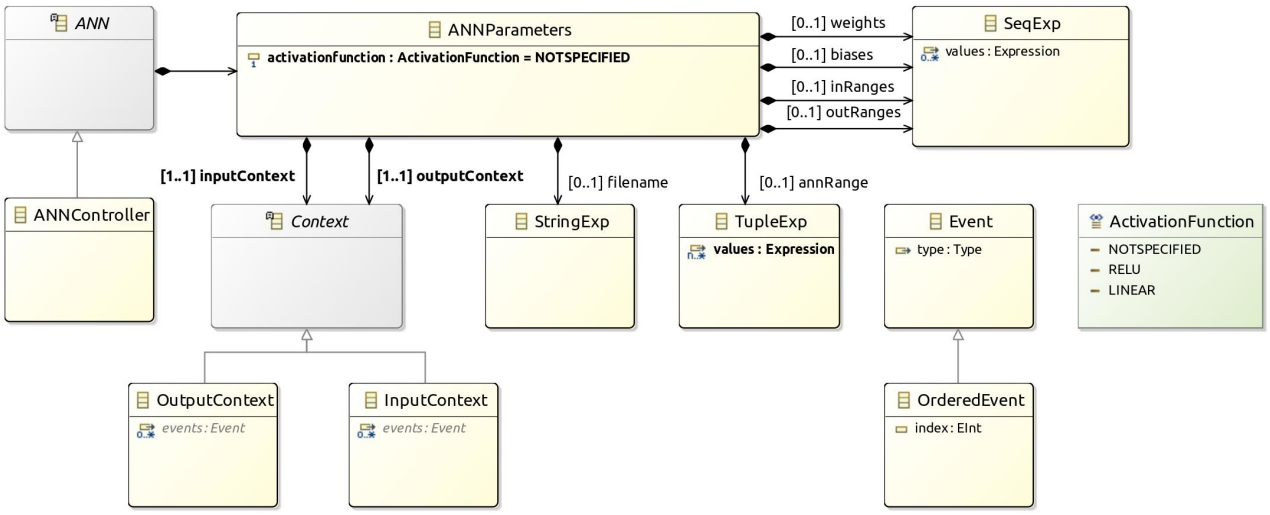


Figure 10: Metamodel for ANN components in RoboChart: classes in grey are abstract, attributes in grey are inherited, and attributes in bold are not optional.

We capture the trained parameters through references *weights*, a three-dimensional tensor (represented as a triple-nested sequence), and *biases*, a matrix (a double-nested sequence). The class *SeqExp* captures sequence expressions.

The range to which an ANN is normalised is captured through a pair *annRange*. (Typically, this range is between -0.5 and 0.5, or 0 and 1.) We capture the range that each input value can take through *inRanges*, and the range for outputs with *outRanges*: both are sequences of pairs. Normalisation is a common consideration when defining an ANN; it involves scaling all input ranges in the training data to a new range with a mean close to 0 [LBOM98].

The reference *filename* supports using ANN parameter files (using formats such as ONNX, for instance) instead of explicitly defining parameter values.

Our well-formedness conditions for *ANN* are listed in Appendix B.1. They assert that the parameters are defined either in the model itself or in a linked file. They also ensure that the trained parameters' size and shape correspond to the hyperparameters and that the size of the normalisation sequences matches the number of inputs and outputs to the component, and, for each element in these sequences representing a range, the maximum is strictly greater than the minimum. Finally, at the RoboChart model level, we ensure that connections to and from the ANN component are in accordance with its definition of input and output events.

Next, we present the semantics of these well-formed ANN components.

3.3 Semantics: overview and rules

This section presents our denotational semantics. We introduce *Circus* and then provide an overview of the semantics and present key definitions. The complete definition of all semantic rules is given in Appendix B.2.

Funded by
the European Union

3.3.1 Circus

Circus [Oli06] is a process-algebraic language for the specification of concurrent systems; it defines data models and control behaviour of independent components: processes. State declarations and actions define their behaviour; every process has one main action that defines its behaviour. Actions are similar to CSP processes [Hoa85] but define stateful behaviour. The definition of a basic *Circus* process has the form: **process** *Proc* $\widehat{=}$ **begin** ... • *A* **end**. Here, the process is named *Proc* and has main action *A*. The body of the definition, between **begin** and **end**, contains declarations of state variables and actions that are local to *Proc*. We describe selected *Circus* action operators used in our semantics in Table 1; further details are provided as needed. A *Circus* process can also be defined in terms of other processes, using process operators similar to CSP's. We do not use a composition of processes here, but it is through these operators that we can combine the semantics of an ANN controller, which is a *Circus* process, with those of other components in a RoboChart model.

The semantics of *Circus* [Oli06] is defined using Unifying Theories of Programming [HJ98a] (UTP). It defines processes and actions as reactive contracts [FCC+20]: alphabetised predicates that capture their reactive behaviour. The UTP-based reactive contracts semantics is ideal for supporting theorem proving.

3.3.2 Overview

As mentioned earlier, the semantics of an ANN controller is a *Circus* process. Figure 11 depicts the structure of the main action of such a process. We capture every node of the ANN as a *Circus* action, and then define each layer as the parallel composition of these actions, with the whole ANN being the parallel composition of the layer actions. We treat the input nodes as input for the first layer. The parallel composition of node actions, represented by the parallel lines between them in Figure 11, and the parallel composition of layer actions define the expected data flow.

The Exchange of information between nodes, represented by the lines connecting node actions in Figure 11, is captured by CSP events on a channel *layerRes*. The event *layerRes.l.n* is for communication from the *n*th node of the *l*-th layer. A *layerRes*.0.*n* event represents the *n*-th input of the ANN, and *layerRes.layerNo.n*, the *n*-th output; here, *layerNo* is the number of layers.

Each node action is defined by the parallel composition of actions that receive communications from the previous layer, passing this information, after applying the node's weight, to another action that calculates the node's overall output. Intra-node action communications are via a channel *nodeOut*. A *nodeOut.l.n.i* event is for the *i*-th input of the node *n* in layer *l*.

3.3.3 Semantic Rules

We formalise our semantics via a set of rules that together define a function $[\![C]\!]_{ANN}$ from a RoboChart ANN controller *C* to a *Circus* program including some channel and constant declarations, and a process. This semantics fits into the definition of a process formalising a RoboChart model that includes the ANN component as

Funded by
the European Union

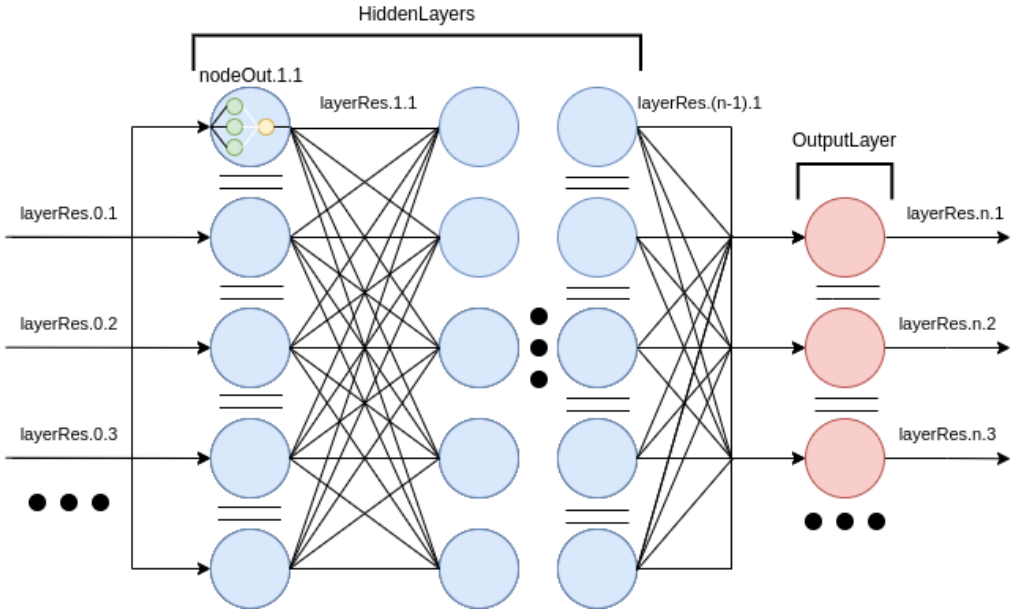


Figure 11: A diagram showing the structure of our semantics of ANN controllers. Circles represent actions representing nodes, and labelled edges represent communications. The parallel lines denote parallel composition between actions. We have parallel composition at both the node and layer levels. The ellipses indicate that we allow for an arbitrary hidden layer structure and support any input and output layer size. In the top left-hand corner, we also show the internal structure of a node action.

Rule 1. Semantics of ANN Components $[\![\text{c} : \text{ANNController}]\!]_{\mathcal{ANN}} : \text{Program} =$

ANNChannelDecl(c)

ANNConstants(c)

ANNProc(c)

specified in [ACW23]. Here, we provide an overview of these rules; the complete set can be found in Appendix B.2.

A rule definition consists of a number and a brief description, followed by the declaration of the function defined by the rule and an expression in a metalanguage that specifies the function. In that expression, elements of the metalanguage are underlined. Our top-level rule (Rule 1), specifying an element of the top-level *Circus* syntactic category *Program*, is defined by three functions: *ANNChannelDecl*, which specifies channel declarations (as described above); *ANNConstants*, which specifies constants; and *ANNProc*, which gives the behaviour of the process. We define *ANNProc* in Rule 2; the complete definitions of *ANNChannelDecl* and *ANNConstants* are omitted here but can be found in Rules 3 and 4 of Appendix B.2.

The rule *ANNConstants* first records constants corresponding to attributes of *C*: *weights*, *biases*, *annRange*, *inRanges*, *outRanges*, and *layerstructure*. Further, the rule defines constants for the function *relu* (the activation function), the input to

Funded by
the European Union

each layer *layerInput* (derived from the controller *C*), and the normalisation functions *norm*, *normIn*, and *denormOut*. Our semantics defines a normalised ANN, so we normalise every input using the function *normIn*, then denormalise every output using *denormOut*. These functions are defined as *norm*, which scales a value from one range to another.

The constants *layerstructure* and *layerInput* give the shape of our *Circus* semantics; for our example, these take the values as shown below. We obtain these constants from the ANN component in RoboChart, AnglePIDANN: we present a graphical representation of this component in Figure 13. We use these constants to illustrate our semantic rules throughout this section.

$$\mathit{layerstructure} = \langle 1, 1 \rangle$$
$$\mathit{layerInput} = \langle 2, 1, 1 \rangle$$

The process, an element of the *Circus* syntactic category *ProcDecl*, is defined by the function *ANNProc(C)* defined in the Rule 2 presented here. That process is named according to the *name* attribute of *C*. Its main action is CircANN(C), shown after the ●. This action uses the local actions *ANN* and *Interpreter*, which capture the data flow of the ANN, to define its behaviour within the RoboChart context: using the input and output events of the RoboChart model, dealing with normalisation, and handling termination.

For example, we consider the RoboChart module *SegwayANN* partially shown in Figure 13. It has two controller blocks: *SegwayController* defined by a state machine, and *AnglePIDANN* defined by an ANN. We give the complete semantics of *AnglePIDANN* in Figure 12, which was generated using Rule 2. In the semantics of *AnglePIDANN*, the action *ANN* captures the behaviour of the ANN in terms of its hyperparameters and its trained parameters. The action *Interpreter* captures normalising all input communications to the ANN, then denormalising all output communications from the ANN. Using these actions, we define the main action of the process for *AnglePIDANN* in Figure 12 using parallel composition ($A \llbracket \mid cs \mid \rrbracket B$), hiding ($\setminus$), and interrupt ($\Delta$).

The parallel composition of *Interpreter* and *ANN* captures the behaviour of our ANN in terms of the input and output events of the controller. The definition of the simple *Interpreter* action is determined by the semantic function Interpreter(C), which is omitted here. *Interpreter* takes inputs for the controller in any order and outputs their normalised values, in any order, via *layerRes*.0 events to *ANN*. It also takes outputs from *ANN* via *layerRes*.layerNo (where layerNo is the index of the last layer) events and outputs their denormalised values. Figure 12 presents the *Interpreter* action for our example; here, we have two inputs (*anewError_in* and *adiff_in*), and one output (*angleOutputE_out*). *Interpreter* first behaves as the interleaved composition of two sub-actions, each of which accepts an input event and then communicates the normalised (with *normIn*) value of this event to a *layerRes*.0 event. Next, *Interpreter* waits on the single output event of the *ANN* action (*layerRes*.2.1); when received, it outputs the denormalised (with *denormOut*) value on the *angleOutputE_out* channel. Finally, *Interpreter* repeats and waits for fresh input events.

In *Circus*, $A \llbracket \mid cs \mid \rrbracket B$ defines the parallelism of actions *A* and *B*, which can per-

Funded by
the European Union

Rule 2. Function ANNProc $ANNProc(C) : ProcDecl =$

process C.name $\widehat{=}$ **begin**

$Collator \mathrel{\widehat{=}} l, n, i : \mathbb{N}; sum : Value \bullet$

$\quad (i = 0)\ \&\ layerRes.l.n\,!(relu(sum + (biases(l)(n)))) \longrightarrow$ **Skip**

$\quad \Box$

$\quad (i > 0)\ \&\ \ nodeOut.l.n.(layerInput(l) - i + 1)\,?x \longrightarrow$

$\qquad\qquad Collator(l, n, (i - 1), (sum + x))$

$Edge \mathrel{\widehat{=}} l, n, i : \mathbb{N} \bullet$

$\quad layerRes.(l - 1).i\,?x \longrightarrow nodeOut.l.n.i\,!(x * (weights(l)(n)(i))) \longrightarrow$ **Skip**

$Node \mathrel{\widehat{=}} l, n, inpSize : \mathbb{N} \bullet$

$\quad ((;\, i : 1 \,..\, inpSize \bullet Edge(l, n, i))$

$\qquad [\![\mid \{\!| nodeOut.l.n |\!\} \mid]\!]$

$\quad Collator(l, n, inpSize, 0)) \setminus \{\!| nodeOut.l.n |\!\}$

$HiddenLayer \mathrel{\widehat{=}} l, n, inpSize : \mathbb{N} \bullet$

$\quad ([\![\{\!| layerRes.(l - 1) |\!\}]\!]\ i : 1 \,..\, s \bullet Node(l, i, inpSize))$

$HiddenLayers \mathrel{\widehat{=}}$ HiddenLayers(C)

$OutputLayer \mathrel{\widehat{=}}$ OutputLayers(C)

$ANN \mathrel{\widehat{=}}\ \ (HiddenLayers [\![\mid \{\!| layerRes.(\text{layerNo(C)} - 1) |\!\} \mid]\!] OutputLayer)\ ;\ ANN$

$Interpreter \mathrel{\widehat{=}}$ Interpreter(C)

$\bullet$ CircANN(C)

end

Funded by
the European Union

process *AnglePIDANN* $\widehat{=}$ **begin**

$\ldots$

$HiddenLayers \mathrel{\widehat{=}} HiddenLayer(1,1,2)$

$OutputLayer \mathrel{\widehat{=}} [\![\{\!| layerRes.1 |\!\}]\!] i : 1..1 \bullet Node(l,i,1))$

$ANN \mathrel{\widehat{=}} (HiddenLayers [\![| \{\!| layerRes.1 |\!\} |]\!] OutputLayer) \,;\, ANN$

$Interpreter \mathrel{\widehat{=}}$

$(anewError\_in?x \longrightarrow layerRes.0.1.(normIn(1,x)) \longrightarrow \textbf{Skip} \;|||$

$adiff\_in?x \longrightarrow layerRes.0.2.(normIn(2,x)) \longrightarrow \textbf{Skip});$

$layerRes.2.1?y \longrightarrow angleOutputE\_out.(denormOut(1,y)) \longrightarrow \textbf{Skip};$

$Interpreter$

- $((Interpreter [\![| \{\!| layerRes.0, layerRes.2 |\!\} |]\!] ANN) \setminus \{\!| layerRes |\!\})$

  $\triangle\, terminate \longrightarrow \textbf{Skip}$

end

Figure 12: *Circus* semantics for the ANN controller *AnglePIDANN*, generated via an application of Rule 2. The ellipsis (...) denotes that the definition of the *Collator*, *Edge*, *Node*, and *HiddenLayer* actions are constant in Rule 2, so are omitted here.

form any events outside of *cs* set independently but must engage on any event in *cs*. Here, *Interpreter* and *ANN* synchronise on the set containing all *layerRes*.0 and *layerRes*.layerNo. We hide all communications on *layerRes*, so *Proc* defines interactions over the inputs and outputs of the controller, as expected: *anewError_in*, *adiff_in*, *angleOutput_out*.

Finally, the main action of an ANN controller process can be interrupted (operator $\triangle$) by an event *terminate*, raised if all the controllers of the RoboChart terminate, and so the whole software, including the ANN, also terminates (**Skip**).

ANN is defined by the parallel composition of actions *HiddenLayers* and *OutputLayer*, synchronising on *layerRes*.(layerNo(C) − 1) events, and sequentially composed with *ANN* in a tail recursion. *HiddenLayers* and *OutputLayer* are defined by semantic functions (omitted here) that compose in parallel actions for the hidden layers and the output layer's nodes. *HiddenLayer* has parameters *l*, *n*, and *inpSize* and captures the semantics of the *l*th layer with *n* nodes and *inpSize* inputs coming from the previous layer or to the ANN as a whole. In our example, shown in Figure 12, the ANN component has only a single hidden layer, so *HiddenLayers* is simply $HiddenLayer(1,1,2)$, denoting the 1st layer with 1 node and 2 inputs. We can derive the number and size of the hidden layers using the *layerstructure* sequence, and we know the input size of each layer through either that sequence or the number

Funded by
the European Union

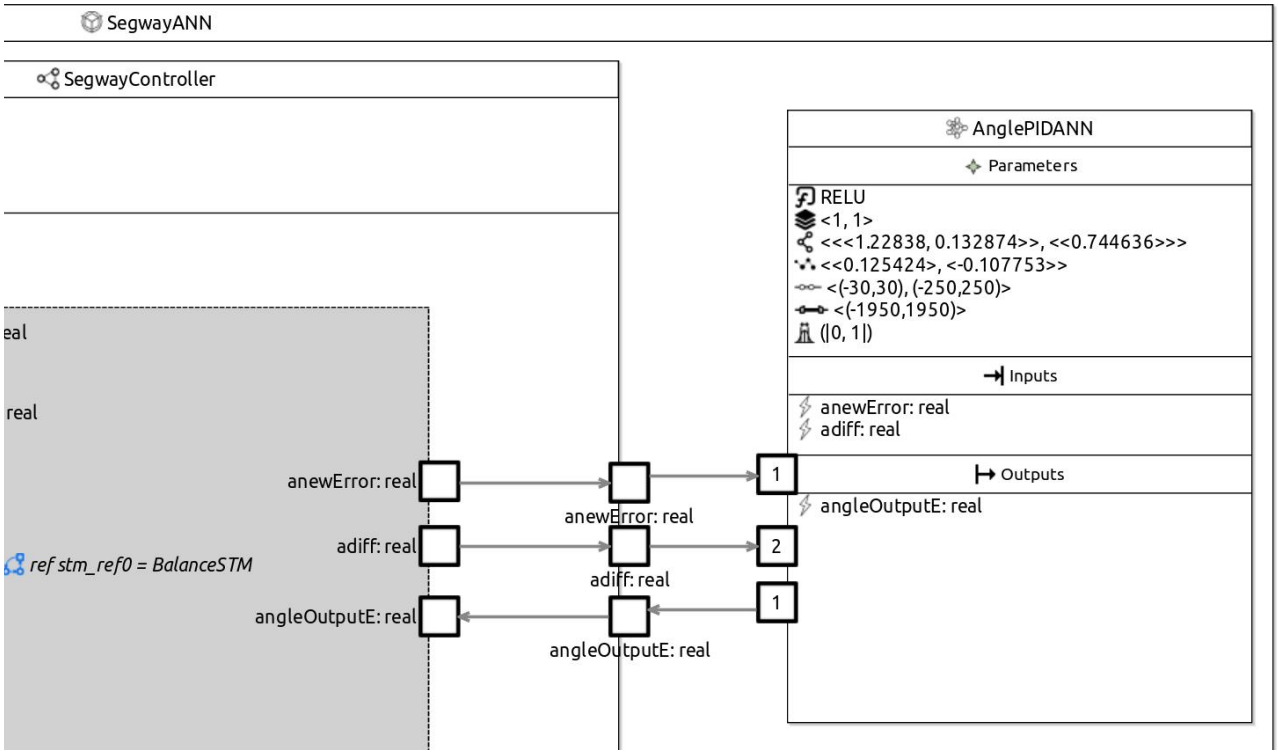


Figure 13: Part of RoboChart module with an ANN for segway control defined in the block *AnglePIDANN*. The first parameter is the activation function, ReLU. The second parameter is the layer structure: $\langle 1, 1 \rangle$. The third and fourth parameters are the trained parameters: the weights and biases. The remaining three parameters are associated with normalisation: the *inRanges*, the *outRanges*, and the *annRange*.

of elements in the input context—the input size. We define *OutputLayer* as a layer with no parameters; in our example, see Figure 12, *OutputLayer* is the distributed parallel composition of just 1 *Node* action, because our example has one node in its output layer (the last element of *layerstructure* denotes the size of the output layer). The definition of this action is similar to that of *HiddenLayer* in Rule 2.

The definition of *HiddenLayer* uses a replicated alphabetised parallelism $[\![cs]\!]\, i : T \bullet A(i)$ composing actions $A(i)$ in parallel, with i ranging over T synchronising on all events in the set cs. For *HiddenLayer*, the index i ranges from 1 to n, composing n processes $Node(l, i, inpSize)$ which capture the semantics of the ith node of the layer l. Synchronisation is on the $layerRes.(l-1)$ events representing the inputs to the layer (see Figure 11).

$Node(l, n, inpSize)$ is defined by the replicated sequential composition $(;)$ of $inpSize$ actions $Edge(l, n, i)$, in parallel with a *Collator* action. *Edge* actions collect inputs from the $layerRes.(l-1)$ channels providing those values to *Collator* after applying the *weights* via the *nodeOut* channel. *Collator* sums its inputs to define the node output, communicated via $layerRes.l$. This output includes the bias and reflects the output of the activation function *relu*. This action uses a one-based index for *nodeOut* events to match the sequential composition of *Edge* actions; we use the sequence *layerInput* to define this index, where $layerInput(l)$ is the size of layer $l-1$,

Funded by
the European Union

or the input size for the first layer. The definitions of *HiddenLayer*, *Node*, *Edge*, and *Collator* are identical for every ANN.

The automatic generation of the semantics of a RoboChart model that includes an ANN component is already implemented as described in Chapter 6.

3.4 Related work

We consider two related work categories: component-level and system-level verification. We have discussed component-level verification in [ACW20] and [ACW23]. Here, we will discuss other approaches to capturing larger systems that involve neural network components. Furthermore, we will discuss work focused on enabling formal verification and the differences in what they can capture.

The first approach we will discuss is the work presented in [IJH+21]. They present ANN components in the context of Simulink diagrams, a tool for modelling software systems. They propose training ANN components to replace Simulink controllers. Further, they discuss how their approach can enable formal reasoning using the Simulink Design Verifier [1]. However, they do not present formal semantics for the ANN components, and they mention that this verification is limited.

The approach in [IJH+21] enables code generation via C, and they discuss how to utilise the CMBC model checker[2]. Using CMBC enables them to prove system-level properties obtained from the Simulink diagram for the code that implements an ANN component. The authors comment that this verification approach is challenging to scale for realistic properties. While this approach is based on models used for specification, they are not formal models, so the formal material is not application-independent, but rather specific to the case study considered.

The approach in [IJH+21] provides pre- and postconditions of components in automata. These conditions can be used to verify the system-level properties of AI-enabled software. The authors discuss a framework for task splitting, where each task is subdivided into smaller tasks, an ANN component is learnt for every task, and the entire system is verified. The method is compositional, that is, it includes training a separate ANN component for each sub-task and contains a method to support compositional verification of the resultant system of ANN components. Due to the type of formalisation, the approach is designed to capture the physical dynamics of a robotic system, rather than its reactive properties.

VEHICLE [DKA+24] is a domain-specific language designed for the high-level specification of ANN components. The VEHICLE language is based on functional programming, λ-calculus, with support for arithmetic, vectors, and logic. The authors translate this language to a format that Marabou can read via the approach detailed in [DAK+23]. This approach does not provide an explicit model of the system's behaviour, which is provided on a case-by-case basis in a separate interactive theorem prover (the authors mention Agda [LB06] as an example). Using the interactive theorem prover Agda would allow them to prove system-level properties.

[1] uk.mathworks.com/products/simulink-design-verifier.html
[2] cprover.org/cbmc/

Funded by
the European Union

The VEHICLE compiler also translates the specification to a loss function that TensorFlow can use to train an ANN based on this specification.

Our work is distinct in that it can establish system-level reactive properties of systems involving ANN components. We consider systems based on abstractions of system communications: events and channels. We design specifications that formally characterise a system's intended interaction with the environment and its users. This allows us to design specifications that capture timing, probability, and reason about a system's reactivity and availability for interaction.

3.5 Final considerations

In this section, we have presented an extension and mechanisation of RoboChart with novel ANN components. This work enables automated reasoning about the reactive behaviour of software systems with AI components. To the best of our knowledge, it is also the first work to support this style of verification. This work is specifically tailored to embedded design and verification of software for robotics using the RoboChart framework. Verification is the topic of the next chapter.

In the next chapter, we discuss how, given the formal semantics of ANN components as presented here, we can enable formal verification via theorem proving for properties of software involving ANN components.

Funded by
the European Union

4 Verification with RoboChart and ANN

Our semantics of an ANN component uses real-valued communications on events. As such, model checking is not sufficient: we require a theorem proving approach to prove properties about systems involving our ANN components.

As mentioned, *Circus*, which we use to define the RoboChart semantics, itself has a UTP semantics [Oli06]. We use it here to describe and justify a proof-based approach to verify that a RobChart software model with an ANN controller is correct with respect to a similar model that, instead, uses a traditional controller.

In the next section, we give an overview of the UTP. Section 4.2 presents the rationale for our notion of conformance and its formalisation. Section 4.3 studies compositionality, allowing us to make system-level conclusions when we replace a traditional controller with an ANN controller that is conformant. Section 4.4 addresses the verification of conformance using theorem proving. Related work is covered in Section 4.5, before we conclude in Section 4.6.

4.1 UTP

Our semantic domain for ANNs is the same as that of UTP for Communicating Sequential Processes (CSP) [HJ98a, Chap. 8]. CSP conventionally has a denotational semantics given by failures and divergences, a domain used to describe and analyse the behaviour of concurrent systems. UTP describes this domain predicatively and pointwise. As the name suggests, we capture two critical aspects of a process's behaviour: failure and divergence. We define a failure as a trace-refusal pair, where a trace is a sequence of events a process can perform, and a refusal is a set of events that the process can choose not to engage in after performing the trace.

A divergence is a trace after which the process's behaviour is undefined: anything might happen. However, divergences are not relevant to our semantics of ANNs. Since the process that defines an ANN is divergence free (see Chapter 3).

Hehner-style predicative semantics involves pointwise alphabetised predicates that define various programming theories [Heh84a, Heh84b]. These theories encompass sequential programs and their refinement calculus, as well as process algebras like ACP, CCS, and CSP [HJ98a]. We have established UTP theories for the following programming theories: object-orientation [CSW05, ZSCS14], probabilistic programming [WCF+19, YFW21, YCF+22b, YWF23, YW24], agents [WDWF24], and even hardware [PW08b, BFW09, PW08a, PWSI11].

Remark. To define and reason about predicates, we use the logic described by Woodcock and Davies in *Using Z* [WD96b].[3] The following law is useful.

$$\text{one-point} \qquad x \text{ not free in } e \Rightarrow (\exists x \bullet (x = e) \land P = P[e/x])$$

The one-point rule says that, providing the variable x is not free in the expression e, we can eliminate the quantifier and replace $\exists x \bullet (x = e) \land P$ by $P[e/x]$.

[3] Morgan and Sanders [MS89] have a set of useful laws of the logical calculi that is compatible with the logic used by Woodcock and Davies [WD96b].

Funded by the European Union

We define a programming theory by its alphabet, signature, a complete lattice of predicates, and healthiness conditions that determine theory membership [HJ98a]. The refinement relation orders the complete lattice, making it a specification-oriented structure. A specification is just a predicate in the lattice, and its implementations are all the predicates above it in the lattice. These predicates denote programs. Monotonic and idempotent predicate operators characterise healthiness conditions.

An example of such a theory is the theory of sequential programming [HJ98a]. Its alphabet consists of the observation variables ok and ok', as well as the program's variables. The boolean ok is the observation that the program has started, and the matching ok' is the observation that the program has terminated. The signature of the theory is the syntax of the programming language, in this case, the combinators of Dijkstra's simple nondeterministic programming language: assignment ($x := e$), conditional ($P \lhd b \rhd Q$), nondeterministic choice ($P \sqcap Q$), and recursion ($P = F(P)$). We order the complete lattice of alphabetised predicates by refinement, which is the universal inverse implication ($(P \sqsubseteq Q) \;=\; [Q \Rightarrow P]$), where $[R]$ is the universal closure (quantification) of the predicate R over its alphabet.

Two healthiness conditions characterise the predicates that define the valid observations ok and ok'. First, we cannot observe before the program has started.

$$\mathbf{H1}(P) \;=\; ok \Rightarrow P$$

The fixed points of this function are the predicates that are **H1**-healthy.

Next, we cannot insist that a program fails to terminate. This condition is necessary so that a terminating program can refine an aborting program: anything is better than nontermination. We express this as the monotonicity of a predicate in its ok' variable concerning implication as defined below by **H2**.

$$\mathbf{H2} : P[false/ok'] \Rightarrow P[true/ok']$$

We express this also as a monotonic idempotent function.

$$\mathbf{H2}(P) \;=\; P \,;\, (ok \Rightarrow ok') \wedge \mathit{II}(v)$$

Here, we compose the relation P with one that allows the ok' variable to stay the same or to increase while the program variables stay the same: $\mathit{II}(x) \;=\; (x' = x)$.

We characterise the sequential program theory by the joint fixed points of **H1** and **H2**: only the **H1** and **H2**-healthy predicates are included.

$$\mathbf{H} \;=\; \mathbf{H1} \circ \mathbf{H2}$$

The composition commutes.

Reactive Process Theory. The term *reactive system* was popularised by Harel in his seminal work on Statecharts and systems theory in the 1980s. His influential paper [Har87] introduced Statecharts to model the behaviour of reactive systems: those that maintain an ongoing interaction with their environment over time rather than simply producing a result after a computation. A transformational

Funded by
the European Union

program converts an input stream into an output stream. In contrast, a reactive process pauses to interact with its environment and is often nonterminating.

UTP highlights the distinction between these two types of processes by interpreting a pair of observations *ok* and *wait* [HJ98a]. In this theory, processes can abort, like programs; this is now called *divergence*, but we treat the observations *ok* and *ok′* the same way. A process that is not diverging is called *stable*. The new observations *wait* and *wait′* signify waiting for interaction with the environment. The combination of the values of *ok′* and *wait′* describe the following behaviours:

1. $\neg\, wait' \wedge ok'$ stable termination
2. $wait' \wedge ok'$ stable pause for interaction
3. $wait' \wedge \neg\, ok'$ diverging!
4. $\neg\, wait' \wedge \neg\, ok'$ diverging!

The combination of the values of *ok* and *wait* describe similar behaviour for the process's predecessor concerning sequential composition. There are three healthiness conditions, with the overall condition $\mathbf{R} = \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}$ characterising the valid predicates.

R1 A process cannot undo its history.

$$\mathbf{R1}(P) \;=\; P \wedge tr \leq tr'$$

R2 A process cannot depend on its predecessor's history (analogous to the Markov property).

$$\mathbf{R2}(P) = P[\langle\rangle, tr' - tr / tr, tr']$$

R3 A process depends on its predecessor's termination.[4]

$$\mathbf{R3}(P) = \mathit{II} \lhd wait \rhd P(P \lhd b \rhd Q) = (b \wedge P) \vee (\neg\, b \wedge Q)$$

These conditions are monotonic, idempotent, and commutative.

CSP and *Circus* processes, which are reactive imperative processes, whose alphabet includes programming variables, have the healthiness condition $\mathbf{CSP} = \mathbf{R} \circ \mathbf{H}$ [HJ98a]. We note that this composition is not commutative.

4.2 Overview of conformance

In this section, we first explain a fundamental result in deep learning: the Universal Approximation Theorem and its generalisations (Section 4.2.1). We give some intuition to explain why it applies to *approximate* ANNs (Section 4.2.2) and its restrictions (Section 4.2.3). Finally, we offer informal explanations of refinement and conformance (Section 4.2.4), and formalise conformance (Section 4.2.5).

[4] The relation $P \lhd b \rhd Q$ behaves like P if b is true and Q otherwise.

Funded by
the European Union

4.2.1 Universal Approximation Theorem

The original *Universal Approximation Theorem*, reproduced below, is due to George Cybenko [Cyb89]; it was published in 1989. His formulation applies only to feed-forward ANNs with a single hidden layer and finite neurons.

Theorem 1 (Universal Approximation)**.** *A neural network can approximate any continuous function on a compact subset of* $\mathbb{R}^n$,[5] *given sufficient parameters (weights and biases). For any small* $\epsilon > 0$*, there exists a neural network configuration such that, for all inputs within the domain, the difference between the network's output and the target function is less than* ϵ*.*

A consequence of the Universal Approximation Theorem is that neural networks can model any data or function, regardless of its complexity.[6] This result underpins neural networks' flexibility and power, making them a universal tool for approximation tasks in various domains, such as regression and classification, and even more complex tasks, such as image recognition and natural language processing.[7]

The theorem specifies that, when considering continuous functions over compact subsets of $\mathbb{R}^n$, an ANN can approximate the target function to any desired degree of accuracy. The activation function used in the network is non-constant, bounded, and continuous, examples of which include sigmoid (σ), tanh, and ReLU. However, the theorem does not specify the number of neurons required, how to find the optimal weights and biases, or guarantee the efficiency of the approximation.

The *sigmoid* function, also known as the *logistic* function, takes any real value as input and outputs a value between 0 and 1. It is S-shaped; we usually denote it as $\sigma(x)$. We use the sigmoid function in models that predict probabilities. The sigmoid function is defined as $\sigma(x) = 1/(1 + e^{-x})$. The *hyperbolic tangent* function, tanh, is similar to sigmoid but has an output range of -1 to $+1$. It is also S-shaped and is often preferred over sigmoid because its outputs are zero-centred. The tanh function is used to classify two classes. It is defined as $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$.

The Rectified Linear Unit, *ReLU* [GBB11], is half-rectified and ranges from 0 to ∞. It has a linear, non-saturating property and is used particularly in convolutional ANNs and deep learning. It is defined as $\text{ReLU}(x) = max(x, 0)$. ReLU can accelerate the convergence of gradient descent by maintaining strong, stable gradients (mitigating the vanishing gradient problem), promoting sparse activations (which enhances computational efficiency and generalisation), and avoiding the complexity of other activation functions. This combination of benefits enables deep networks to learn more effectively and efficiently, resulting in faster convergence during training.

[5] A *compact subset* of $\mathbb{R}^n$ within the n-dimensional real space has two important properties: closed and bounded.

[6] The Universal Approximation Theorem is a fundamental result in the theory of ANNs. While we typically prove it using traditional mathematical methods, there have been efforts to formalise and prove aspects of such theorems using mechanical theorem provers. However, we know of no complete mechanical proof of Cybenko's theorem.

[7] See Goodfellow et al. for a foundational understanding of ANNs and their applications in computer vision, natural language processing, reinforcement learning, and generative models [GBC16].

Funded by
the European Union

On the other hand, this may also kill some neurons in the network. This phenomenon is known as the *dying ReLU* problem, which can occur when the weighted sum of neurons becomes negative during training. When this happens, the ReLU permanently sets those neurons to be inactive, which can reduce the model's capacity. In more detail, some neurons can only output zero for any input during training, especially with large learning rates or poor initialisation. Once a neuron outputs zero, the gradient for that neuron also becomes zero (because the derivative of ReLU is zero for negative input values). If a neuron outputs only zeros, its weights stop updating because the gradient is zero.

This behaviour means that these neurons essentially "die" and no longer contribute to learning or to the overall function the ANN is approximating. Once a neuron enters this state, it remains inactive for the remainder of the training process, resulting in a significant portion of the network becoming unresponsive and reducing model capacity and performance. Xu et al. define and empirically evaluate various ReLU-based activations, including Leaky ReLU and Parametric ReLU (PReLU), and discuss how these modifications help alleviate the dying ReLU problem in convolutional neural networks [XWCL15].

The ReLU activation function is called *rectified* because it transforms the input to remove or *rectify* negative values. In mathematics, rectification refers to converting all negative values of a function to zero while leaving positive values unchanged. We call it *linear* because, for positive input values, the function's output is directly proportional to the input. In this region, it is the identity, which is a *linear* function. Hence, the term *Rectified Linear Unit* captures both its linear characteristic for positive inputs and its rectifying nature for negative inputs.

ReLU is described as *non-saturating* because it does not have a ceiling or asymptote that limits the output value as the input increases. This characteristic is important in how activation functions influence the learning process in ANNs.

4.2.2 Intuition for the Approximation

Each neuron in a hidden layer applies a non-linear transformation to its input, enabling the ANN to create increasingly complex functions by stacking these transformations. This piecewise nature allows ANNs to break down complex functions into simpler, locally accurate segments. While locally accurate pieces are essential for building the overall function, the global accuracy of the function depends on how well these pieces are combined. Factors such as network capacity, overfitting, smooth transitions, data coverage, and the effectiveness of the training process all play crucial roles in determining whether the combination of locally accurate pieces yields a globally precise function. Proper network design, sufficient and well-distributed training data, and practical training techniques are crucial.

By modelling these segments individually and then combining them, the ANN can create a flexible and powerful *approximation* of the overall function. Without non-linearity, the ANN would merely calculate a linear combination of its inputs, severely limiting its expressive power. By adjusting the weights and biases, the ANN shapes the output of each neuron to fit different parts of the target function, which we then combine across its entire domain. Adding more layers builds on these approximations, refining the ANN's function representation. Each layer captures dif-

Funded by
the European Union

ferent aspects or features, and deeper layers refine and combine these features to achieve a more and more accurate approximation.

ANNs learn by adapting their weights and biases during training, allowing them to fit various functions. This process minimises the error between the ANN's output and target values. ANNs have finite capacity, and the limited number of parameters restricts their ability to represent complex functions *exactly*. An infinite number of parameters, infinite width (neurons per layer), or infinite depth (number of hidden layers) might be required to represent these functions. Therefore, practical implementations rely on finite resources, resulting in approximations.

ANNs learn from a finite set of training data, and the quality, quantity, and representativeness of this data limit the ANN's ability to generalise to unseen data. Training an ANN involves solving a complex optimisation problem, often using algorithms like gradient descent. These algorithms may only sometimes find the global minimum of the loss function, resulting in an approximation rather than an exact solution.

In addition, real-world data often contains noise and uncertainty [Woo23]. We design ANNs to model the underlying patterns in the data rather than fitting the noise perfectly. As a result, the network produces a smoothed approximation that generalises well to new data, but does not precisely match every data point.

4.2.3 Restrictions on the Universal Approximation Theorem

Cybenko's original Universal Approximation Theorem [Cyb89] established that a feed-forward neural network with a single hidden layer and a non-linear activation function (like the sigmoid function) can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to any desired level of accuracy, given a sufficient number of neurons in the hidden layer. Despite its simple structure, a single hidden layer in a neural network can still provide robust approximations. However, more generalised theorems apply to bounded or arbitrary-width and depth networks. Other researchers have developed these generalisations and extensions of Cybenko's theorem to accommodate different network architectures, activation functions, and types of functions.[8] Here are some of the critical generalisations.

Different Activation Functions Later work showed that the Universal Approximation Theorem holds for activation functions other than sigmoid, such as ReLU and other piecewise linear functions. This extension is significant because many practitioners use ReLU in modern deep learning due to its computational efficiency and effectiveness in training deep networks.

Deep Networks (Multiple Hidden Layers) While Cybenko's theorem applies to single-hidden-layer networks, generalisations of the theorem show that deep neural networks (with multiple hidden layers) can also serve as universal approximators. Deep networks can often achieve the same level of approximation with fewer neurons than shallow networks, due to their ability to capture hierarchical structures in data. Researchers have demonstrated that deep networks can approximate certain functions more efficiently (that is, with fewer

[8] See Augustine for a survey of Universal Approximation Theorems [Aug24].

Funded by
the European Union

neurons or parameters) than shallow networks. This efficiency becomes crucial as the complexity of the target function increases.

Approximation of Non-Continuous Functions While the original theorem focused on continuous functions, extensions have been made to cover the approximation of non-continuous functions. For example, researchers have shown ANNs to approximate measurable functions (integrable, despite discontinuities) under certain conditions. Some extensions focus on functions that are continuous and have bounded derivatives (Lipschitz continuous functions). These results often require more in-depth analysis and specialised techniques.

Function Spaces The original theorem applies to continuous functions on a compact subset, but generalisations extend the results to function spaces like L^p spaces, which include functions that may not be continuous but are integrable to the p-th power. These spaces are essential in many areas of functional analysis and applied mathematics. Further generalisations extend the theorem to Sobolev spaces, which include functions with derivatives up to a particular order that are square-integrable. We use these spaces to study partial differential equations and other advanced topics.

Probabilistic Approximation Some generalisations consider the approximation of functions in a probabilistic sense. For example, instead of requiring the ANN to approximate a function uniformly across the input space, these versions might need the ANN to approximate the function well on average or with high probability over a distribution of inputs.

Convolutional Neural Networks (CNNs) Recent generalisations consider specific ANN architectures, such as CNNs, which are particularly effective for processing structured data like images. Researchers have demonstrated that CNNs with specific architectures can also serve as universal approximators, particularly for functions that exhibit spatial hierarchies or translational invariance.

Residual Networks and Other Architectures (ResNets) There are generalisations considering architectures like residual networks, where the ANN can approximate identity mappings easily. This helps train deep networks and enhance universal approximation capabilities in practical scenarios. Generalisations also extend to Recurrent Neural Networks (RNNs), suitable for sequential data. Under certain conditions, RNNs approximate functions over sequences.

Representation Power of Deep Networks Modern generalisations explore the representation power of deep networks, focusing on how the depth of an ANN influences its approximation capabilities. These studies often explore the trade-off between depth, width, and the complexity of the approximated function.

Summary These generalisations expand the scope of the original Universal Approximation Theorem, accommodating a wider variety of activation functions, network architectures, and function spaces. They provide a deeper understanding of why ANNs are so powerful in practice and offer theoretical assurances for their application across many different problems. For us, this establishes the unavoidable need to accept and deal with approximations when using an ANN.

Next, we explore the impact of approximation on verification.

Funded by
the European Union

4.2.4 Refinement and conformance

Programmers progress through stages of development, including specification, design, and implementation in code, which may involve the use of an ANN. Refinement is the process of establishing a correct relation between the artefacts produced or generated by calculation at these stages. Cavalcanti et al. provide an overview of refinement methods [CSW04]. Morgan provides a refinement calculus for sequential programs [Mor94]; see also Woodcock's tutorials [Woo91b, Woo91a]. A program Q refines a program P ($P \sqsubseteq Q$) if Q is more detailed than P while still satisfying the properties of P. Every execution of Q corresponds to some execution of P.

At different abstraction levels, P represents an abstract specification, while Q is a more concrete implementation. Behavioural consistency requires that Q's behaviour aligns formally with P's. Refinement is typically used in program development to transform specifications into executable code while preserving correctness. As the abstraction levels change, the properties are preserved: every behaviour of the refined program is consistent with what the specification allows.

The related notion of conformance that we define here is a less stringent requirement than refinement. An implementation is considered conformant if it closely approximates the specification. We formalise it in the next section.

4.2.5 Formalisation of conformance

Value Approximation We begin our formal account of conformance by defining the simple concept of one value approximating another within a specified tolerance. The fundamental idea is that value approximation is defined using intervals.

Definition 4.1 (Value Approximation)**.** *We approximate x within ϵ as*

$$approx(\epsilon)(x) \;\hat{=}\; [x-\epsilon, x+\epsilon] \qquad \textit{provided } \epsilon \geq 0$$

The proviso avoids an empty interval, which does not usually present problems.

Sequence Approximation Next, we extend value approximation to sequence approximation. Our motivation for this is to describe how one process trace approximates another, thereby defining approximation for processes.

Definition 4.2 (Sequence Pointwise Approximation)**.** *We lift value approximation pointwise to sequences:*

$$seq\_approx(\epsilon)(xs) \;=\; \{ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\epsilon)(xs(i)))\}$$

Process Approximation We have lifted value approximation to sequence approximation to account for approximate traces. Now, we lift sequence approximation to process approximation, leading to the definition of *conformance* later.

To show that Q conforms within ϵ to P, we characterise all acceptable ϵ-approximate behaviours of P. We then require Q to have one or more of these approximate behaviours. The function Approx(ϵ) defines all the required behaviours.

Funded by
the European Union

Definition 4.3 (Process Approximation)**.**

$$\text{Approx}(\epsilon)(P) \;\widehat{=}\; \textbf{var}\ t := tr \bullet P_{+t}\ ;\ upd\_tr(\epsilon)$$

$$upd\_tr(\epsilon) \;\widehat{=}\; tr :\in \{\, s : seq\_approx(\epsilon)(tr - t) \bullet t \frown s \,\}$$

where t is a fresh variable.

We refer to this definition as a *simulation*.[9] The definition starts with a variable block **var** $t := tr \bullet \cdots$ introducing a new variable t (which must not be free in P). This variable records the value of tr before P is executed ($t := tr$). Next, we execute P with its list of programming variables augmented by t, unaffected by the execution of P: (P_{+t}). Finally, the relation $upd\_tr(\epsilon)$ updates the trace.

The auxiliary definition of $upd\_tr(\epsilon)$ uses the set $\{\, s : seq\_approx(\epsilon)(tr - t) \bullet t \frown s \,\}$. Here, tr is the trace produced by P, and t is the value of tr recorded before P's execution. So $tr - t$ is the contribution to the trace made by the process P. The set comprehension defines all possible traces $t \frown s$ where this portion is replaced by an ϵ-approximation s. Finally, tr is assigned the value of one of these approximations ($tr :\in \cdots$). So, $\text{Approx}(\epsilon)(P)$ describes all the acceptable approximations of P.

We note here that, for simplicity of notation, we are regarding traces as sequences of real values. It is a simple exercise to define a similar function that considers traces of events that convey values, as well as traces with some events that may convey real values and are subject to approximation.

Conformance We now have all the mathematical machinery to define conformance.

Definition 4.4 (Conformance)**.**

$$Q\ conf(\epsilon)\ P \;\widehat{=}\; \text{Approx}(\epsilon)(P) \sqsubseteq Q$$

This says that Q conforms to P, within the tolerance ϵ, if, and only if, Q is an ϵ-approximations of P as defined by *Approx*.

Appendix C provides extensive results about the above definitions. In the next section, we examine the impact of replacing a component P of a process R with another component Q that is conformant to P to obtain a new process S. Ideally, we would like for R to conform to S, but it is not so direct.

4.3 Compositionality

As proof of principle, our key result at present is the monotonicity of internal choice regarding conformance. If two implementations R and S each conform approximately (within ε) to specifications P and Q respectively, then a system that nondeterministically chooses between R and S also conforms (within ε) to a specification that nondeterministically chooses between P and Q .

[9] We adopt the terminology from UTP's *separating simulations* [HJ98a, Chap. 7], where Hoare and He utilise it in the definition of their parallel-by-merge operator.

Funded by
the European Union

Theorem 2 (Conformance theory:- conformance internal choice monotonicity)**.**

$$(R\ conf(\varepsilon)\ P) \wedge (S\ conf(\varepsilon)\ Q) \Rightarrow (R \sqcap S\ \ conf(\varepsilon)\ \ P \sqcap Q)$$

Why does this matter? It ensures that approximate correctness is preserved under nondeterministic composition. In reactive or neural systems, R and S might represent different learned controllers or operational modes. The specification allows the system to behave either as P or Q. This theorem tells us that as long as each controller conforms individually, the overall controller (which switches between them nondeterministically) still conforms—within the same ε bound.

Why is it sound? The logic behind this is that conformance is defined pointwise. If all behaviours of R are ε-approximated by P, and similarly for S and Q, then any behaviour of $R \sqcap S$ is a behaviour of either R or S. So it must be ε-approximated by some behaviour of either P or Q, which are the behaviours of $P \sqcap Q$.

What is the practical relevance of this theorem? Suppose each of the two implementation processes conforms approximately (within the same tolerance ε) to its respective specification. In that case, their nondeterministic combination also conforms—under the same ε bound—to the nondeterministic combination of the specifications.

In practice, it means that we can verify components of a system in isolation and then soundly infer the approximate correctness of a nondeterministic combination of those components. The result is particularly relevant when modelling systems that include mode switches, failovers, or learned behaviours from multiple networks. It guarantees that combining approximate implementations cannot lead to an overall violation of the specification, as long as each branch is itself conformant to the specification.

In the context of RoboChart semantics, nondeterminism in the model is captured using nondeterministic choice. If it involves an ANN controller, then conformance of the ANN controller implies conformance of the nondeterministic choice as a whole. Similar results are needed for the other *Circus* operators.

A weaker result would be to let the two bounds be different. This would perhaps imply that the nondeterministic choice conforms to the maximum of the two bounds. Moreover, our current treatment of conformance is based on the traces model, where the semantics of external choice are the same as those of internal choice; therefore, external choice must also be conformant-monotonic. Our theory provides a principled basis for expressing and reasoning about bounded variation in ANN behaviour.

Our next step is to explore the monotonicity of other CSP operators in terms of conformance. Our experience enables us to predict the likely outcomes. Prefixing seems unproblematic. Sequential composition is expected to be conformant-monotonic in state-free CSP. However, the addition of a state means that a process can depend on its predecessor. This allows the aggregation of tolerances over sequential composition, just like in the pseudo-transitivity property of *Approx*.

Parallel composition is unlikely to be monotonic, because even slight differences between an implementation and its specification can be magnified when synchro-

Funded by
the European Union

nisation is required: one side may refuse to proceed where the other can. As a result, a process that approximately matches its specification on its own may fail to do so when placed in a shared unrestricted context.

Hiding may break conformance because it removes observable actions that are crucial for measuring how closely an implementation matches its specification. By turning visible events into internal ones, hiding may erase distinguishing behaviour or introduce ambiguity, disrupting the trace-based comparison that conformance relies on.

With semantic restrictions (such as determinism and divergence freedom) and structural adjustments (like abstraction-aware conformance), it may be possible to recover monotonicity. These adjustments may make conformance a congruence, and thus better suited for compositional reasoning.

4.4 Proof

In this section, we describe how theorem proving can be used to prove conformance. This paves the way for automation via tactics in the Isabelle/UTP theorem prover for the UTP and Marabou. Specifically, we identify verification conditions sufficient to establish conformance that can be discharged using Isabelle or SMT-based tools, such as Marabou, in conjunction with Isabelle.

We use here UTP reactive contracts: a triple consisting of a precondition (P_1), a pericondition (P_2), and a postcondition (P_3), as shown below.

$$\left[P_1(tt, s) \,\middle|\, P_2(tt, s, ref') \vdash P_3(tt, s, s') \right]$$

The precondition is a predicate over the trace variable (tt) and the initial state variable (s), which characterises the non-divergent behaviours. The pericondition ranges over tt, s and the refusal set (ref'), characterising behaviours where the process awaits interaction. Finally, the postcondition ranges over tt, s, and the final state (s'), characterising terminating behaviours.

For specifications of ANNs, we consider cyclic, memoryless RoboChart controllers, which process all inputs before generating any outputs and then repeat, with no memory between cycles. We denote these controllers with *StandardController*. The semantics of such a controller, as a reactive contract, follows a pattern that captures the relationship between input and output communications using a predicate p that ranges over the trace. Equally, our semantics for an ANN defined by Rule 2 can be captured as a reactive contract of a particular pattern. We describe the patterns and how we obtain contracts instantiating them in [ACW23].

Both the specification and ANN patterns follow tail recursions. Thus, the compositionality of conformance enables us to focus on verifying that a single iteration of the ANN conforms to a single interaction of the cyclic controller. Definition 4.4 captures the pattern for a single iteration of an ANN.

Funded by
the European Union

definition[ANN Reactive Contract Pattern]

$$
\begin{array}{l}
[\ \mathit{true} \\
\vdash\ \#\mathit{in} < \mathit{insize} \land \mathit{tt} = \mathit{in} \land \{\!|\ \mathit{layerRes}.0.(\#\mathit{in}+1)\ |\!\} \not\subseteq \mathit{ref}' \\
\quad \lor \\
\quad \#\mathit{in} = \mathit{insize} \land \\
\quad \exists\, l : 1\,..\,\mathit{layerNo} \bullet \exists\, n : 1\,..\,\mathit{layerSize}(l) \bullet \\
\qquad \mathit{tt} = \mathit{front}(\mathit{layeroutput}(l,n,\mathit{in})) \land \\
\qquad \mathit{last}(\mathit{layeroutput}(l,n,\mathit{in})) \notin \mathit{ref}' \\
|\ \#\mathit{in} = \mathit{insize} \land \mathit{tt} = \mathit{layeroutput}(\mathit{layerNo}, \mathit{layerSize}(\mathit{layerNo}), \mathit{in})\ ]
\end{array}
$$

An instance of this pattern above, for the parallelism between *HiddenLayers* and *OutputLayer* (see Rule 2), is a reactive contract L that has a *true* precondition, indicating that the ANN cannot diverge. The pericondition of L is a predicate specifying that, in an intermediate state, the trace of events observed so far includes only $\mathit{layerRes}.0.n$ events (denoted by *in* here), without repeated values for n, and that $\mathit{layerRes}.0.m$ events that have not been observed are not refused. Alternatively, the trace can include $\mathit{layerRes}.0.n$ events, for all values of n, followed by $\mathit{layerRes}.m.n$ events, for unique values of m and n, but covering all but one $\mathit{layerRes}.\underline{\text{layerNo}}$ event. The postcondition states that the last output is added to the trace, and the action terminates. We use an instance of this pattern to define *ANNController*: the semantics of an ANN component as specified in the previous chapter. We specify the traces that include events other than inputs ($\mathit{layerRes}.0$ events) using a function *layeroutput*. With $\mathit{layeroutput}(l,n,\mathit{in})$, we get the trace up to the point where the n-th node of the layer l has produced its output, with input *in*. Here, *in* is a trace of input events, and l and n are natural numbers.

The definition of $\mathit{layeroutput}(l,n,\mathit{in})$ uses a function $\mathit{annout}(l,n,\mathit{inv})$, which specifies the value communicated by the n-th node of the layer; here, *inv* is the sequence of values of the inputs defined in the trace *in*. We can automatically extract the definition of this function from our process-algebraic semantics.

Next, we present a theorem that establishes verification conditions that can be discharged to prove conformance and avoid proof from first principles.

Theorem 3 (Verification condition for conformance)**.**

$$
\begin{array}{l}
(\forall\, x_1,..,x_{\mathit{insize}} : \mathit{Value} \bullet \forall\, y_1,..,y_{\mathit{outsize}} : \mathit{Value} \mid p \bullet \forall\, i : 1\,..\,\mathit{outsize} \bullet \\
\qquad |\ \mathit{denormO}(i, \mathit{annout}(\mathit{layerNo}, i, \mathit{inpv})) - y_i\ | \leq \epsilon) \\
\Rightarrow \mathit{ANNController}\ \mathit{conf}(\epsilon)\ \mathit{StandardController}
\end{array}
$$

where ANNController and StandardController are instances of the patterns for reactive contracts for ANN controllers and cyclic memoryless controllers; the sequence of input values inpv is given by $\langle \mathit{norml}(1,x_1),..,\mathit{norml}(\mathit{insize},x_{\mathit{insize}})\rangle$*; and the predicate p is the part of the pericondition and postcondition of the instance StandardController that relates its inputs to its outputs.*

Funded by
the European Union

The verification condition identified by Theorem 3 requires that for all sequences of inputs and all sequences of outputs, whose values x_j and y_i are related by a predicate p arising from the *StandardController* specification, the outputs of the ANN must be acceptable. For example, p for the RoboChart Controller *AnglePID* is a function $y_1 = P{*}x_1 + D{*}x_2$. Precisely, acceptability requires that for each i indexing an output y_i, the output of the ANN does not differ from y_i by more than ϵ. The i-th output of the ANN is defined by $denormO(i, annout(layerNo, i, inpv))$, in terms of *annout*. Here, *inpv* is the sequence of input values obtained by the normalisation of each input x_i, that is, $\langle norml(1, x_1), .., norml(insize, x_{insize})\rangle$.

For improved automation of verification of RoboChart models, we provide a further result that can justify the combined use of IsaCircus and Marabou. This is possible when the domain of each input is already bounded, and the predicate p from Theorem 3 defines a monotonic function F on sequences of input and output communications. We can then use the verification condition in Theorem 4.

The input to Marabou needs to define a split of the domain of all possible input values. A split is characterised by a constant *noInt*, representing the number of closed intervals into which the range of every input value x_i is divided. The smaller the value of ϵ, the larger the value of *noInt* required. For each of the *noInt* intervals int_i in a split, we need to provide to Marabou its lower bound $int_i.min$ and its upper bound $int_i.max$. A valid split needs to satisfy the restriction that, for every value v of every input x_j, there is an interval int_i such that $int_i.min \leq v \leq int_i.max$.

Since the ANN is normalised to the range of our *StandardController*, we can prove conformance in Marabou with the verification condition below.

Theorem 4 (Verification condition for Marabou)**.**

$$
\begin{aligned}
\neg\ & \exists x_1, .. x_{insize} : Value \bullet \exists y_1, .. y_{outsize} : Value \bullet \\
& \langle int_1.min, ..., int_{(noInt)}.min\rangle\ F\ \langle y_1, .. y_{outsize}\rangle \wedge \\
& \exists y'_1, .. y'_{outsize} : Value \bullet \\
& \quad \langle int_1.max, ..., int_{(noInt)}.max\rangle\ F\ \langle y'_1, .. y'_{outsize}\rangle \wedge \\
& \quad \exists i : 1 .. outsize \bullet \\
& \quad\quad annout(layerNo, i, \langle x_1, .. x_{insize}\rangle) \leq y'_i - \epsilon \vee \\
& \quad\quad annout(layerNo, i, \langle x_1, .. x_{insize}\rangle) \geq y_i + \epsilon \\
\Rightarrow\ & ANNController\ conf(\epsilon)\ StandardController
\end{aligned}
$$

Theorem 4 states that, given that all intervals are valid, there does not exist an input valuation (x_i) such that either of the following conditions holds. First, *annout* is less than or equal to F evaluated at the maximum point of the interval (y'_i) under ϵ. Second, *annout* is greater than F at the minimum point of the interval (y_i) with ϵ applied. In this case, *annoutput* must be within F under all possible values of x.

One query in Marabou encodes this verification condition for every interval int_i. If Marabou returns UNSAT for every condition, we can use the value of ϵ in Isabelle soundly for conformance, via a certificate that can be automatically obtained

Funded by
the European Union

through Marabou [IBZK22]. Such certificates can be reconstructed using CVC4 and CVC5 in Isabelle [LFA+24]; this would constitute progress towards integrating Marabou with Isabelle via Sledgehammer [BKPU16].

By applying Theorem 4 to our example with *AnglePID* and *AnglePIDANN*, we obtain a verification condition of the form described below.

$$
\begin{aligned}
&\neg\ \exists x_1, x_2 : \mathit{Value} \bullet \\
&\quad \forall i : 1 \mathinner{..} 2 \bullet \\
&\qquad \mathit{int}_i.\mathit{min} \leq x_i \leq \mathit{int}_i.\mathit{max} \land \\
&\qquad y_1 \geq (P * (\mathit{int}_i.\mathit{min}) + D * (\mathit{int}_i.\mathit{min}) + \epsilon) \lor \\
&\qquad y_1 \leq (P * (\mathit{int}_i.\mathit{max}) + D * (\mathit{int}_i.\mathit{max}) - \epsilon)
\end{aligned}
$$

When we instantiate the constants in the conditions above, we obtain an ϵ error value of 0.085. We use the range $\{0..1\}$ as *Value*, given our normalisation assumption. We have also instantiated F as a function capturing the behaviour of *AnglePID* in terms of the constants P and D. Finally, *noInt* is set to 100. In this way, as a proof of principle, we have used Marabou and Isabelle to verify the ANN component *AnglePIDANN* against the standard controller *AnglePID*.

4.5 Related work

The verification of ANNs is a popular and increasingly researched topic in the literature [ZWL+24], but the challenges involved vary depending on the type of verification considered. Here, we address system-level verification.

When considering the verification of an ANN model in isolation, its functional behaviour can be captured using a composition of functions with n real input variables and m real output variables. This formalism enables multiple mathematical analysis techniques, such as transformations upon geometric objects, linear algebraic reasoning, and abstract interpretation analysis. However, these functions capturing an ANN are non-convex and non-linear due to the nature of the activation functions, so verifying properties forms an NP-Complete computational problem [K+19]. Therefore, using SMT solvers without dedicated support for such networks is intractable for even small ANNs. There are variations in the style of properties these solvers accept, but in general they establish a reachability condition, which is a predicate over the input and output vector spaces of an ANN function.

When we consider what verification looks like for software systems involving ANN components, however, we face a different problem. First, engineering with imprecise components from a formal analysis perspective is challenging. The impact that an imprecise component has on the behaviour of the software, and on refinement and compositional reasoning is a research challenge. Furthermore, identifying the contexts in which we can accept such imprecision and proving that this imprecision does not cause unintended consequences for an arbitrary system is another key challenge. Finally, when considering a software component that implements the model of an ANN, its behaviour cannot be accurately captured by just a functional

Funded by
the European Union

model. To model and reason about the behaviour of this software component, we need to prove the reactive properties of this component.

At the component level, another challenge is the need for ANN-specific verification techniques. We can capture software implementing an ANN as a process to prove its structural and reactive properties. This model, however, is not adequate to establish properties about the ANNs behaviour, as discussed earlier. A challenge is relating properties of systems to properties of ANN functions, to effectively use these solvers to prove high-level properties about this component in context.

In the literature currently, verification of AI-enabled systems using hybrid automata is the dominant approach [LCTJ23, IJH+21, ZWL+24, SKS19]. It is suitable for modelling equations that capture real phenomena, as it focuses on state transitions captured using differential equations and can model physical phenomena and the transitions between states through events. In a hybrid automata, events are transitions between states, labelled optionally to enable parallelism with other automata. This means that the properties of the interaction and communication between these events are very challenging, or impossible, to establish tractably with this approach.

Some authors have attempted to add this level of reasoning to the hybrid automata model [LSV03]. These are much more complex and require a significant amount of work to reorient the model to establish these properties, and especially to implement and create tools for the automated use of these models.

The neural network verification software tool (NNV) [10] appears to be the system-level ANN tool with the most support and features. It supports neural ordinary differential equations, semantic segmentation networks, and recurrent neural networks [LCTJ23]. Also, it supports systems involving ANNs specified as hybrid automata, including automated reasoning facilities to support this analysis. Its tooling supports multiple file formats, many more types of neural networks that we support, and is based in Matlab. Its analysis for system-level properties is based on reachability analysis. The primary difference to our work is the representation of ANNs as functions, and systems as captured using hybrid automata, which changes the types of system-level properties that can be established.

The work in [ZWL+24] proposes an approach to unifying qualitative and quantitative methods to ANN system verification [SSS22]. The approach synthesises a barrier function on a hybrid system with multiple ANN components. The intuition is that, given a set of safe states, a barrier condition is used to prove that the system can reach no unsafe state, or obtain an upper and lower bound on the probability of doing so. This approach enables probabilistic analysis, analysis of multiple large ANN component models, and timed reasoning. It shares the same drawbacks discussed about hybrid system analysis and functional abstractions.

BehaVerify is a tool for verifying behavioural trees [SJ22]. Its approach uses a behavioural tree defined in Python, and records information about the tree depending on the type of encoding. This work is related to ours because its profile of a behavioural tree is similar to a network of interacting processes. This formalism is more expressive than ours, as it captures node memory and blackboard variables

[10] github.com/verivital/nnv

Funded by
the European Union

(shared state), making it significantly more challenging to relate these models to ANN tools. Also, the work considers specifications defined in LTL.

In [BS23], the authors use Isabelle/HOL to formalise concepts common to ANN models. This is the only work we are aware of that utilises Isabelle for ANN verification, similar to our approach. The models from [BS23] are platform independent, and can be used to verify properties over the input and output of an ANN using Isabelle's theories for real arithmetic. The authors define two encodings of an ANN: the 'textbook' style, as graphs, and the layer style, capturing Tensorflow layers. This work has close links to practical file formats: it can support the automated interpretation of a network trained in Tensorflow, and can potentially be used to establish correctness and sound transformations of one file format to another, proving semantic correspondence between the networks. Their work also supports a broader range of activation functions and layer types. Unlike our work, however, they do not consider a formal model capturing a system, and are concerned with ANNs in isolation.

Showing that an ANN approximates a logical property is an exciting area that bridges formal logic, approximation theory, and machine learning. This work is crucial for understanding capabilities and limitations of ANNs in representing logical structures. We provide an overview of some key developments in this area below.

Habeeb and Prabhakar discuss approximate conformance of ANNs [PP24]. They explore a method for verifying that two ANNs behave approximately the same when given the same inputs. The key concept is ϵ-*conformance*, which allows for slight differences in output, controlled by the parameter ϵ. Given two ANNs N_1 and N_2 with the same input/output structure, the goal is to check if their outputs differ by no more than ϵ for any input within a given range. Like we do, they note that exact conformance ($\epsilon = 0$) is too strict, and so use approximate conformance.

Existing verification methods, such as ReluDiff and StarDiff, either work only with very similar ANNs or are computationally expensive. So in [PP24] the authors propose a more efficient method by transforming the conformance-checking problem into a reachability analysis problem. The technique transforms the two ANNs into a single network N that simulates the joint behaviour of both ANNs. Analysing the reachability of this combined ANN determines conformance. This technique checks the reachable set of outputs for given inputs and determines whether the output difference between N_1 and N_2 remains within ϵ.

The authors use Python to evaluate conformance and apply various ANN verification tools (for example, nnenum and α-β-CROWN). They experimented on 27 network pairs from the ACAS Xu benchmark, including ANNs for aircraft-collision avoidance. The authors present evidence that the proposed method is faster and more reliable than existing approaches. They can verify conformance or detect violations for more ANN pairs in less time using reachability tools. They test the method in a real-world scenario: an automatic rocket landing system. They analyse the effects of replacing a controller ANN with an ϵ-conformant one in a closed-loop system and observe that the deviation between system states increases over time due to the accumulation of input-output deviations. Their method for approximate conformance verification shows promise, outperforming traditional approaches.

Funded by
the European Union

The authors suggest that future work could focus on extending conformance verification methods to more complex closed-loop systems.

Next, we discuss more general approaches.

Neural-Symbolic Integration This field combines ANNs with logical reasoning systems. Besold et al. survey work in neural-symbolic learning, reasoning, and approaches to integrating ANNs with symbolic AI systems [BdGB+17].

Logic Tensor Networks LTNs provide a framework for integrating logical reasoning with deep learning. This framework shows how logical formulas and data are combined into tensor representations, processed through an ANN, and evaluated using fuzzy logic. The system then optimises its weights based on this evaluation. We refer to Serafini and Garcez's work on LTNs, deep learning, and logical reasoning from data and knowledge [SdG16].

Approximating Logical Formulas Research has been done on how well ANNs can approximate specific types of logical formulas. We refer to Lee et al. for a discussion of the expressiveness of ANNs [LGM+17].

Rule Extraction from ANNs This involves deriving logical rules that approximate the behaviour of ANNs. The review by Hailesilassie on rule extraction algorithms for ANNs [Hai16] provides an overview.

Neurosymbolic AI This approach combines ANNs with symbolic reasoning to approximate logical properties. Garcia and Lamb discuss neurosymbolic AI and how combining neural and symbolic approaches can lead to better approximations of logical properties [GL23].

Probabilistic Logic Networks These networks represent and reason with uncertain logical knowledge. Qu and Tang propose a framework for reasoning with uncertain logical knowledge using ANNs [QT19].

Fuzzy Neural Networks These networks incorporate fuzzy logic into ANN architectures to better approximate logical properties with degrees of truth. Ras et al. provide a comprehensive overview of the work in this area [DSM20].

These approaches often face challenges in scalability, interpretability, and handling complex logical structures. Current research focuses on improving the fidelity of logical approximations, handling more complex logical systems, and developing more efficient training and verification methods.

4.6 Final considerations

In this chapter, we have introduced and explored the concept of conformance for ANNs, drawing inspiration from established formal methods and theories such as the UTP. We have presented conformance as a refinement-like relationship where an ANN N conforms to a specification S if N is a refinement of the ϵ-approximation of S. With this approach, we can verify that the refined or implemented ANN adheres to the specified properties within a given margin of error.

The application of conformance to ANNs addresses the challenges posed by their inherent approximation, ensuring that the ANN's behaviour aligns closely with its

Funded by
the European Union

intended specification. Through the use of ϵ-approximations, we can verify ANNs even when exact conformity is computationally infeasible, thus making it possible to establish rigorous correctness properties in practical implementations.

Funded by
the European Union

5 Uncertainty identification and quantification

5.1 Introduction

This chapter presents our work on uncertainty identification and quantification in robotics systems and their environments. Uncertainty is a fundamental challenge in robotics, arising from sensor noise, environmental variability, model inaccuracies, and dynamic interactions with environment elements. Addressing these uncertainties is essential for improving the reliability, adaptability, and decision-making capabilities of robotic systems operating in real-world conditions.

In this chapter, uncertainty identification refers to the process of recognising and categorising uncertainties arising from various sources. Identifying uncertainty in self-adaptive robotics is challenging due to the inherent complexity of these robots, including their complex interactions, evolving configurations, and adaptive behaviours, as well as the lack of comprehensive knowledge about unpredictable operational contexts and environmental dynamics. As a result, practitioners often rely on their intuition and insights from previous experiences with similar systems to identify and manage uncertainties in self-adaptive robotics. Considering this challenge, our objective is to develop systematic and automated methods for identifying uncertainty in self-adaptive robots. Furthermore, we aim to develop a comprehensive taxonomy of uncertainties by analysing existing literature and gathering perspectives from industry practitioners, which will ultimately lay the groundwork for advancing uncertainty quantification techniques.

Following the identification process, uncertainty quantification aims to measure and represent the degree of uncertainty in a mathematically rigorous manner, and we are specifically interested in quantifying uncertainty in DL models employed in robotic systems. DL has become a key component in modern robotics, enabling perception, decision-making, and control in complex and dynamic environments. However, these models often operate in uncertain conditions where sensor noise, partial observability, and data distribution shifts can significantly impact their performance. Accurately quantifying uncertainty in deep learning models is therefore crucial for improving the reliability and robustness of robotic decision-making. Various methods are employed for this purpose, including Bayesian deep learning [TDVDWH19], which incorporates probabilistic reasoning into neural networks, Monte Carlo Dropout (MC-Dropout) [GG16], which approximates Bayesian inference through stochastic sampling, and deep ensembling [LPB17], which leverages multiple model predictions to estimate uncertainty. These techniques allow for more informed decision-making by providing uncertainty-aware estimates that enhance the robustness and safety of robotic systems.

In the following sections, we first present our work on uncertainty identification in Section 5.2. We then introduce our work on DL uncertainty quantification based on MC-Dropout in Section 5.3.

5.2 Uncertainty identification

Self-adaptive robots are expected to operate in highly dynamic environments while effectively managing uncertainties. A fundamental challenge in such robots is man-

Funded by
the European Union

aging uncertainty throughout their entire engineering lifecycle, from initial design to active operations [WCM+23b]. Uncertainty can arise from various sources, such as unpredictable environmental conditions, sensor and actuator noise, and human-robot and robot-robot interactions. Such factors directly influence the dependability of a robot, ultimately impacting its overall performance and decision-making capabilities. Therefore, an important concern is to identify the sources and potential impacts of uncertainties at early stages of the robotics software engineering lifecycle [WCM+23b]. Identifying uncertainty in self-adaptive robots is challenging due to their inherent complexity, including their complex interactions, evolving configurations, and adaptive behaviours, as well as a lack of comprehensive knowledge about unpredictable operational contexts and environmental dynamics. Consequently, practitioners often depend on their intuition and insights from previous experiences with similar systems to identify and manage uncertainties.

To tackle the current challenges of uncertainty identification, our goal is to develop a systematic and automated approach to assist industry practitioners in identifying uncertainties based on robotics system requirements. Considering the increasing popularity and widespread adoption of large language models (LLMs) across various domains [YLL+23, HOL+24], we investigate their potential for comprehending robotics system requirements and identifying associated uncertainties. For this purpose, we leveraged requirements from four industrial robotics use cases: the Industrial Disassembly Robot from the Danish Technological Institute (DTI), the Warehouse Robotic Swarm from PAL Robotics, the Prolonged Hull of an Autonomous Vessel from the Norwegian University of Science and Technology (NTNU), and Human-Robotic Interaction from Fraunhofer IFF. We selected 10 advanced LLMs with varying capabilities, namely Gemini Pro 1.5, Perplexity Sonar, Nemotron 70B, Nova Pro, Mistral Large 2411, LLama 3.3 70B, o1 Preview, GPT-4o, Gemini Flash 2.0, and Claude 3.5 Sonnet. For evaluation, we provided LLMs with the software requirements of robots and carefully designed prompts containing clear instructions and an uncertainty questionnaire (inspired by existing work [RJC12]). After obtaining and compiling responses from the LLMs, we conducted in-person sessions with the practitioners to discuss and gather their feedback. Using input provided by practitioners, we analysed the quality, relevance, and usefulness of the uncertainties identified by LLMs.

Our evaluation results demonstrated that practitioners agreed with 63–88% of the responses generated by all LLMs. Discussions with practitioners indicated that LLMs can effectively comprehend and analyse robotics system requirements, using both the given information and their prior domain knowledge to identify novel uncertainties. It was observed that uncertainties are most prevalent during testing and operational phases, with environmental dynamics identified as a major source. Moreover, our analysis revealed that techniques like modelling, simulation, digital twins, and uncertainty quantification are commonly used to manage uncertainty. However, applying these techniques in practice requires substantial manual effort, highlighting the need for automated frameworks and tools. Finally, our evaluation suggested that LLama 3.3 70B, o1 Preview, GPT-4o, Mistral Large 2411, and Nova Pro are the most effective LLMs for uncertainty analysis.

The complete paper, including technical details and all results, is available in Appendix D. The paper has been submitted to IEEE Software and is currently under

Funded by
the European Union

review. The preprint reference is provided below.

- Hassan Sartaj, Jalil Boudjadar, Mirgita Frasheri, Shaukat Ali, and Peter Gorm Larsen. "Identifying Uncertainty in Self-Adaptive Robotics with Large Language Models", *arXiv:2504.20684* (2025).

5.3 Deep learning uncertainty quantification

Uncertainty may arise during the design and training of deep learning (DL) models; therefore, uncertainty quantification (UQ) is crucial for understanding the model's reliability and performance. Section 5.3.1 introduces the UQ method based on Monte Carlo dropout and two types of UQ metrics. Section 5.3.2 presents the specific applications of the developed UQ method for assessing uncertainty and robustness in one of our case studies, i.e., DL-based sticker detectors in the laptop refurbishment process.

5.3.1 Monte-Carlo dropout-based uncertainty quantification

DL has revolutionised various domains, such as computer vision [VDDP18] and autonomous systems [GTCM20], by providing powerful tools for predictive modelling and decision-making. On the other hand, DL systems are prone to unexpected and incorrect behaviours, especially when faced with edge cases or complex scenarios, such as adversarial attacks, noisy data, or unforeseen input distributions [CW17]. This vulnerability highlights the importance of quantifying uncertainty and evaluating model robustness, which is crucial for trustworthy predictions and decision-making, as incorrect predictions can cause critical applications to fail with serious consequences. DL is affected by two main types of uncertainty: aleatoric and epistemic uncertainty [HW21]. Aleatoric uncertainty arises from inherent randomness or variability in the system, such as sensor noises, which cannot be reduced even by collecting more data. Epistemic uncertainty is caused by a lack of knowledge about the best DL model, which can be reduced by involving more data.

In this deliverable, we aim to quantify epistemic uncertainty. Bayesian Neural Network (BNN) provides a probabilistic approach to neural network modelling that allows quantifying epistemic uncertainty through Bayesian inference [TDVDWH19]. BNN produces a distribution of possible outputs that captures the uncertainty about the prediction, but it requires computationally expensive posterior calculations and is complex to implement. A more practical alternative is Monte Carlo Dropout (MC-Dropout) [GG16], which offers a Bayesian approximation for quantifying epistemic uncertainty. In DL, the dropout layer is commonly applied in deep neural networks (DNNs) to avoid over-fitting by randomly dropping units along with their connections from the network [SHK+14], and MC-Dropout shows that introducing dropout in DNNs can be interpreted as the approximation of a probabilistic Bayesian model in deep Gaussian processes. MC-Dropout approximates the predictive distribution by generating multiple predictions based on which the uncertainty is quantified.

Therefore, we developed a UQ method based on MC-Dropout to capture the uncertainty in DNNs used in robotic systems. To address measures for uncertainty, we create two types of UQ metrics to quantify uncertainty in two primary fundamental

Funded by
the European Union

DL problems: UQ metrics for classification and regression. For classification tasks, we employ three standard metrics: variation ratios, predictive entropy, and mutual information. For regression tasks, we use a standard metric called total variance and a specialised metric called prediction surface [CYA21]. The prediction surface was initially proposed to quantify uncertainty in object detection models for autonomous driving. The object detection problem can be considered as a combination of classification and regression tasks. The model first identifies the bounding box to locate an object (regression) and then assigns it a class label (classification). Since object detection models typically detect multiple objects simultaneously in a single prediction, and MC-Dropout requires multiple predictions, clustering is necessary to assign each prediction to an object. To this end, we adopt the HDBSCAN algorithm [CMS13] to cluster objects based on their predicted bounding boxes. We then compute UQ metrics for each detected object, including metrics for bounding box regression and label classification. In addition to the UQ metrics, we develop novel uncertainty-based robustness metrics to measure the robustness of DL models under uncertainty. Finally, we create a benchmark dataset construction method based on the Vision Language Models (VLMs) and adversarial generation techniques.

5.3.2 Assessing uncertainty and robustness in laptop refurbishment

The European Union's Circular Economy Action Plan (CEAP) highlights the need for sustainable operations to promote circular economy processes and encourage sustainable consumption [CfC20]. One essential activity is refurbishing electronic devices (e.g., laptops) to extend their lives, reduce electronic waste, and provide affordable options for consumers. A critical and time-consuming step in refurbishment is removing stickers from the laptop, which involves first identifying the stickers and their locations on the device. Manual cleaning is time-consuming and requires finding sufficient workers; current automation solutions are not designed for this level of variation, which limits sustainability and scalability. Thus, novel solutions to transition from manual labour to automated processes are paramount. Robotics offers a promising solution to simplify and scale up this process, increasing efficiency and reducing labour costs.

The Danish Technological Institute (DTI) develops, applies, and transfers technology to industry and society. One key area that DTI focuses on is the automatic refurbishing of laptops. Targeting the cleaning process in laptop refurbishment, DTI built several DNN-based sticker detection models (*SDM*s) for automatic sticker detection, which is the basis for successful automatic sticker removal. The *SDM*s are built on open-source object detection DNNs and trained using a sticker detection dataset specially designed by DTI. However, like other DNN models, *SDM*s are vulnerable to challenging scenarios such as adversarial attacks or unforeseen input distributions. This vulnerability emphasises the need to quantify uncertainty and assess the robustness of the *SDM*s, which is crucial for trustworthy sticker removal, as inaccurate sticker detection could damage the laptop surface or lead to incomplete sticker removal, affecting the overall refurbishment quality.

To this end, we conduct a comprehensive empirical evaluation to assess the *SDM*s that DTI uses in the laptop refurbishing process, focusing on detection accuracy, prediction uncertainty, and adversarial robustness. Specifically, we adopt MC-

Funded by
the European Union

Dropout as the UQ method to capture the uncertainty in model predictions. We run the model to perform multiple predictions, and based on these predictions, we calculate two types of UQ metrics: label classification UQ metrics and bounding box regression UQ metrics. In addition to UQ metrics, we define *robustness score (RS)* to measure the robustness of the *SDM*s. *RS* measures robustness from two perspectives: robustness concerning predictive precision and robustness concerning prediction uncertainty. Regarding benchmark datasets, we construct benchmark datasets from three data sources: datasets provided by our partner DTI, datasets synthesised by prompting two VLMs, i.e., *DALL-E-3* [BGJ+23] and *Stable Diffusion-3* [RBL+21], and datasets created using an adversarial attack technique named Dense Adversary Generation [XWZ+17]. Our evaluation results show that different *SDM*s achieve different performance regarding different evaluation metrics. Specifically, regarding sticker detection accuracy, *Faster R-CNN_v2* is recommended as the best *SDM*, while *RetinaNet_v2* achieved the overall best performance regarding prediction uncertainty. Regarding adversarial robustness, *Faster R-CNN_v2* and *RetinaNet_v2* are recommended as the best SDMs. We also quantitatively studied the correlation of uncertainty quantification metrics and accuracy (detailed in Appendix E). Finally, we also provide guidelines for *SDM* selection and lessons learned.

The complete paper, including technical details and results, is available in the Appendix E. That paper has been submitted for review.

- Chengjie Lu, Jiahui Wu, Shaukat Ali, and Mikkel Labori Olsen. "Assessing the Uncertainty and Robustness of the Laptop Refurbishing Software", 18th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2025.

5.3.3 Assessing uncertainty in ML-based anomaly detector

Anomaly detection identifies data patterns that deviate significantly from expected behaviour and plays a vital role in diverse research areas, including cybersecurity, robotics, and healthcare. In the context of robotic systems, anomaly detection is particularly important due to the increasing autonomy and complexity of robots operating in dynamic, real-world environments. Anomalies in robotic systems, such as hardware malfunctions and sensor degradation, can compromise performance, safety, and decision-making. Traditional monitoring methods often fall short in capturing such diverse and subtle irregularities. Therefore, incorporating advanced anomaly detection techniques, especially those based on ML and real-time data analytics, is essential to detect and respond to failures proactively, enhance system resilience, and build trust in robotic systems.

However, a critical limitation of many ML-based anomaly detectors is their tendency to make overconfident predictions, even in situations where the model has limited or no prior experience. This is particularly critical in robotics, where high-stakes decisions must be made in real time, often under uncertain and dynamic conditions. Quantifying uncertainty in anomaly detection enables the system to express its confidence in each prediction, allowing it to distinguish between apparent anomalies and ambiguous or novel situations. By incorporating uncertainty estimates—such as those derived from Bayesian neural networks, ensemble mod-

els, or probabilistic approaches—robotic systems can make more cautious and informed decisions, inform human intervention when appropriate, and prioritise data collection or model updates in uncertain regions.

To this end, we employ the MC-Dropout UQ method to an ML-based anomaly detector developed by AUTH. Specifically, the anomaly detector detects anomalies in lidar readings of the Turtlebot4, which takes lidar readings from the Turtlebot4 as its input and outputs 0 for normal readings and 1 if an anomaly is detected. We employ the MC-Dropout UQ method to quantify the uncertainty of the anomaly detection outputs. This work represents an initial concept that has been preliminarily implemented in the Turtlebot4. Further exploration and development are still ongoing.

Funded by
the European Union

6 Tool support

In this chapter, we present the tools we have developed for RoboArch and RoboChart to support the notations in techniques presented in the previous chapters. Section 6.1 presents our support for use of RoboArch, embedded in the RoboStar tool: RoboTool. Section 6.2 presents our encoding of *Circus* in Isabelle, which is a front-end to use of Isabelle/UTP. Automatic generation of *Circus* semantics for RoboChart models is the topic of Section 6.3. The application and extension of that work to provide tool support to reason about RoboChart models that have ANN controllers is the topic of Section 6.4. In Section 6.5, we describe how we can use the semantics of RoboChart models to prove deadlock freedom. This work addresses the limitation of the use of model checkers in terms of the size of the state of the models that can be handled, while maintaining a high-level of automation. These results also provide the stepping stones for future exploration of the semantics for proof of other properties and generation of traces. The latter will be useful for legitimisation and trustworthiness checking.

6.1 RoboArch in RoboTool

RoboTool[11] provides a set of tools, implemented as Eclipse plugins, for the RoboStar notations, including RoboArch and RoboChart. As part of our efforts to formalise in RoboArch the MAPLE-K architecture, as presented in Chapter 2.1, we have extended the support for RoboArch in RoboTool to handle MAPLE-K layers. In this section, we present the modelling plug-in in Section 6.1.1, and the RoboChart generation plug-in in Section 6.1.2. The tools are available for download and use[12], where instructions for installation and use can be found.

6.1.1 RoboArch Xtext Implementation

The RoboArch language itself is built upon an Eclipse Modelling Framework (EMF) metamodel, which we have partially presented in Section 2.1. As described in Section 2.3, we have extended the metamodel to include a new *MAPLEK* pattern, and this enables the inclusion of MAPLE-K into the underlying RoboArch models.

The textual language of RoboArch has been implemented on top of the metamodel using Xtext[13]. This has produced an editor with support for syntax highlighting and checking of the well-formedness conditions as specifications are written. We have extended the Xtext grammar to support writing models with the *MAPLEK* pattern in the textual format shown earlier in Fig. 3.

Fig. 14 shows a screenshot of the editor, where we can see part of the description in RoboArch of the architecture for the DTI case study. The editor is implemented as an Eclipse plugin, and can be seen with a RoboArch file named `screw-detection.rac` opened. The specification shows the definition of a `datatype` and an `enumeration` as well as two `layer`s: `ScrewDetection` and `Adaptation`. The editor features syntax highlighting, which is visible through keywords displayed in bolder magenta text and

[11] https://robostar.cs.york.ac.uk/robotool/
[12] https://drive.google.com/file/d/1vy_SBFBp2vxyPIDk-wq_VmYN0rol2a_r/view?usp=sharing
[13] https://eclipse.dev/Xtext/

Funded by
the European Union

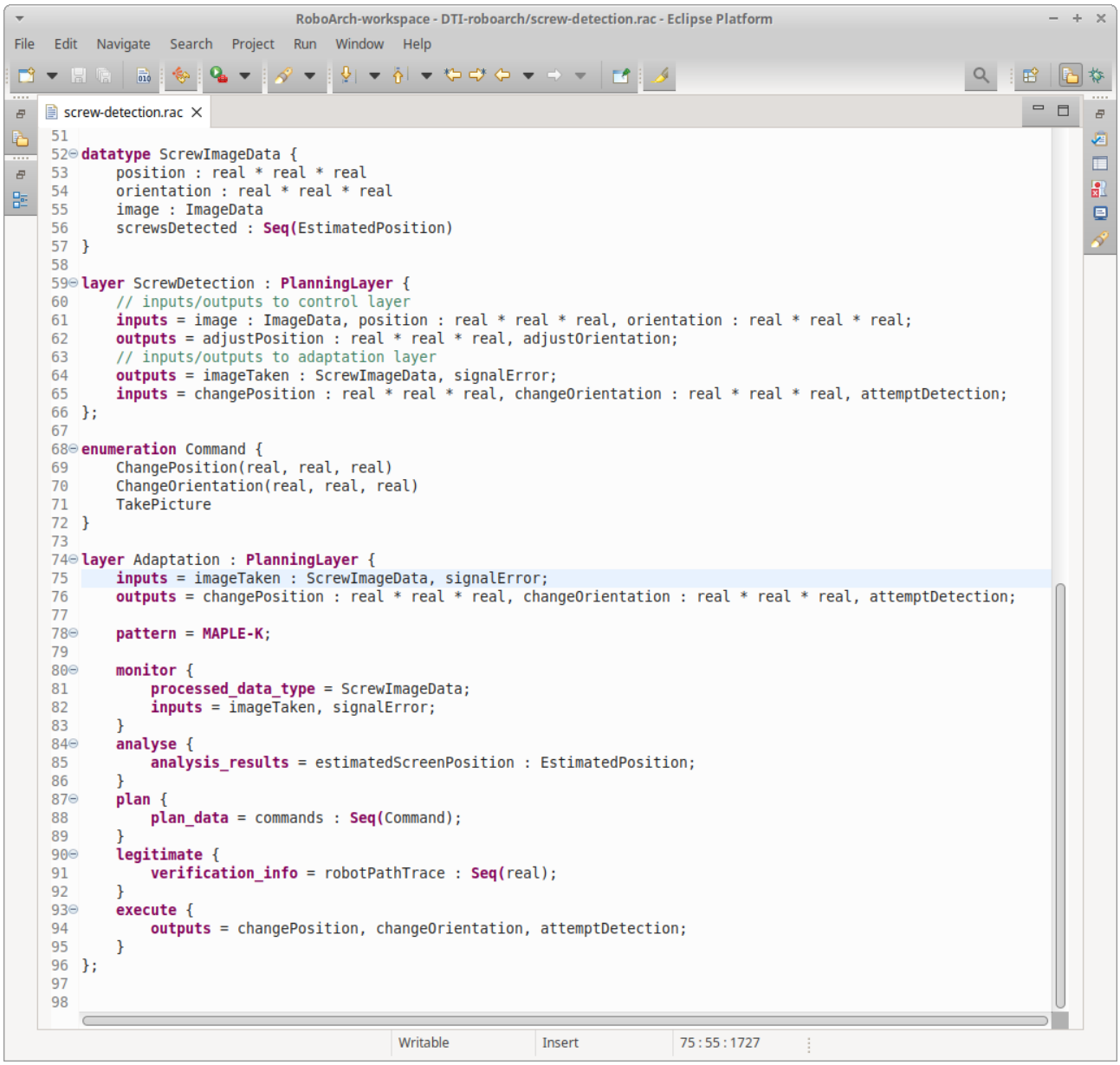


Figure 14: The RoboArch editor, showing part of the RoboArch specification for the DTI case study

Funded by
the European Union

```
// MK2
@Check
public void maplekWithLegitimateRequiresPlan(MAPLEK pattern){
    if ( pattern.getLegitimate() != null ) {
        if ( pattern.getPlan() == null ) {
            if (pattern.eContainer() instanceof Layer) {
                error("Layer '" + ((Layer) pattern.eContainer()).getName()
                        + "' has a legitimate component but no plan component.",
                        RoboArchPackage.Literals.MAPLEK__LEGITIMATE);
            } else {
                error("A legitimate component must be accompanied by a plan component.",
                        RoboArchPackage.Literals.MAPLEK__LEGITIMATE);
            }
        }
    }
}
```

Figure 15: The implementation of the **MK2** well-formedness condition

comments shown in green. The `Adaptation` layer has the `MAPLE-K` pattern, showing how the grammar has been extended to handle the extensions to RoboArch.

Fig. 15 shows, as an example, our implementation of the well-formedness condition **MK2** in the Java programming language. It consists of a function, annotated with `@Check` to integrate it into Xtext's validation infrastructure. The function takes a `pattern`, which is of the `MAPLEK` type representing an instance of the *MAPLEK* metamodel element, over which the well-formedness condition is checked. The function first calls `getLegitimate()` on the `pattern`, checking if its result is not `null` to determine that the pattern has a *legitimate* component. If a *legitimate* component is present, it performs a similar check that there is not a *plan* component.

If both conditions hold (there is a *legitimate* component but no *plan* component), **MK2** is violated, so an error is reported using Xtext's `error` function. This is split into two cases so that the name of the containing layer can be included in the case where the pattern belongs to a layer. (We recall the possibility of patterns occurring elsewhere). In both cases, a literal `MAPLEK__LEGITIMATE` is used to signal where the error should be reported: on the pattern's `legitimate` block.

6.1.2 Translation to RoboChart with Epsilon

The translation of a RoboArch model to RoboChart has been implemented using Epsilon[14] to perform a model-to-model transformation, using the rules from Section 2.4 to map from the elements of RoboArch's metamodel to corresponding RoboChart elements. We have extended the existing rules with the rules for our translation discussed in Section 2.4.2, thus allowing the automatic generation of RoboChart models from RoboArch models using the *MAPLEK* pattern.

Fig. 16 shows a screenshot of the RoboChart generator available for use in Eclipse. It shows the *Convert RoboArch to RoboChart* right-click context menu option for running the tool on the RoboArch file `screw-detection.rac`, along with directories

[14] https://eclipse.dev/epsilon/

Funded by
the European Union

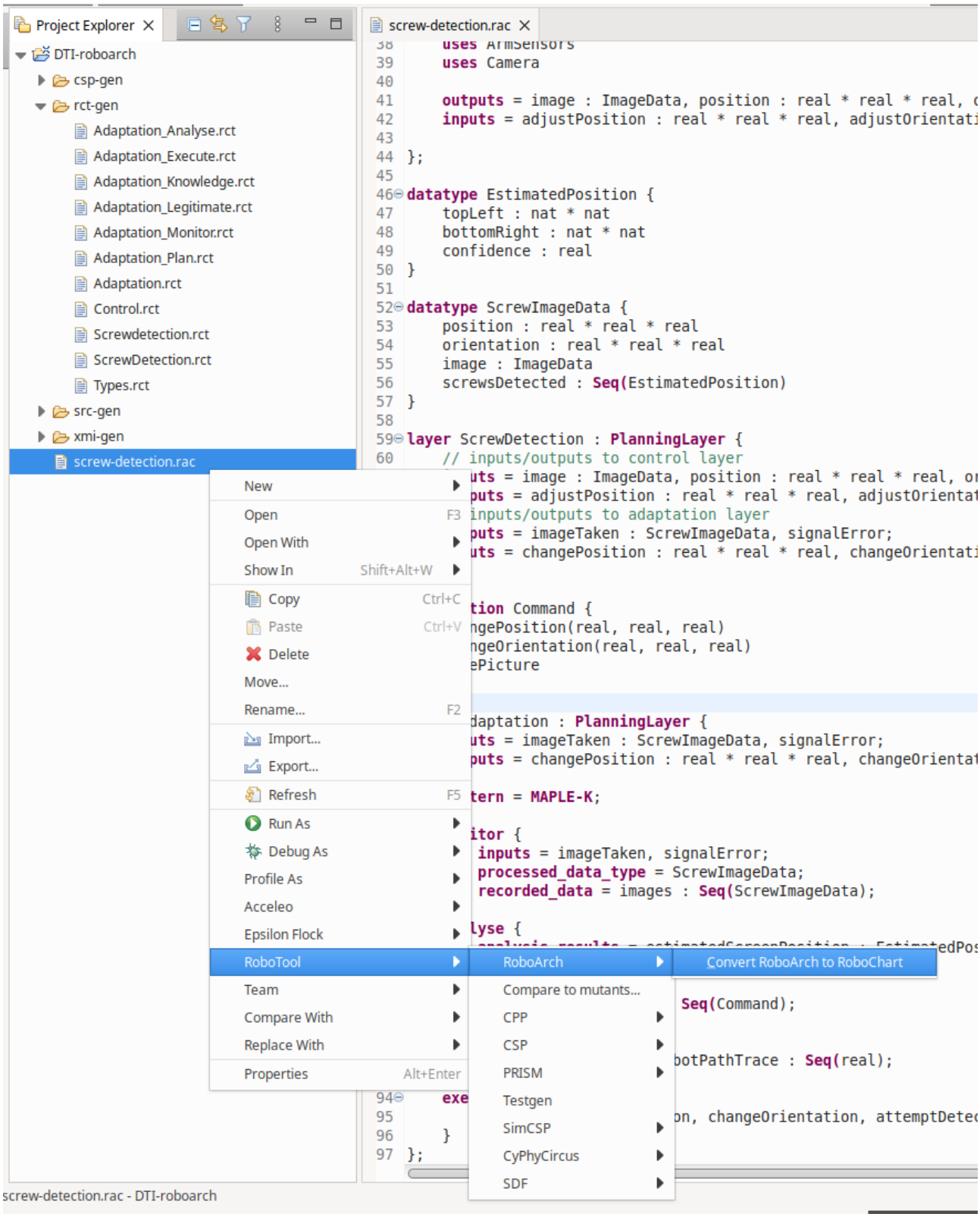


Figure 16: The tool for converting RoboArch to RoboChart

generated from a previous run of the tool. Our new tool generates intermediate artefacts in the *xmi-gen* directory, and the RoboChart model in the *rct-gen* directory, which is open to show the RoboChart files produced. The *csp-gen* and *src-gen* files are also automatically produced by existing RoboChart plugins available in RoboTool, based on the RoboChart model our tool generates.

Epsilon is structured as a series of rules that transform a specified element of the source model to one or more elements of the target model. The rules can have guards restricting their application. Within the body of a rule, elements of the model can be created and modified, and operations can be called. The elements generated by other rules can also be obtained by invoking the rule for the element has not already been translated. With this, we can implement the RoboArch semantics using Epsilon rules in direct correspondence with the rules from Chapter 2.

The existing rules for RoboArch include a top-level rule that transforms a RoboArch *System* to a RoboChart *RCPackage*, which invokes operations that dispatch to pattern-specific translation rules for each layer to generate elements such as interfaces and state machines. The components in these rules are generated by operations that take the arguments specified in the translation rules. The implementation of the translation thus relies on a mixture of rules and operations. For example, our Rule 1 from Chapter 2 is implemented as a rule transforming a *Layer* to a set of *Interfaces*, whereas the rules for generating the component state machines are implemented as operations taking multiple arguments.

Fig. 17 shows, as an example, part of our implementation of a RoboArch MAPLE-K rule. As said, there is a one-to-one correspondence between the rules presented in Chapter 2.1 and the Epsilon implementation. In our example, we can see the implementation of the `ExecuteStateMachine` operation, which corresponds to the translation function in Rule 9. The operation takes the parameters specified in the rule: `execute`, an instance of the *ExecuteComponent*, `amptn`, the overall *MAPLEK* pattern, and `lyrName`, the name of the layer (used to make names unique).

A new `StateMachineDef` named `result` is created, which is returned as the result of the operation at the end, and its `name` component is set as specified in the rule. The `interfaces` and `events` are added to the `result`. The `interfaces` are obtained using an operation `ref` operation, which obtains references to model elements that have already been created: in this case, RoboChart interfaces used by the state machine. The `events` are added by creating new events using an operation `CreateRCEvent`. The *adaptationCompleted* event is conditionally included based on the presence of the *analyse* component. Therefore, it is constructed and added inside an `if` statement, although the variable recording it is declared outside, since it is referred to later.

After the events and interfaces have been added, the states and transitions are created. The states are created as instances of `State`, although the initial junction is instead an instance of `Initial`. Their names are set and they are added to the `nodes` of the `result`. The transitions are instances of `Transition`, which have their `source` and `target` defined as references to the elements of `nodes`, using the variables in which those elements were stored. We omit the remaining transitions for brevity.

Funded by
the European Union

```
operation ExecuteStateMachine(execute: RoboArch!ExecuteComponent,
        amptn: RoboArch!MAPLEK, lyrName: String): RoboChart!StateMachineDef {
    var result = new RoboChart!StateMachineDef;

    result.name = lyrName + "_Execute";

    // add PlanData_events interface
    var PlanDataEventsIface = ref(lyrName + "_PlanData_events", RoboChart!Interface);
    result.interfaces.add(PlanDataEventsIface);

    // add Outputs interface
    var OutputsIface = ref(lyrName + "_Outputs", RoboChart!Interface);
    result.interfaces.add(OutputsIface);

    // make executePlan event
    var executePlan = CreateRCEvent("executePlan");
    result.events.add(executePlan);
    var adaptationCompleted;

    if (amptn.analyse <> null) {
        // make adaptationCompleted event
        adaptationCompleted = CreateRCEvent("adaptationCompleted");
        result.events.add(adaptationCompleted);
    }

    // create states and junctions
    var i0 = new RoboChart!Initial;
    i0.name = "i0";
    result.nodes.add(i0);

    var Initialise = new RoboChart!State;
    Initialise.name = "Initialise";
    result.nodes.add(Initialise);

    var WaitForSignal = new RoboChart!State;
    WaitForSignal.name = "WaitForSignal";
    result.nodes.add(WaitForSignal);

    var SendOutputs = new RoboChart!State;
    SendOutputs.name = "SendOutputs";
    result.nodes.add(SendOutputs);

    var FinishAdaptation = new RoboChart!State;
    FinishAdaptation.name = "FinishAdaptation";
    result.nodes.add(FinishAdaptation);

    // create transitions
    var t = new RoboChart!Transition;
    t.name = "t0";
    t.source = i0;
    t.target = Initialise;
    result.transitions.add(t);

    var t = new RoboChart!Transition;
    t.name = "t1";
    t.source = Initialise;
    t.target = WaitForSignal;
    result.transitions.add(t);
```

Figure 17: Implementation of *ExecuteStateMachine* (Rule 9)

Funded by
the European Union

The RoboChart automatically generated can be used in conjunction with the tools presented later on in this chapter to reason about the RoboSAPIENS application.

6.2 *Circus* in Isabelle

To verify RoboChart models in Isabelle, we have created IsaCircus: a mechanisation of *Circus* that supports the encoding of *Circus* models for theorem proving. In this section, we present the IsaCircus syntax (Section 6.2.1), and the IsaCircus template for presenting Circus models in Isabelle, with an example (Section 6.2.2).

6.2.1 IsaCircus Syntax

In this section, we introduce the IsaCircus syntax and the foundational types and operators we have defined to support the syntax definition. A key feature of our approach is the abstraction of actions and processes, providing a modular and flexible modelling framework while ensuring concise and manageable definitions.

1. Declaring the Abstract Action Type

We begin by defining an abstract type for Circus actions, parametrised by two types: `'e` for events and `'s` for state. This type of declaration is foundational because actions represent the basic building blocks of processes in Circus.

```
typedecl ('e,'s) "action"
```

Here, we only declare an abstract type, meaning no specific internal structure is provided yet. This allows us to use `('e,'s)action` as a placeholder for any type of action that might involve event-based interactions or state transformations.

2. Defining a Type Synonym for Processes

We then declare a type synonym for processes, which are actions with no visible state (that is, actions whose state type is set to unit). This definition helps distinguish processes from actions in general, emphasising that a process in Circus operates without any visible state. The state of a *Circus* process is encapsulated.

```
type_synonym 'e process = "('e, unit) action"
```

Here, `'e process` refers to actions that only involve events but no state updates.

3. Defining Operators for Circus Actions and Processes

We specify action operators for Circus as Isabelle semantic constants. The operators include assignment, sequential composition, interrupt, guard, choice, renaming, and hiding. Each operator is defined with specific type constraints to capture its intended behaviour within a Circus process.

1) Assignment: `cassigns` is used to assign values to the state. It is parameterised by a state-update function, allowing both single and multi-variable assignments.

Funded by
the European Union

```
axiomatization cassigns :: "('s ⇒ 's) ⇒ ('e,'s) action"
```

Here, `('s ⇒ 's)` represents a function that takes a state and returns an updated state. This allows for flexible assignments within an action. The operator returns a `('e,'s)action`, representing an action where the state is updated.

2) Sequential Composition: `cseq` represents a sequence of actions, where, as expected, the first action needs to terminate before the second starts.

```
axiomatization cseq ::
"('e,'s) action ⇒ ('e,'s) action ⇒ ('e,'s) action"
```

The input consists of two `('e,'s)actions`, and it returns a combined `('e,'s)action` that represents the sequence of the two actions in the given order.

3) Interrupt: `cinterrupt` enables an action to be interrupted by another.

```
axiomatization cinterrupt ::
"('e,'s) action  ⇒  ('e,'s) action ⇒ ('e,'s) action"
```

The input includes two `('e,'s)actions` where the first action is interrupted by the second, and it returns a `('e,'s)action` representing the interrupt behaviour.

4) Guard: The `cguard` operator combines a Boolean condition with an action, meaning that the action only executes if the guard condition (predicate) is true.

```
axiomatization cguard ::
"(bool,'s) expr ⇒ ('e,'s) action ⇒ ('e,'s) action"
```

Here, `(bool,'s)expr` represents the guard's predicate, which must be true for the action to proceed. The operator takes a Boolean expression `(bool,'s)expr` and a `('e,'s)action`, and it returns a `('e,'s)action` for the guarded action.

5) Internal Choice: `cIChoice` models a non-deterministic choice among a set of actions, allowing us to define actions where multiple outcomes are possible.

```
axiomatization cIChoice ::
"'i set ⇒ ('i ⇒ ('e,'s) action) ⇒ ('e,'s) action"
```

The operator `cIChoice` takes a set of indices `('i set)`, which defines the possible choices, and a function `('i ⇒ ('e,'s) action)` that maps each index to a corresponding action. It returns a `('e,'s)action` representing the non-deterministic internal choice, where one action is chosen from the set and executed.

6) External Choice: Similar to internal choice, `cEChoice` allows an action to be selected from multiple alternatives, but external inputs define the choice.

```
axiomatization cEChoice ::
"'i set ⇒ ('i ⇒  ('e,'s) action) ⇒ ('e,'s) action"
```

Funded by
the European Union

7) Renaming: `crenaming` applies a renaming function to the events of an action, mapping each event of the input action to a given event in the output action.

```
axiomatization
crenaming :: "('e ↔ 'f) ⇒ ('e,'s) action ⇒ ('f,'s) action"
```

The operator `crenaming` takes a renaming relation and an action, and returns a `('f,'s)action`, allowing for flexible adaptation of events within an action.

8) Hiding: `chide` makes specified events invisible to the external environment, effectively creating events for internal use only.

```
axiomatization chide :: "('e,'s) action ⇒'e set ⇒ ('e,'s)
   action"
```

The operator `chide` takes an action and a set of events `'e set` to be hidden, and returns an action where these events are concealed from the environment.

9) Iterated Interleaving: The `cInterleave` operator models the interleaved execution of a set of actions, where actions execute independently without synchronisation.

```
axiomatization cInterleave ::
"'i set ⇒ ('i ⇒ ('e,'s) action) ⇒ ('e,'s) action"
```

The operator `cInterleave` takes an index set `'i set` and a function `('i ⇒ ('e,'s) action)` mapping each index to an action, and returns an action that represents the interleaved execution of the actions indexed by `'i`.

10) Iterated Parallel Composition: The `cParallelIte` operator extends parallel composition to a set of actions that synchronise on a shared event set, allowing for more complex parallel combinations via iterated parallelism.

```
axiomatization cParallelIte ::
"'e set ⇒ 'i set ⇒ ('i ⇒ ('e,'s) action) ⇒ ('e,'s) action"
```

The operator `cParallelIte` takes an event set `'e set` for synchronisation, an index set `'i set` for identifying actions, and a function mapping each index to an action. It returns an action that represents the parallel execution of all the indexed actions, synchronising on the specified event set.

11) Parameterised Action: The `cParam` operator allows for the parameterisation of an action over a set of indices, creating a more generalised form of an action.

```
axiomatization cParam :: "'i set ⇒ ('e,'s) action ⇒ ('e,'s)
   action"
```

The operator `cParam` takes an index set `'i set` and an action, then returns an action representing the parameterised action over the given index set.

12) Process: The `cProcess` operator is used to define a process from an action, ef-

Funded by
the European Union

fectively encapsulating an action as a process. This operator allows an action to be lifted to a standalone process that can be composed with other processes.

```
axiomatization cProcess :: "('e,'s) action ⇒ 'e process"
```

The operator `cProcess` takes a `('e,'s)action`, which represents a specific set of behaviours involving events `'e` and states `'s`, and returns an `'e process`, encapsulating the given action. This process can then be used as a component to model complex system behaviours through the composition of multiple processes.

13) Prefix: The `cprefix` operator enables an action to wait for an event on a specific channel and then proceed if a certain condition is met.

```
axiomatization cprefix :: "('a,'e) chan ⇒ ('a ⇒ (('s ⇒ bool) ×
   ('e,'s) action)) ⇒ ('e,'s) action"
```

The input for `cprefix` consists of a channel `('a,'e)chan` and a function. The channel `('a,'e)chan` represents the prefixing communication channel. For this `('a,'e)chan` type, `'a` represents the value type associated with the event, which might hold the information that a channel transmits; `'e` represents the event type of the channel.

The function `('a ⇒ (('s ⇒ bool) × ('e,'s) action))` takes an event value of type `'a` and returns a tuple consisting of a state predicate `('s ⇒ bool)` and an action `('e,'s)action`. The predicate determines whether the events are enabled. The prefix operator returns a `('e,'s)action`, representing the complete action that waits for an event, checks a state-based condition, and then executes the appropriate action if the condition is met.

Using the prefix operator, we can further define prefix actions with specific communication types, such as input or output prefixes. The `cinput` operator defines an action that waits for an input value on a specified channel and then executes a subsequent action based on the received input.

```
definition cinput ::
"('a,'e) chan ⇒ ('a ⇒ ('e,'s) action) ⇒ ('e,'s) action"
where "cinput c A = cprefix c (λv. ((λs. True), A v))"
```

The operation `cinput` is implemented using `cprefix`, which has two parameters: the channel c serves as the input channel, the function `λv. ((λs. True), (A v))` takes an input value v and returns a tuple. The first element of the tuple `(λs. True)` is a state condition function that always returns True for any state s, ensuring that the action can proceed regardless of the state, and allowing a value `v` to be received on channel `c`. In the second element `(A v)`, A is a function of type `('a ⇒ ('e,'s) action)`, which means it takes an input value v of type `'a` and returns an action of type `('e,'s)action`. `A v` is the result of applying `A` to the input `v`, producing an action that defines what should happen next based on the input `v`.

The `csync` operator uses `cprefix` to define a synchronisation action that waits for an event on a channel and executes an action when the event occurs.

Funded by
the European Union

```
definition csync :: "(unit,'e) chan ⇒ ('e,'s) action ⇒ ('e,'s)
   action" where
"csync c A = cprefix c (λv. (λs. True, A))"
```

The channel `c` is of type `(unit,'e)chan`. The use of unit indicates that the channel is used purely for synchronisation and does not transmit any data. This reflects the fact that csync is about waiting for an event to occur without processing any input data. `A` is an action of type `('e,'s)action`. Unlike `cinput`, `csync` does not use an input value to determine the action; instead, it always executes the given action A when an event occurs on channel `c`. The function `(λ v. (λ s. True, A))` is used in the definition of `csync` as the second parameter for `cprefix`. The function `(λs. True)` is the state condition indicating that the action can proceed in any state. `A` is the action to be executed when the synchronisation event occurs.

`csync` is defined using `cprefix`, but because the channel type is `(unit,'e)`, it signals that the synchronisation occurs without any data being sent or received. The`v` parameter exists as part of `cprefix`'s interface but does not carry any significant data, as the channel type indicates that there is no meaningful input.

The `coutput` operator defines an action that outputs a value on a channel and then proceeds with a given action, independent of the output value.

```
definition coutput :: "('a,'e) chan ⇒ ('a,'s) expr ⇒ ('e,'s)
   action ⇒ ('e,'s) action" where
"coutput c e A = cprefix c (λv. ((λs. v = e(s)), A))"
```

The operator `coutput` takes three parameters: `c` is a channel of type `('a, 'e)chan`, where `'a` represents the type of data transmitted on the channel; and `e` is an expression of type `('a,'s)expr`. The definition of `expr` is as below.

```
type_synonym ('a,'s) expr = "'s ⇒ 'a"
```

It evaluates to a value of type `'a` based on the current state `s`. It represents the value to be sent on the channel. Finally, `A` is an action of type `('e,'s)action` that is executed after the value is output. The operator `coutput` is defined using `cprefix`. The two parameters passed to `cprefix` are the channel `c` and a function that defines a pair: `(λv. (λs. v = e(s), A))`. The first element `(λs. v = e (s))` is the state condition checking that the received value `v` matches the value computed by `e(s)`. The value `v` is the data transmitted on the channel `c`. The second element `A` of the pair defined by the function is the action to be executed if the condition is met.

The operator `coutinp` combines both `coutput` and `cinput` behaviours. It outputs a value on a channel `c` and then performs an action based on the input value received.

```
definition coutinp :: "('a × 'b,'e) chan ⇒ ('s ⇒ 'a) ⇒ ('b ⇒
   ('e,'s) action) ⇒ ('e,'s) action" where
"coutinp c v A = cprefix c (λ(x, y). (λs. x = v s, A y))"
```

The `coutinp` operator waits for an event on the specified channel `c`, then sends a

Funded by
the European Union

value calculated by `v(s)` based on the current state `s`. Once the output is sent, it waits for a response value of type `'b`. The received value is passed to the function `A`, which determines and returns the action to be executed based on the input.

Some other operators are also defined based on the available axiomatization. `Skip` represents an action that terminates immediately and leaves the state unchanged. It is defined using `cassigns` with the identity function id, which means the state remains the same after the action is executed.

```
definition Skip :: "('e, 's) action" where "Skip = cassigns id"
```

`Stop` represents an action where the process cannot proceed and is blocked. It is defined as an external choice over an empty set, which means no external event can trigger the continuation, effectively halting the process.

```
definition Stop :: "('e, 's) action" where
"Stop = cEChoice ({}::bool set) (λi. Skip)"
```

The operator `cichoice` models an internal choice between two actions `P` and `Q`. The internal choice is non-deterministically decided by choosing either True (which leads to `P`) or False (which leads to `Q`).

```
definition cichoice :: "('e, 's) action ⇒ ('e, 's) action ⇒
('e, 's) action" (infixl "⊓" 59) where
"cichoice P Q = cIChoice {True, False} (λ b. if b then P else
   Q)"
```

The operator `cechoice` represents an external choice between two actions `P` and `Q`. The choice is externally determined by an environment event that chooses True (leading to `P`) or False (leading to `Q`).

```
definition cechoice :: "('e, 's) action ⇒ ('e, 's) action ⇒
('e, 's) action" (infixl "□" 59) where
"cechoice P Q = cEChoice {True, False} (λ b. if b then P else
   Q)"
```

The operator `cparallel` represents a binary parallel composition.

```
definition cpapallel :: "'e set ⇒ 'i set ⇒ ('i ⇒ ('e,'s)
   action) ⇒ ('e,'s) action" where
"cparallel P A Q = cParallelIte A {True, False} (λ b. if b
   then P else Q)"
```

4. Defining Syntactic Constant for IsaCircus Operators

IsaCircus defines the *Circus* action operators as Isabelle semantic constants. After the definition of the operators, we define syntactic constants for each of the semantic constants that provide a readable syntax for these operators in Isabelle, allowing users to write complex logical expressions using concise symbols. For

Funded by
the European Union

example, the syntax for `cinput` is defined as below.

```
syntax "_cinput" :: "id ⇒ pttrn ⇒ logic ⇒ logic" ("_?'(_') →
  _" [61, 0, 62] 62)
```

Here `id` refers to the identifier of the channel (for example, `c`), corresponding to the `('a, 'e)chan` type in the `cinput` definition. The argument `pttrn` is typically used for pattern matching, and here represents the pattern for the input data (for example, `x`), corresponding to the `'a` type input value used in `cinput`. The argument `logic` corresponds to any term constructible in HOL, and here refers to a logical expression or action performed after receiving the input (for instance, `A`). This corresponds to the `('e, 's)action` type in the `cinput` definition.

The syntax definition specifies how user-friendly input notation, such as `c?(x)→ A`, maps to its logical form in Isabelle. The notation `("_?'(_')→ _"[61, 0, 62] 62)` defines how the input is parsed, with `c?(x)→ A` being interpreted as a prefix on `c` waiting for an input value `x` (matching `pttrn`), and, after receiving `x`, performing `A`.

5. Translating between Syntax and Semantics

A further task is to define translation rules that map the syntactic constants to their corresponding semantic constants. For example, the translation rule between `_cinput` (the syntactic constnat above) and `cinput` is as below,

```
translations "_cinput c x P" == "CONST cinput c (λ x. P)"
```

According to the rule above, the syntax `c?(x)→ A` for an input action is translated into the function call `cinput c (λx. P)`. The `CONST` keyword is used in translation rules to mark the defined logical functions as semantic constants. The rule ensures that when Isabelle processes symbolic input, it is mapped correctly to the underlying semantics, or vice versa. In the example rule above, the left side `"_cinput c x P"` is the user-facing notation that is written or presented in Isabelle. The right side `"CONST cinput c (λx. P)"` tells Isabelle to parse this notation as a call to the `"cinput"` function, with `c`, `x`, and `P` as parameters.

In summary, the axiomatic approach for IsaCircus, as described above, allows for the systematic definition and structuring of the types and operators needed for modelling RoboChart processes. The types and operators defined here provide a versatile and extensible foundation for representing process behaviours.

6.2.2 Template of a Circus model in IsaCircus

Based on the types and operators defined in Section 6.2.1, we have designed an IsaCircus template for the RoboChart semantics in *Circus*. It is shown in List 1. This template shows the structure of IsaCircus theory and is also used as the basis for model-to-text transformation from the *Circus* model.

In the IsaCircus template, placeholders and specific formatting conventions are used to help structure and customise the model. Text enclosed in angle brack-

Funded by
the European Union

Listing 1: IsaCircus template

```
1  subsection < Types declaration>
2  type_synonym <type_name> = <type>
3  datatype <Etype> = <literal> (| <literal>)+
4  record <record1> = (<field>:: "<type>")+
5
6  subsection < Constants declaration>
7  consts <const> :: "<type>"
8
9  subsection <Channel Declaration>
10 chantype mychan = (<chan> :: <type>)+
11
12 subsection <Channel Set Declaration>
13 definition "<chanset> = {|<chan> (,<chan>)+|}"
14
15 subsection <Function Definition>
16 definition <fun_name> :: "<type> (⇒ <type>)+"
17     where "<fun_name> (<para>)+ = <definitionOfFun>"
18
19
20 locale <proc>
21 begin
22 actions  is "(mychan, unit) action" where
23     "<action> = <action_definition>"
24     (| "<action> = <action_definition>")+
25
26 definition "MainAction = cProcess <action>"
27 end
```

ets <...> represents placeholders that should be instantiated with specific names or values depending on the particular model being created. For instance, the placeholder <`type_name`> (line 2) should be replaced with a relevant data type name, and <`action`> (lines 26-27) with an appropriate action label.

In the template, the text in dark blue represents Isabelle's command keywords, e.g., **`datatype`** (line 4), **`type_synonym`** (line 2), **`consts`** (line 7), and **`definition`** (lines 18, 19). The green text (e.g., **`where`** (lines 18, 25)) represents reserved keywords.

Each process in IsaCircus is defined within a locale environment (line 26), where a process is represented by a set of actions that define its behaviour. This template provides a structure to define a *Circus* process, covering key elements such as types, constants, channels, channel sets, functions, and process actions.

Types are declared using the **`type_synonym`** (line 2) and **`datatype`** (line 3) keywords, which are used to define type synonyms and data structures relevant to the model. Records are declared using the **`record`** keyword (line 4) to group related fields into structured data types. They correspond to schema types, available in Z and *Circus*, in the scope of the process. Constants are specified with the **`consts`** keyword (line 7), representing fixed values or parameters referenced within the process. These correspond to axiomatic descriptions, from Z and *Circus*, for the process.

Channels are declared using the **`chantype`** keyword (line 10).The clause introduced by **`chantype`** fixes the number of channels you have. No channel should be declared outside **`chantype`**. A Channel Set is then defined with the **`definition`** keyword (line

Funded by
the European Union

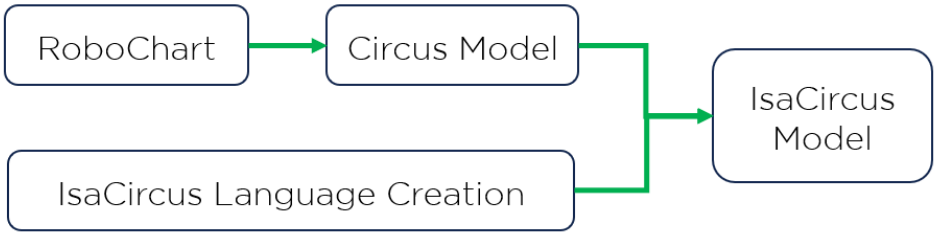


Figure 18: A two-step transformation strategy to convert RoboChart models into IsaCircus.

13), grouping channels into a set for synchronised communication.

Functions can be declared in the clause introduced with the `definition` keyword by filling in the placeholders for their names, parameters, and definitions (line 16). Again, this corresponds to an axiomatic description local to the process.

Each *Circus* process is encapsulated within a locale environment (using the `locale` keyword), enabling modular design and parameterisation of process-specific actions. Within the locale, a set of actions is defined in clauses introduced using the `actions` keyword to specify behaviours that the process can perform. The main action (line 26) is identified using `"MainAction"`, which serves as the core execution body of the process and is modelled using the `cProcess` keyword.

In the next section, we show how to use this template to capture the semantics of RoboChart in IsaCircus, covering also RoboChart models that include an ANN.

6.3 RoboChart transformation to IsaCircus

In this section, a two-step transformation strategy is employed to give semantics to RoboChart models using IsaCircus, as shown in Fig. 18. This process consists of an initial transformation from RoboChart to *Circus* by model-to-model transformation, followed by a model-to-text transformation from *Circus* to IsaCircus.

To transform RoboChart models into *Circus* models, we have defined a Circus metamodel that aligns with the CZT [RNS+05] abstract syntax tree for *Circus*. RoboChart models are equipped with CSP semantics. Therefore, we do not define a new set of formal semantics using *Circus*. Instead, we consistently follow the CSP semantics of RoboChart models, as documented in the RoboChart manual [MRY+21], to generate *Circus* models. Based on the *Circus* metamodel, along with the existing RoboChart metamodel, we have defined and implemented a set of mapping rules to guide the transformation from RoboChart to *Circus* according to the CSP semantics of RoboChart. A summary of the mapping rules is listed below.

- Each RoboChart model is transformed into a *Circus* Spec, which contains a narrative section and a ZSection.
- ZSection is the main body of the Circus model, including a list of Z paragraphs of different types:
  - An Enumeration type in RoboChart is mapped to a FreePara in *Circus*.

Funded by the European Union

  - A RoboChart Primitive type is mapped to a *Circus* GivenPara.
  - A RoboChart Record type is mapped to a *Circus* AxPara with SchBox.
  - A RoboChart function is mapped to a *Circus* AxPara with AxBox.
  - A RoboChart constant declaration is mapped to a AxPara which has schemaText.
  - A RoboChart event is mapped to a *Circus* ChannelPara.
  - An event set is mapped to a ChannelSetPara.
  - In the CSP semantics, each RoboChart component is a CSP process. However, this is not the case in our *Circus* semantics. The high-level components of RoboChart are transformed into *Circus* Process Paragraphs, including Module, Module Memory, Buffer, and Controller. This is to allow us to use the full range of CSP operators to combine the actions that capture the individual elements of the RoboChart model.
  - The rest of the RoboChart components are transformed into *Circus* Action Paragraphs. The components to be transformed into *Circus* Action Paragraphs include Controller Memory, State Machine, Variable Memory, Shared Variable Memory, Individual Variable Memory, Individual Shared Variable Memory, Node Container, Transitions, and Node. A Node can be a junction, a final state, a simple state, or a composite state. Each node is transformed into a set of Action Paragraphs.

- The exception operator has been used in the CSP semantics of RoboChart. As the exception operator is not supported in *Circus* we avoid it by introducing synchronisation on termination.
- Channel declarations in the CSP semantics are not documented in the RoboChart Manual, but needed to be implemented, as they are in the CSP automatic generator.
- Modular design allows complex systems to be decomposed into multiple modules, each with nested submodules. This approach helps to organise the hierarchical structure of complex systems, especially when modelling large systems or subsystems with intricate behaviours. Modular design also allows components in different modules to have the same name, distinguished by identifying their scope. Both RoboChart and CSP support modular design: RoboChart via packages, and CSP via modules. However, *Circus* does not support modular design for processes, so when we transform RoboChart to *Circus*, we need to name each component using its fully qualified name.

The complete set of mapping rules has been implemented using the Eclipse Epsilon framework, specifically the Epsilon Object Language (EOL), enabling the full automation of the transformation process from RoboChart to Circus models.

For the second step of the transformation strategy, using the syntax defined in Section 6.2.1 and the template in Section 6.2.2, we have implemented the model-to-text transformation from *Circus* to generate IsaCircus using Epsilon Generation

Funded by
the European Union

Language (EGL). Listing 2 sketches the IsaCircus encoding of the process for the *AnglePIDANN* controller defined by Rule 2 in Appendix B.

As illustrated in Listing 2, the notation of the IsaCircus definition of `AnglePIDANN` is very similar to that indicated in Rule 2. In detail, we declare an event type `ANNChan` (line 1 in Listing 2). It includes all channels used in the semantics. The definition of the activation function, `relu`, is shown in line 7. We show part of the `locale` for `AnglePIDANN` (lines 9-23). Three of the nine actions of `AnglePIDANN` are shown in Listing 2, among which the main action `MainAction` is identified as `CircANN` (line 22) using the `cProcess` operator.

Listing 2: AnglePIDANN example in IsaCircus

```
1  chantype ANNchan = terminate          :: "unit"
2                   layerRes            :: "nat × nat × real"
3                   nodeOut             :: "nat × nat × nat × real"
4                   adiff               :: "real"
5                   ...
6
7  definition relu :: "real ⇒ real" where "relu x = max 0 x"
8
9  locale <AnglePIDANN>
10 begin
11 actions is "(ANNchan, unit) action" where
12
13 "Collator(l, n, i :: nat, sum :: real) =
14     (i = 0) & layerRes.l.n!relu(sum + biases(l)(n)) → Skip □
15     (i > 0) & nodeOut.l.n.i?x → Collator(l,n,(i-1),(sum+x))"  |
16 ...
17 "ANN = (HiddenLayers [| {|layerRes.1.1|} |] OutputLayer);; ANN" |
18
19 "CircANN = ((Interpreter [| {|layerRes.0, layerRes.2|} |] ANN) \ {
     layerRes|}) △ (terminate → Skip)"
20
21 definition "MainAction = cProcess CircANN"
22 end
```

6.4 Reasoning about ANN

This section provides an overview of our toolchain for modelling, validating, and verifying AI-enabled robotic software using RoboChart (see Figure 19). We have built upon the work in the previous section and existing work on generating a CSP semantics for RoboChart to mechanise the semantics of ANN controllers.

Our modelling approach is mechanised as part of RoboTool, where RoboChart and its two semantics (one in *Circus*, implemented using the Epsilon framework as described in the previous section; and one in CSP, mechanised via Xtend, a Java variant[15]) are mechanised to support transparent automatic generation. We have

[15] eclipse.dev/Xtext/xtend/

Funded by
the European Union

leveraged EMF to implement the RoboChart metamodel, and the Sirius platform [16] to enable graphical modelling. We enable textual editing of models via an Xtext grammar [17]. Further, we mechanise the set of well-formedness conditions for these models through a validation checker written in Xtend. See Figure 20 for a screen-shot of the extended RoboTool.

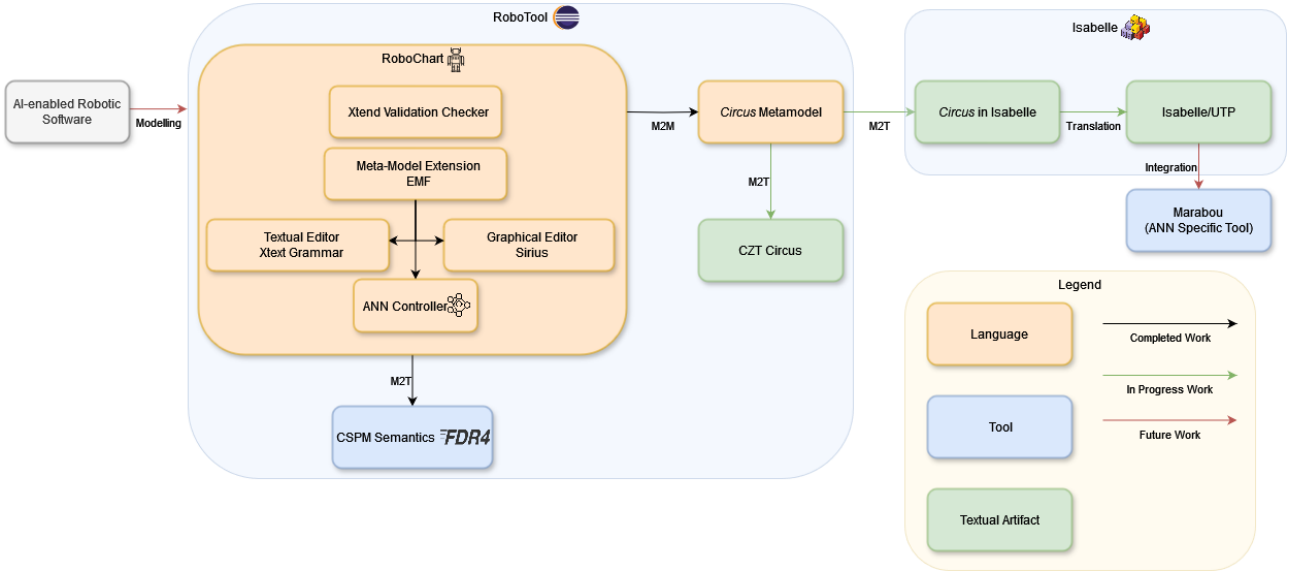


Figure 19: Our toolchain for the modelling and verifying AI-enabled robotic software. M2M: model-to-model; M2T: model-to-text.

We mechanise the semantics of RoboChart using a model-to-text translation from RoboChart to CSPM[18], enabling the use of FDR4[19] for model checking. This allows process verification of AI-enabled systems, which can assert structural properties on these systems, but not functional properties.

We enable verification of these models using Isabelle, following the approach presented in the previous section. As explained in Chapter 4, however, to enable use of AI-specific verification tools, such as Marabou, we require a reachability condition expressed as a predicate on a vector space. To formally obtain this condition, we use Isabelle/UTP [FZN+19]. This allows us to develop a formalism that captures reactive processes using alphabetised predicates. Using this theory, we can create lemmas that link the semantics of RoboChart directly to ANN reachability conditions. When we have these conditions, we can use dedicated ANN theorem provers.

6.5 Deadlock-freedom verification in Isabelle

This section describes our approach to verifying deadlock freedom of IsaCircus models with the Isabelle proof assistant. We develop this method by building on

[16] eclipse.dev/sirius
[17] eclipse.dev/Xtext/
[18] cocotec.io/fdr/manual/cspm.html
[19] cocotec.io/fdr/

Funded by the European Union

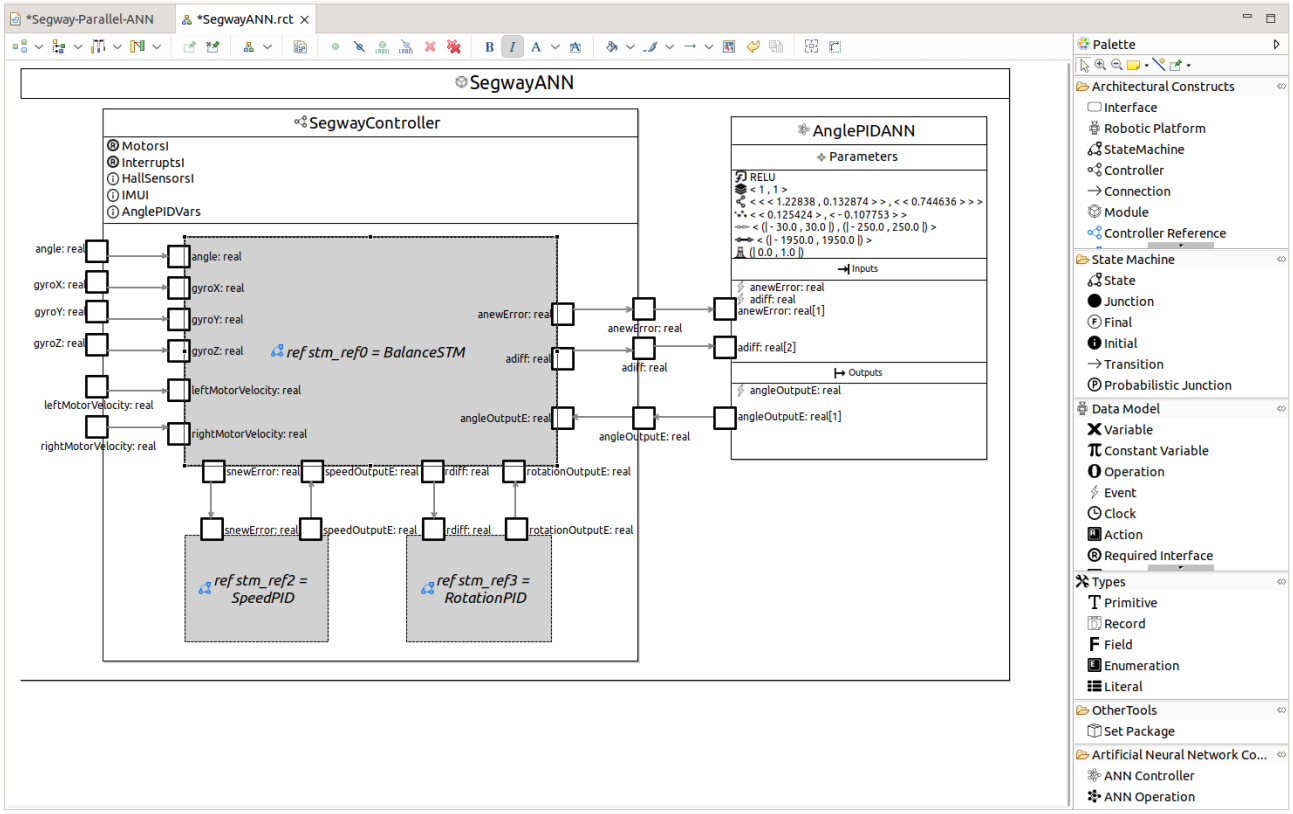


Figure 20: A screenshot of our tool: RoboTool with ANN components. Users can specify ANN components using the *Artificial Neural Network Components* tab on the palette, located on the right-hand side. We enable the specification of the complete software module via the other components in the palette.

proof principles from HOL-CSP [TWY20] and adapting them to IsaCircus. The approach leverages the syntactic and semantic similarity between the two languages, with *Circus* including the whole of CSP, and having a UTP semantics based on an extended version of the failures-divergences semantics of CSP. (The UTP theory of *Circus* allows for programming variables in the state, and defines a complete lattice.) So, we use the HOL-CSP results to guide the expression of verification laws and support automation through IsaCircus-specific tactics.

Beyond checking for deadlock-freedom alone, our method also supports reasoning about invariants embedded in the model. This is achieved by using an assumption–guarantee strategy in the process definitions. For verification, each process is automatically enriched during model-to-model transformation with guarantee conditions, including invariants that it must preserve. These guarantees are encoded directly into the semantics, such that violating a guarantee results in an immediate deadlock.

As a result, a successful proof of deadlock freedom ensures not only that the system has no deadlocks, but also that all guarantee conditions, interpreted as invariants, are upheld throughout execution. Conversely, if deadlock freedom cannot be established, the failure may be caused either by a genuine deadlock or by a violation of the specified guarantees. So, our work can be used to prove deadlock freedom,

Funded by
the European Union

as we describe here, and also for invariant checking.

We begin this section by presenting a semantics enrichment for deadlock-freedom formal verification (in Section 6.5.1). The proof method used to establish deadlock freedom is then subsequently presented (in Section 6.5.2).

We have applied the verification technique to a more detailed version of the navigation robot discussed in Section 2. For clarity and brevity, we use an abstract example to illustrate the semantics enrichment process in Section 6.5.1. We discuss in detail the verification proof for that navigation robot example in Section 6.5.2.

6.5.1 Semantics enrichment for deadlock-freedom checking

To facilitate deadlock freedom analysis, we have enriched the semantics of the Circus model without altering its original meaning.

Conversion of IsaCircus into Parametrised recursive Processes Our IsaCircus models, which are automatically generated from RoboChart models, have no state variables. To facilitate deadlock-freedom verification in Isabelle, we rewrite IsaCircus actions into parametrised recursive actions. A key transformation involves the `Trans` process, which captures the transition behaviour of a RoboChart state machine. While `Trans` generated from RoboChart in IsaCircus does not explicitly reference the active node, we introduce a parameter `st` representing the current active state.

According to the RoboChart Circus semantics, Nodes and Transitions synchronise on trigger events. Each trigger event name encodes the source node identifier (e.g., `b__!NID_s1.in`), ensuring that a transition is only enabled when the system is in its corresponding source state. That is, transitions are semantically guarded by their source node, even if no explicit guard appears in the model.

Adding an explicit guard `st = src` to each transition does not alter the behaviour of the model. It simply makes the existing semantic constraint explicit. Since synchronisation on trigger events is only possible when the active node matches the transition's source, the added guard is redundant from a behavioural perspective.

However, this transformation significantly benefits verification. By making the active state an explicit parameter, we enable structured reasoning in Isabelle, such as case analysis over the possible values of st.

We illustrate this alteration with a simple example. Fig. 21 illustrates a RoboChart state machine named `stm04`, which declares an integer variable `v1`. The initial node `i` transits to state `s0` with a transition action assignment `v1 = 2`. From `s0`, there is a single transition to `s1` with guard `v1` $\geq$ `1`. From `s1`, there are two transitions back to `s0`, one triggered by the event `a`, and the other with guard `v1 < 1`, respectively. The IsaCircus semantics of this RoboChart model can be automatically calculated using our tool; we only show here the transition action in Listing 3.

Listing 3 uses IsaCircus's **`actions`** block to specify the behaviour of two actions: the self-looping action `SSTOP`, and the recursive action `Trans`. As shown in Listing 3, `Trans` reads an input value via `get_v1`, and then branches through a sequence of internal and external actions based on the value of the shared variable `v1` and the

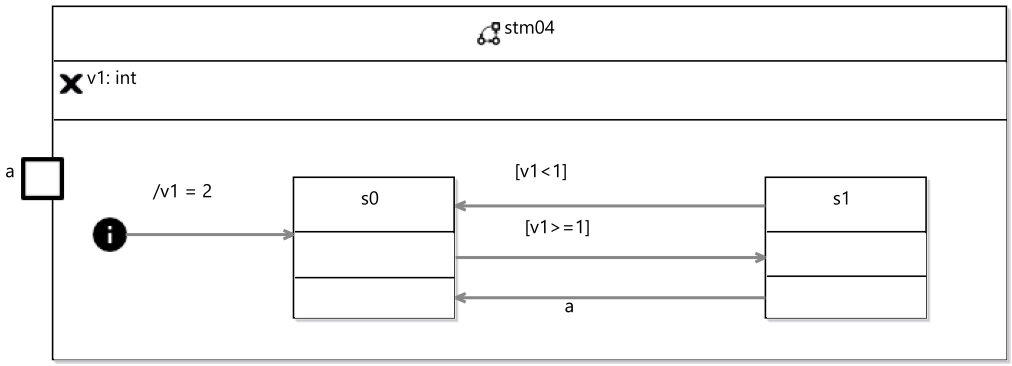


Figure 21: A RoboChart example.

Listing 3: Trans example in IsaCircus

```
locale <Trans>
begin
actions is "(mychan, unit) action" where
"SSTOP = share → SSTOP" |
"Trans =
    (SSTOP △ (get_v1?v1 → (
     ((internal_.NID_i0 → (SSTOP △ (set_v1!2 → Skip)));; (enter_s0
        → Skip)
     □
     (v1 ≥ 1) & ((internal_.NID_s0 → Skip);; ((SSTOP △ (exit →
        Skip));; (SSTOP △ ((exited → Skip);; (enter_s1 → Skip)))))
     □
     (v1 < 1) & (((internal_.NID_s1 → Skip);; ((SSTOP △ (exit →
        Skip));; (SSTOP △ ((exited → Skip);; (enter_s0 → Skip))))))
     □
     ((a__in.NID_s1 → Skip);; ((SSTOP △ (exit → Skip));; (SSTOP △
        ((exited → Skip);; (enter_s0 → Skip))))))
    ;; Trans)))"
end
```

current control state. Each branch uses guarded actions and sequential composition to model specific transition paths, followed by a tail recursion. The enriched model with parametrisation and modified recursion is shown in Listing 4.

Compared to the original IsaCircus model in Listing 3, in Listing 4 `Trans` differs in two ways: state parametrisation and distributed recursion. The active state is made explicit via a parameter `n`, representing the active node in the state machine. Because `Trans` is now parametrised by `n`, the recursion is no longer applied at the top level, but pushed into each choice branch. That is, every path that continues the execution invokes `Trans` with an updated state argument.

Semantics enrichment of invariants In the verification of deadlock freedom for RoboChart models, our focus is on the actions `Trans`, which capture the behaviours of all transitions in the state machines. It is the transitions that can become all

Funded by
the European Union

Listing 4: Trans example in IsaCircus with parameterization and recursive features

```
locale <Trans>
begin
actions is "NIDS ⇒ (mychan, unit) action" where
"SSTOP = share → SSTOP" |
"Trans(n) =
    (SSTOP △ (get_v1?v1 →
        ((n = NID_i0) & ((internal_.NID_i0 → (SSTOP △ (set_v1!2 →
           Skip)));; (enter_s0 → Trans(NID_s0)))

          □
          (n = NID_s0) & (((v1 ≥ 1)) & ((internal_.NID_s0 →
             Skip);; ((SSTOP △ (exit → Skip));; (SSTOP △ ((exited →
              Skip);; (enter_s1 → Trans(NID_s1)))))))
          □
          (n = NID_s1) & ((v1 < 1) & ((internal_.NID_s1 → Skip);;
             ((SSTOP △ (exit → Skip));; (SSTOP △ ((exited →
             Skip);; (enter_s0 → Trans(NID_s0)))))))
          □
          (n = NID_s1) & ((a__in.NID_s1 → Skip);; ((SSTOP △ (exit →
              Skip));; (SSTOP △ ((exited → Skip);; (enter_s0 →
             Trans(NID_s0)))))))))"
end
```

disabled and, therefore, lead to a deadlock.

We first consider the potential causes of deadlock. Our analysis begins with the transitions in state machines. When focusing solely on the behaviour of transitions, one primary cause of deadlock is the presence of a state (excluding final states) with not enough outgoing transitions. Additionally, outgoing transitions may become disabled due to actions performed during transitions or within states, which may cause the transition conditions to evaluate to `false`.

Apart from such internal causes, deadlock may also result from external factors. For instance, a parallel state machine may write to a shared variable used in the state machine to be checked in a way that invalidates the condition of a transition, thus disabling it and potentially leading to deadlock.

To address external causes of deadlock in our verification, we enrich the semantics of our model using invariants. This is achieved through an assumption–guarantee strategy, where assumptions state that values read from the environment satisfy the invariant, and guarantees ensure that variable updates made by the model preserve it. Specifically, we enrich the semantics of shared variable access by inserting assumptions and guarantees that restrict reading and writing actions.

When the model under verification reads a shared variable, we assume that, even though the environment may have updated its value, the value still satisfies a given invariant. Likewise, when the model itself writes to a shared variable, it must guarantee that the new value continues to satisfy the invariant. Otherwise, a violation occurs, and the model transitions to `STOP`, representing a deadlock.

Funded by
the European Union

Listing 5: Trans example in IsaCircus with assumption-guarantee feature

```
1  locale <Trans>
2  begin
3  abbreviation "assume b Q P ≡ (if b then P else aviol → Q)"
4  abbreviation "guar b P ≡ (if b then P else gviol → STOP)"
5  actions is "(mychan, unit) action" where
6  "SSTOP = share → SSTOP" |
7  "Trans(n) =
8          (SSTOP △ (get_v1?v1 → (assume (v1 ≥ 1) (Trans(n))
9          ((n = NID_i0) & (((internal_.NID_i0 → (SSTOP △ (guar
              (2::int ≥ 1)(set_v1!2 → Skip))));; (enter_s0 →
              Trans(NID_s0))))
10           □
11           (n = NID_s0) & ((((v1 ≥ 1)) & (((internal_.NID_s0 →
                Skip);; ((SSTOP △ (exit → Skip));; (SSTOP △ ((exited →
                 Skip);; (enter_s1 → Trans(NID_s1)))))))))
12           □
13           (n = NID_s1) & ((((v1 < 1)) & (((internal_.NID_s1 →
                Skip);; ((SSTOP △ (exit → Skip));; (SSTOP △ ((exited →
                 Skip);; (enter_s0 → Trans(NID_s0)))))))))
14           □
15           ((n = NID_s1) & ((a__in.NID_s1 → Skip);; ((SSTOP △ (exit
               → Skip));; (SSTOP △ ((exited → Skip);; (enter_s0 →
                Trans(NID_s0))))))))))"
16 end
```

To enable the above approach, we introduce two new events: `aviol` for assumption violation, and `gviol` for guarantee violation. The enriched IsaCircus model that uses these events for the RoboChart example in Fig. 21 is shown in Listing 5.

In this model, two abbreviations are defined to express common contract-based behaviours concisely. The first one, `assume b Q P`, means that if `b` is true, the system proceeds with `P`. However, if the assumption is violated, the system raises an assumption violation via the `aviol` channel and then executes the fallback process `Q`; in the case illustrated in Listing 5 (line 8), this is `Trans(n)`. This mechanism captures the idea that the system behaves normally when its assumptions are satisfied; otherwise, it reacts with a violation signal and recurses back to `Trans(n)`.So, if the violation happens, it is signalled, and the process continues to accept the next value. A deadlock is not reported because this is not caused by `Trans(n)`.

When verifying the deadlock freedom of `Trans` for this example, we introduce the invariant on the shared variable `v1` that `v1` should always be greater than `1`. This invariant can be derived automatically from the RoboChart models by calculating its weakest condition using the conditions of each transition. This automation will be implemented in a future phase of the project. Therefore, for the assumption we have `v1` ≥ 1 (line 8). So, we are, in effect, considering two branches in the proof: one where the assumption holds, and one where it does not. However, our interest lies in the case where the assumption is satisfied, since deadlocks that result from assumption violations are attributed to the environment's behaviour rather than to

Funded by
the European Union

that of the verified process.

To explicitly represent the assumption-violation case in the model, we use, via the `assume` construct, the action `aviol →Trans(n)`. The deadlock freedom of this construct is discharged intuitively, as `Trans(n)` is recursive. Therefore, in the deadlock-freedom proof, we then focus exclusively on the assumption-satisfaction case, which is the true subject of verification of interest here.

The second abbreviation, `guar b P`, represents that if condition `b` holds, it continues as `P`. In the definition of `Trans`, the action `P` models the writing of the variable value to the memory via a `set` channel. According to the invariant of the RoboChart example, we have the guarantee condition that the written value `2` should be greater than `1`, i.e., `2::int ≥1` (Listing 5 line 8). But if the guard is false, it raises a guarantee violation through the `gviol` channel and then halts execution with `STOP`.

This enrichment is introduced for checking the deadlock freedom of the model, and it does not alter the original semantics of the RoboChart model. When `aviol` is not available, the process behaves exactly as before. When aviol is offered, the process behaves as `aviol →Trans(n)`. Therefore, during the deadlock-freedom analysis of the transitions of an individual state machine, we restrict our attention to execution paths where all assumptions on shared variables are satisfied. Paths that violate these assumptions are considered outside the scope of this analysis, as handling them requires a system-level view where interactions between processes are taken into account. Such assumption-violating paths become relevant only when analysing a larger system composed of multiple interacting processes, where it is necessary to ensure that the assumptions of all components are mutually compatible. As a result, in our proof, we only need to consider traces that respect all stated assumptions. This ensures that any deadlock detected within this analysis reflects the behaviour of the process itself under the assumption of a well-behaved environment, and is not caused by unexpected interference from other processes. The advantage of using assumptions is that it enables modular reasoning and allows us to prove deadlock freedom for individual components under assumption satisfaction. However, ensuring system-wide deadlock freedom requires verifying that all component assumptions can be satisfied simultaneously during system operation.

Similarly, the use of `guar` does not alter the original semantics of the RoboChart model. A guarantee violation corresponds to undefined or invalid behaviour. In the formal model, the construct `gviol <rightarrow> STOP` explicitly models this outcome: any execution path that violates the guarantee will be forced to terminate at `STOP`. This makes such violations observable and ensures that their impact is properly accounted for during verification, including deadlock analysis. This mechanism does not introduce any new valid behaviours. Provided that the guarantee conditions hold which is the case in correct designs the behaviour is that of the original model. Hence, the enrichment is a conservative extension: it preserves the original semantics for executions that respect the invariant, while exposing internal errors for analysis. In particular, the semantics of the original model remains unchanged along traces where all components of the RoboChart model maintain the invariant in their updates to shared variables.

Funded by
the European Union

6.5.2 Proof method of deadlock freedom

Our strategy for verifying deadlock freedom combines normalisation with natural deduction reasoning, applying both techniques throughout the proof process.

We incrementally apply normalisation during the construction of the proof. This allows us to simplify individual parts of the action expression as needed, transforming complex constructs such as guarded choices or sequential compositions into a simplified form in context, thereby facilitating proof.

The application of a natural deduction rule is often preceded or accompanied by normalisation, enabling us to keep the proof tractable and modular. This coordinated use of normalisation and deduction ensures that the action remains amenable to rule application and that the proof steps preserve structure.

In this section, we begin with the necessary definitions and foundational lemmas. We then present the normalisation laws and the natural deduction rules used to derive deadlock-freedom results, followed by our integrated proof method.

Definitions and Theorem We first define several operators that facilitate the specification of the deadlock freedom property for proof.

Definition 6.1 (DF)**.**

$$\mathrm{DF}_A \equiv \mu X.\ \Box_{\alpha \in A}\ \alpha \to X$$

This operator defines a action that ensures deadlock freedom over a set A of events. It uses the least fixed-point operator μX to construct a action that is always ready to engage in some event from A, and then repeats. As a result, the action never reaches a state where all progress is blocked, thus ensuring deadlock freedom.

Definition 6.2 (deadlock_free)**.**

$$\text{deadlock\_free}\ P \equiv \mathrm{DF}_\Sigma \sqsubseteq P$$

This definition expresses the property of deadlock freedom for P in terms of refinement. It checks whether the action P refines DF_Σ, which represents an action that is always ready to perform an event from Σ. If this refinement holds, it indicates that P can match the behaviour of DF_Σ and thus does not get stuck, ensuring deadlock freedom.

Definition 6.3 (GlobalNdet_iterations)**.**

$$\mathrm{DFI}_A(P) \equiv \prod_{i \in \{0<..\}} \left(\prod_{\alpha \in A}^{i} \alpha \to P\right)$$

The operator DFI_A, which stands for Deadlock Freedom Inductive, defines a generalised iteration over a set of events $A$by allowing P to be prefixed by any finite number of nondeterministic choices. The iteration index $i \in \{0 < ..\}$ determines the nesting depth of the prefixing: i layers of nondeterministic choices are applied in sequence. At each layer, a nondeterministic choice over all possible events $\alpha \in$ A is

Funded by
the European Union

made, forming a prefix $\alpha \rightarrow$. The resulting structure can be viewed as an i-fold iterated prefixing, producing a finite, but arbitrarily long, sequence of nondeterministic interactions before executing P.

Definition 6.4 (GlobalNdet_iterations_star)**.**

$$\mathrm{DFI}^{*}_{A}(P) \equiv \prod_{i \in \mathbb{N}} (\prod_{\alpha \in A}^{i} \alpha \rightarrow P)$$

This operator generalises $\mathrm{DFI}_{A}(P)$ by allowing zero or more layers of nondeterministic prefixing. The iteration index $i \in \mathbb{N}$ allows the iteration to start from $i = 0$, meaning the action may perform zero or more prefix interactions before executing P. For $i = 0$, the action behaves as P immediately; for $i > 0$, an i-fold sequence of nondeterministic choices over events $\alpha \in A$ is performed first. Hence, the operator expresses behaviours where the action may optionally delay P through an arbitrary number of interactions.

Theorem 5 (DFI_FD_imp_ddlf)**.**

$\mathrm{DFI}_{\Sigma}(P) \sqsubseteq P \Rightarrow$ deadlock_free P

This theorem establishes a fundamental property of the DFI_{Σ} operator. Specifically, it asserts that if an action P is capable of refining the behaviour $\mathrm{DFI}_{\Sigma}(P)$, then P is guaranteed to be deadlock-free. Intuitively, this holds because $\mathrm{DFI}_{\Sigma}(P)$ represents processes that can engage in any number of interactions without blocking: after each prefix, the system remains ready to continue. Thus, $\mathrm{DFI}_{\Sigma}(P)$ is inherently deadlock-free. Since P refines $\mathrm{DFI}_{\Sigma}(P)$, it cannot introduce deadlocks that are absent in $\mathrm{DFI}_{\Sigma}(P)$. Therefore, P must also be deadlock-free, as refinement preserves deadlock freedom in this case.

Normalisation Lemmas We define a proof method named `normalisation` to automate structural normalisation of actions. Its purpose is to eliminate high-level syntactic constructs such as guards, external choice, and sequencing by rewriting them into a simplified core form. The method uses a collection of algebraic equivalence lemmas described below to flatten and standardise the structure of actions systematically. This normalised form is better suited for reasoning.

The normalisation lemmas provide algebraic rewrite rules, allowing actions to be simplified or brought to a standard form. They are not used to derive semantic properties directly, but are essential for applying deduction rules.

Lemma 1 (bi_extchoice_norm)**.**

$$b \ \& \ P \ \Box \ c \ \& \ Q \ = \ \Box_{i \in \{0,1\}} (\mathit{if}\ i = 0\ \mathit{then}\ b\ \mathit{else}\ c) \ \& \ (\mathit{if}\ i = 0\ \mathit{then}\ P\ \mathit{else}\ Q)$$

A binary guarded external choice can be uniformly represented as an indexed external choice over a finite set. This is useful for unifying patterns and applying general rules over indexed structures.

Funded by
the European Union

Lemma 2 (biextchoic_norm)**.**

$$\left(\Box_{i\in\{0\ldots n\}}\ b(i)\ \&\ P(i)\right)\ \Box\ c\ \&\ Q\ =$$

$$\Box_{i\in\{0\ldots n+1\}}\ \left(\mathit{if}\ i\leq n\ \mathit{then}\ b(i)\ \mathit{else}\ c\right)\ \&\ \left(\mathit{if}\ i\leq n\ \mathit{then}\ P(i)\ \mathit{else}\ Q\right)$$

This lemma shows that an external choice over a range of guarded branches can be extended by one additional guarded branch and uniformly rewritten as an indexed external choice over an enlarged set. It enables reasoning about binary or incremental additions to a nondeterministic choice construct using a uniform, index-driven representation — useful for automation and rule application.

Lemma 3 (biextchoic_norm_nguard_prefix)**.**

$$\left(\Box_{i\in\{0\ldots n\}}\ b(i)\ \&\ P(i)\right)\ \Box\ a\rightarrow Q\ =$$

$$\Box_{i\in\{0\ldots n+1\}}\ \left(\mathit{if}\ i\leq n\ \mathit{then}\ b(i)\ \mathit{else}\ \mathit{True}\right)\ \&\ \left(\mathit{if}\ i\leq n\ \mathit{then}\ P(i)\ \mathit{else}\ a\rightarrow Q\right)$$

This lemma demonstrates how to integrate a prefixed action $a \rightarrow Q$, which does not have an explicit guard, into a uniform indexed guarded external choice. By extending the index range from $\{0\ldots n\}$ to $\{0\ldots n{+}1\}$, we can incorporate the unguarded prefixed action as a special case where the guard is set to the always-true condition True. This transformation is essential in normalisation steps where all branches of an external choice need to be brought into the same syntactic and semantic form for structural processing, reasoning, or tool-based refinement.

We also use three lemmas to normalise the sequence operator.

Lemma 4 (read_Seq)**.**

$$c\,?a\in A\rightarrow P(a)\,;\,Q\ =\ c\,?a\in A\rightarrow(P(a)\,;\,Q)$$

This rule expresses that input-prefixed sequencing distributes over the prefix. That is, sequencing Q after an input-prefixed choice is equivalent to sequencing Q after each branch of that choice.

Lemma 5 (write_Seq)**.**

$$c\,!a\rightarrow P\,;\,Q\ =\ c\,!a\rightarrow(P\,;\,Q)$$

This lemma shows that sequencing Q after an output-prefixed action $c!a \rightarrow P$ is equivalent to distributing the sequencing inside the prefix: first perform the output, then continue with $P; Q$.

Lemma 6 (write0_Seq)**.**

$$a\rightarrow P\,;\,Q\ =\ a\rightarrow(P\,;\,Q)$$

This lemma shows that the sequencing operator distributes over a prefix. Performing event a, then proceeding with $P; Q$, is semantically equivalent to first executing $a \rightarrow P$, and then sequencing Q into the continuation.

Funded by
the European Union

Natural Deduction Rules With the action model normalised, we now introduce the rules for proving the deadlock freedom. These rules support the derivation of semantic properties, such as deadlock-freedom, through structured, logic-driven proof steps. Many of these are based on stepwise refinement.

Lemma 7 (df_step_param_intro)**.**

$$\frac{\Gamma \vdash \forall x.\ P(x) = Q(x) \qquad \Gamma \vdash \forall x.\ \mathrm{DFI}_{\Sigma}\left(\sqcap_{x \in \Sigma} P(x)\right) \sqsubseteq Q(x)}{\Gamma \vdash \mathrm{deadlock\_free}\left(\sqcap_{x \in \Sigma} P(x)\right)}\ (\text{DfStepParam})$$

This rule states that under context Γ, if for all x, $P(x)$ is equal to $Q(x)$, and $\mathrm{DFI}_{\Sigma}\left(\sqcap_{x \in \Sigma} P(x)\right)$ is refined by each corresponding $Q(x)$, then the internal choice over all $P(x)$, written as $\sqcap_{x \in \Sigma} P(x)$, is deadlock free.

This lemma is particularly useful for proving the deadlock freedom of parametrised systems. Instead of proving directly that the external choice over all components is deadlock free, one can (1) express each component $P(x)$ as an equivalent form $Q(x)$, and (2) show that a global iterative refinement structure si refined by each such $Q(x)$. Once these steps are completed, this rule provides a way to conclude that the system as a whole does not deadlock as required.

Lemma 8 (prefix_proving_Mndetprefix_ref)**.**

$$\frac{\Gamma \vdash a \in B \qquad \Gamma \vdash P(a) \sqsubseteq R}{\Gamma \vdash \sqcap\ a \in B \to P(a) \sqsubseteq a \to R}\ (\text{Prefix-Intro})$$

There are two premises in this lemma:

- $\Gamma \vdash a \in B$, means that under context Γ, the name a is an element of the set B.
- $\Gamma \vdash P(a) \sqsubseteq R$, means that under the same context Γ, the action $P(a)$ is refined by R in the *Failures-Divergences* semantics.

The rule expresses that if each possible prefixed $P(a)$ (for $a \in B$) individually is refined by R, then the internal choice over the set B is refined by the single prefix leading to R. In other words, a uniform refinement of all branches implies refinement of the overall nondeterministic prefix structure.

Lemma 9 (prefix_proving_Mndetprefix_UNIV_ref)**.** *This is a specialisation of* Prefix-Intro *with* $B = \Sigma$.

Lemma 10 (one_step_ahead)**.**

$$\frac{\Gamma \vdash \sqcap\ a \in \Sigma \to \mathrm{DFI}^{*}_{\Sigma}(P) \sqsubseteq Q}{\Gamma \vdash \mathrm{DFI}_{\Sigma}(X) \sqsubseteq Q}$$

$$\frac{\Gamma \vdash \mathrm{DFI}_{\Sigma}(P) \sqsubseteq Q}{\Gamma \vdash \sqcap\ a \in \Sigma \to \mathrm{DFI}^{*}_{\Sigma}(P) \sqsubseteq Q}$$

Funded by
the European Union

Statement:

$$\sqcap\ a \in \Sigma \to \mathrm{DFI}^{*}_{\Sigma}(P) \sqsubseteq Q \quad \Longleftrightarrow \quad \mathrm{DFI}_{\Sigma}(P) \sqsubseteq Q$$

This lemma states that Q refining an internal choice over all prefixed versions of $\mathrm{DFI}^{*}_{\Sigma}(P)$ (one step ahead) is equivalent to refining $\mathrm{DFI}_{\Sigma}(P)$, the standard action iteration (at least one step).

Lemma 11 (generalised_refine_guarded_extchoice)**.**

$$\frac{\Gamma \vdash \exists i \in I.\ b(i) \qquad \Gamma \vdash \forall i \in I.\ b(i) \Rightarrow \sqcap\ a \in \Sigma \to X \sqsubseteq P(i)}{\Gamma \vdash \sqcap\ a \in \Sigma \to X \sqsubseteq \Box_{i \in I} b(i)\ \ \&\ \ P(i)}\ \text{(Guarded-ExtChoice)}$$

This rule expresses that, under context Γ, if there exists some $i \in I$ such that the guard $b(i)$ holds, and for every such i, the internal choice action $\sqcap\ a \in \Sigma \to X$ is refined by $P(i)$. The same refinement holds for the guarded external choice over all $i \in I$. This lemma is useful when you wish to prove refinement of a guarded external choice by showing refinement for each branch individually. The existence premise ensures that at least one guard is enabled, making the overall action deadlock-free.

Lemma 12 (eat_lemma)**.**

$$\frac{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(P) \sqsubseteq Q}{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(P) \sqsubseteq a \to Q}\ \text{(Eat)}$$

This rule expresses that if under context Γ, the iteration $\mathrm{DFI}^{*}_{\Sigma}(P)$ is refined by Q, then if we add a prefix a in front of Q, the iteration $\mathrm{DFI}^{*}_{\Sigma}(P)$ is still refined by $a \to Q$.

Lemma 13 (no_step_refine)**.**

$$\frac{}{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(P) \sqsubseteq P}\ \text{(No-Step)}$$

This lemma states that an iteration $\mathrm{DFI}^{*}_{\Sigma}(P)$ is always refined by P itself. The reasoning is intuitive: since $\mathrm{DFI}^{*}_{\Sigma}(P)$ may behave like P, or iterate multiple times then behave like P, it is more permissive than P.

Lemma 14 (write_proving_Mndetprefix_ref)**.**

$$\frac{\Gamma \vdash c(a) \in B \qquad \Gamma \vdash P(c(a)) \sqsubseteq R}{\Gamma \vdash \sqcap_{a \in B}\ a \to P(a) \sqsubseteq c!a \to R}\ \text{(MndetPrefix)}$$

This rule states that if $c(a) \in B$, and the action $P(c(a))$ is refined by some action R, then the nondeterministic choice over all $a \in B$, prefixed by the event a and followed by $P(a)$, is refined by the deterministic communication $c!a \to R$.

Lemma 15 (write_proving_Mndetprefix_UNIV_ref)**.** *This is a specialisation of* MndetPrefix*, with* $B = \Sigma$*.*

Lemma 16 (read_proving_Mndetprefix_ref)**.**

Funded by
the European Union

$$\frac{\text{inj\_on } c\ A \qquad c\ `\ A \subseteq B \qquad A \neq \varnothing \qquad \Gamma \vdash \forall a \in A.\ P(c\ a) \sqsubseteq Q(a)}{\Gamma \vdash \sqcap_{a \in B}\ a \to P(a) \sqsubseteq c?a \in A \to Q(a)}\ \text{(Read-MndetPrefix-Ref)}$$

This rule supports refinement reasoning for input prefixes with value-dependent behaviour. It states that if c is injective over a nonempty set A, maps into a superset B, and for every $a \in A$, $Q(a)$ refines $P(c\ a)$, then $c?a \in A \to Q(a)$ refines the internal choice over B of $a \to P(a)$. This captures the idea that a deterministic input-driven action can refine a nondeterministic prefix structure, provided that each branch satisfies the required relation under the mapping.

Lemma 17 (GlobalNdet_refine_no_step)**.**

$$\frac{\Gamma \vdash a \in A}{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}\left(\sqcap_{x \in \Sigma} P(x)\right) \sqsubseteq P(a)}\ \text{(GlobalNdet-NoStep)}$$

This rule states that for any $a \in A$, any individual behaviour $P(a)$ refines the global iterative nondeterministic action $\mathrm{DFI}^{*}_{\Sigma}\left(\sqcap_{x \in \Sigma} P(x)\right)$. This is useful for decomposing refinement obligations involving nondeterministic iteration into simpler checks over individual branches.

Lemma 18 (interrupt_refine)**.**

$$\frac{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(X) \sqsubseteq P \qquad \Gamma \vdash Q = a \to Q}{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(X) \sqsubseteq Q \triangle P}\ \text{(Interrupt-Ref)}$$

This rule expresses that if, under context Γ, the iteration $\mathrm{DFI}^{*}_{\Sigma}(X)$ is refined by some P, and Q is defined as an infinite prefix loop $Q = a \to Q$, then the iteration $\mathrm{DFI}^{*}_{\Sigma}(X)$ is also refined by $Q \triangle P$. In addition to general-purpose structural laws, we also introduce two lemmas specialised for `SSTOP = share →SSTOP`, which occurs frequently in generated models. These lemmas are direct instantiations of Interrupt-Ref, introduced to improve automation.

Lemma 19 (SSTOP_refine_star)**.**

$$\frac{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(X) \sqsubseteq P}{\Gamma \vdash \mathrm{DFI}^{*}_{\Sigma}(X) \sqsubseteq \mathit{SSTOP} \triangle P}\ \text{(Interrupt-Ref-SSTOP-star)}$$

Lemma 20 (SSTOP_refine)**.**

$$\frac{\Gamma \vdash \mathrm{DFI}_{\Sigma}(X) \sqsubseteq P}{\Gamma \vdash \mathrm{DFI}_{\Sigma}(X) \sqsubseteq \mathit{SSTOP} \triangle P}\ \text{(Interrupt-Ref-SSTOP)}$$

Specifically, in these lemmas, the abstract infinite loop $Q = a \to Q$ is replaced by the concrete definition $\mathit{SSTOP} = \mathit{share} \to \mathit{SSTOP}$.

In our models, `SSTOP` frequently appears as the left-hand side of an interrupt expression. Rather than applying Interrupt-Ref manually and unfolding the definition each time, these lemmas allow this reasoning to be captured directly. This not only simplifies proof scripts but also improves automation by enabling the refinement tactic to match and discharge such interrupt structures without auxiliary steps.

Funded by
the European Union

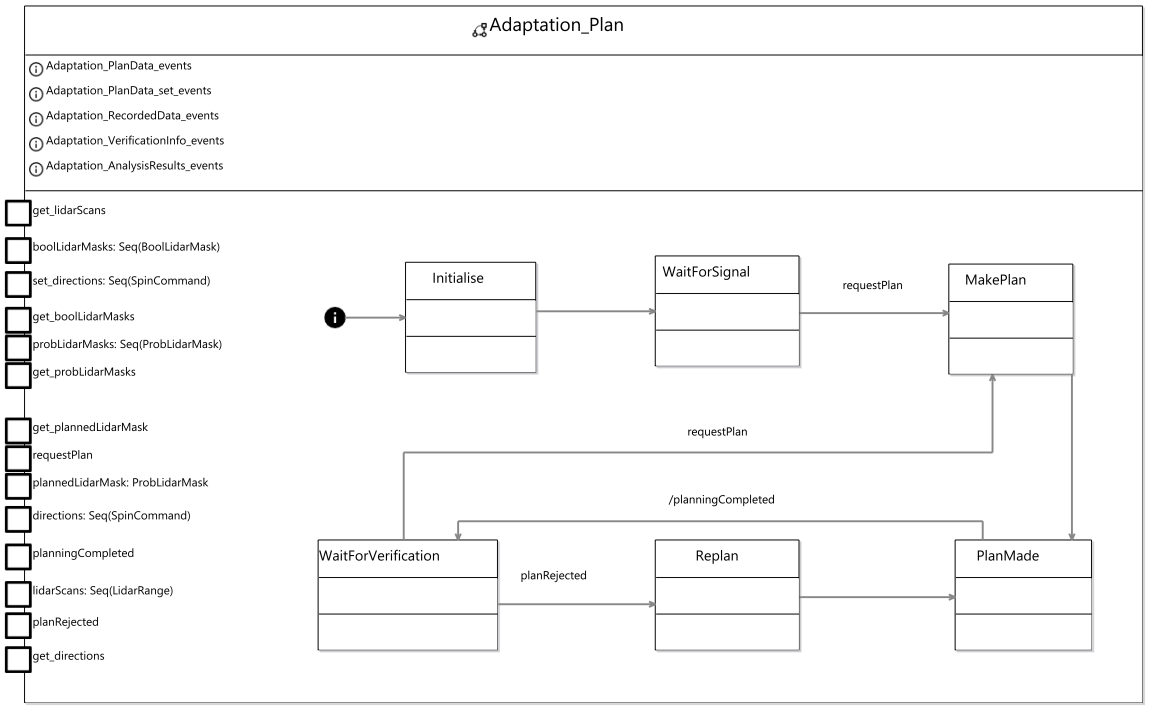


Figure 22: RoboChart model for the navigation robot MAPLE-K plan phase

We are now in a position to define a proof method named `deadlock_free` to automate proof of deadlock freedom. This is described next.

Automatic proof.

(1) Proof goal. To show that an entire action is deadlock-free, we prove that an action of the form

$$\sqcap n \in \Sigma.\ P(n)$$

is deadlock-free, that is, for every arbitrary n, $P(n)$ does not lead to a state where no events are possible. This result ensures that, regardless of which branch is selected, the action can always proceed.

(2) Proof method. A proof method is defined using the lemmas discussed above to realise the automatic proof of deadlock freedom. The proof method is composed of three steps. The first step applies an induction rule (Lemma 6.7); the second step simplifies the model using the normalisation rules, and the last proves deadlock freedom using the natural deduction rules. The proof method is defined according to the pattern of the transition behaviour of a RoboChart state machine.

Mechanisation uses Isabelle's classical reasoner to apply laws, and we may need a fourth step to apply Sledgehammer (a powerful native proof tactic native to Isabelle) to any residual proof goals (verification conditions).

(3) Proof example. We have applied our proof method to the transition behaviour of the RoboChart state machine for the MAPLE-K plan phase of the navigation

Funded by
the European Union

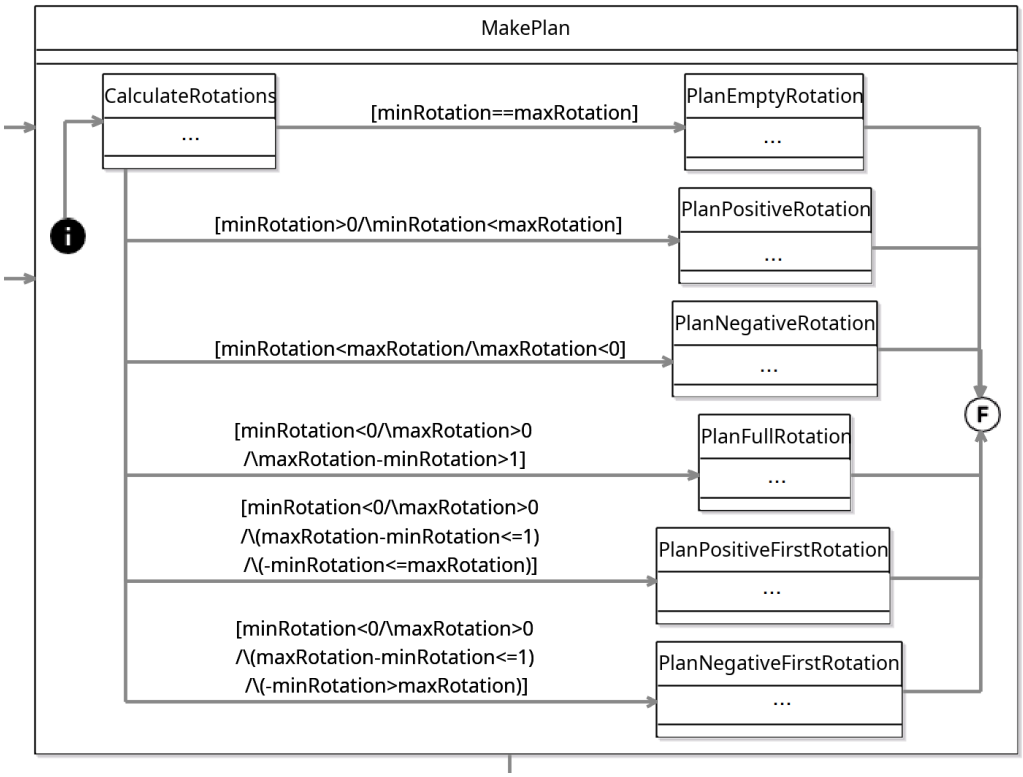


Figure 23: The RoboChart model of *MakePlan* state, enriched with details

robot. The state machine shown in Fig. 22 is automatically generated from the RoboArch model in Fig. 3. When enriching that machine to consider the application-specific features of the navigation case study, we add states and transitions within the *MakePlan* state, among others, to define how to construct plans. The enriched RoboChart definition for the state *MakePlan* is shown in Fig. 23.

In the *CalculateRotations* state, the software calculates the minimum and maximum rotations required to mitigate LiDAR occlusion (details omitted). This is followed by guarded transitions that enumerate the different rotation cases the robot should perform to optimally cover the required range.

The IsaCircus model `Trans_MakePlan_Adaptation_Plan` for the transition behaviour of the *MakePlan* state is automatically generated from the RoboChart model of *MakePlan* as shown in Appendix F. We first applied the proof method `deadlock_free` to `Trans_MakePlan_Adaptation_Plan`, and found that the goal cannot be discharged. Therefore, we have applied another proof method `find_counterexample` to find a counterexample showing the cause of the deadlock. The result is shown in Fig. 24.

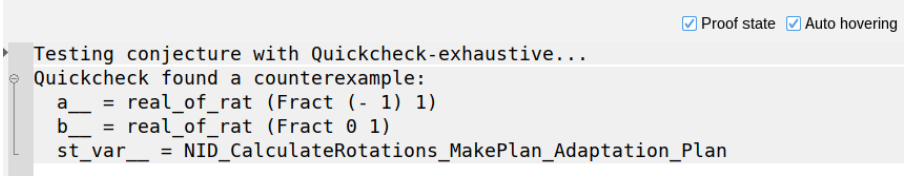


Figure 24: The counterexample causing the deadlock of *MakePlan* in Isabelle/HOL

The counterexample appears below the line "`Quickcheck found a counter example`".

Funded by
the European Union

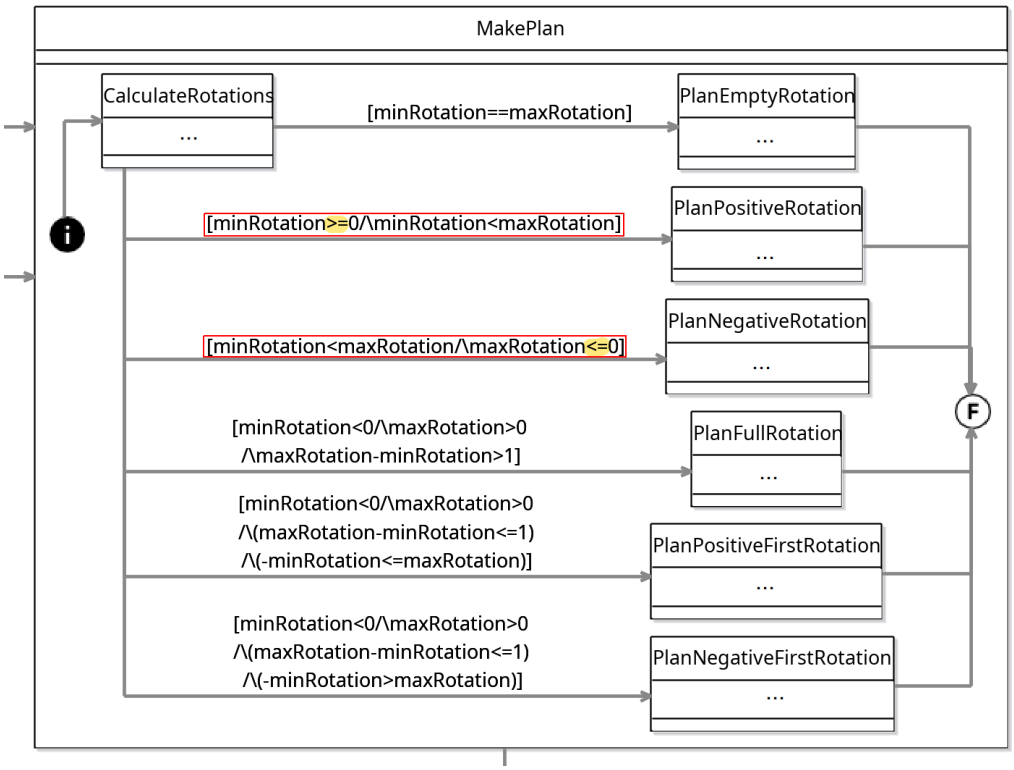


Figure 25: The RoboChart model of the state *MakePlan*, updated with correct transition conditions (highlighted)

In the line `a__ = real_of_rat (Fract (-1)1)`, the variable name `a__` is automatically generated by Isabelle as an internal alias for the model variable *minRotation*; moreover, `Fract (-1)1` represents the rational number `-1/1`, i.e., 1, and `real_of_rat` converts this rational number into a real number in Isabelle. Thus, the expression `real_of_rat (Fract (-1)1)` denotes the real number -1, and this line defines that *minRotation* = -1. Similarly, the line `b__ = real_of_rat (Fract 0 1)` defines that *maxRotation* = 0. In the last line, the variable `st_var` represents the current state of the system. Here, it indicates that the behaviour is in the *CalculateRotations* state of the *MakePlan* state of the *Adaptation_Plan* machine. Providing a description of this counterexample in terms of the elements of the RoboChart model, avoiding terms and variables internal to Isabelle, is a simple piece of future work.

Intuitively, the counterexample indicates that the case where *minRotation* is -1 and *maxRotation* is 0 is not covered by any of the transition conditions in *CalculateRotations*. This case should be covered by the transition to *PlanNegativeRotation*, which handles the case where the entire occlusion is on one side of the robot, requiring an anticlockwise turn. The transition guard considers that both rotations should be negative (indicating the anticlockwise direction) and not equal (as that is handled by another case) for this to be the case, but it does not account for the possibility that one rotation may be zero.

This can be easily corrected by requiring *maxRotation* to be less than or equal to zero, and we can also see that a similar error holds for the dual case in the transition to *PlanPositiveRotation*, corrected by requiring *minRotation* to be greater than or equal to zero. The updated RoboChart model is in Fig. 25.

Funded by
the European Union

We have applied the proof method `deadlock_free` to the updated model, and now the deadlock freedom of *MakePlan* is proved.

6.6 Final considerations

This chapter has presented a rich collection of tools that support the foundational work being carried out in RoboSAPIENS. With these tools, we can use those foundations to actually support the development work in the other workpackages.

We can define and reason about conceptual architectures that use a MAPLE-K pattern. We can consider RoboChart models for those patterns that involve AI components. Finally, we can prove properties of the models. All this will integrate well with efforts for legitimisation and trustworthiness checking. The complementary results in Deliverable D1.2 strengthen the connection.

Funded by
the European Union

7 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. Our formalisation covers several variations, including the traditional MAPE-K.

The verified RoboChart model can be mapped onto an AADL model (Deliverable 5.2), which is enriched to describe a particular platform for deployment. From AADL and RoboChart, an implementation can be developed via automatic code generation. Our work offers a complete pathway from architectural modelling through to code. By utilising our software platform to implement 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.

For modelling and reasoning about intelligent systems, our approach is supported by automated tools and a graphical modelling system. This enables and supports engineering based on the logic of the software, independent of the dynamics of the robotic system or other considerations of the details of the software, which makes our work distinct in the area of system-level ANN verification.

For uncertainty quantification, we have implemented a method called Monte-Carlo DropBlock, which is designed explicitly for CNN networks. We are applying it to the DTI case study. Work on the PAL case study is ongoing. We have also explored the use of Large Language Models (LLMs) to identify uncertainties, with experiments showing promising results and getting support from practitioners.

7.1 Future work in the scope of RoboSAPIENS

In future work, we will consider additional RoboSAPIENS case studies, extending the architectural patterns as required. Currently, we are working on the DTI case study: Deliverable 4.1. More patterns will lead to extensions of the meta-model, well-formedness conditions, and model-to-model transformation. A priority is defining patterns for the use of the trustworthiness checker.

The primary focus of future work, however, is the exploitation of the architecture for verification. This involves taking advantage of the traceability that our approach affords from code to conceptual architecture. This enables compositional reasoning about adaptive systems. The semantic model of RoboChart is based on CSP process algebra [Ros11], whose constructs are compositional concerning 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.

Regarding reasoning about ANN-enabled systems, we have three lines of work to pursue. The first is the complete formalisation and mechanisation of the verification approach presented here. The second is its application to RoboSAPIENS studies. The final one is the investigation of our approach to relearning.

Regarding the identification and quantification of uncertainty, we will conduct a large-scale industrial evaluation to collect insights related to uncertainty from a

Funded by
the European Union

broader range of use cases. Based on these findings and literature analysis, we will develop a comprehensive uncertainty taxonomy for self-adaptive robotics. Second, we will develop an LLM-based framework for automated identification and management of uncertainties. We will assess the usefulness and effectiveness of this framework for different types of self-adaptive robotic systems. We will also investigate the applicability of the UQ method and metrics to other case studies, such as the DL-based digital twins of ships and the DL-based anomaly detection in robotic systems use cases from RoboSAPIENS. Fifth, we will employ other UQ methods, such as Deep Ensembles [LPB17], and develop novel UQ metrics to quantify uncertainty in the environments of robots and DL models. Next, a holistic approach to uncertainty quantification will be developed, considering both environmental uncertainty and DL models. Finally, we will identify the factors contributing to uncertainty and then design methods to reduce overall uncertainty.

Regarding IsaCircus, we will apply the approach to several case studies, from RoboSAPIENS and others available in RoboStar, demonstrating its effectiveness in realistic modelling scenarios. In addition to the deadlock-freedom proofs, we will include the verification of further properties such as invariants and assertion satisfaction. We'll look into extending the deadlock freedom proof method to handle more components of RoboChart, e.g., state actions.

7.2 Future work beyond the scope of RoboSAPIENS

The MAPE-K community is very active (see Deliverable 3.1). We expect many other variations of MAPE-K will be proposed. We have concrete plans to extend it to support the development of robotic software that satisfies normative requirements. The work we present here provides a solid base to formalise other variations of MAPE-K and provide verification support for them.

Our work on verifying RoboChart models with ANN components provides a solid foundation for a rich agenda of future work. We envisage extending the definition of ANN components to support multiple types of ANN: convolutional, for image recognition, and transformer or recurrent networks, for time series data. Our semantics naturally allow the definition of multiple types of activation functions.

Further work on the link between process algebras and AI-system reasoning can enable a rich engineering approach for software that involves AI components. By employing process algebraic reasoning, we can derive properties regarding the level of imprecision and its propagation through the software in a high-level and rigorous manner, enabling tractable compositionality proofs.

Our theory of conformance has also highlighted potential areas for further research, such as improving methods for generating ϵ approximations and integrating probabilistic reasoning into conformance checking. Additionally, the application of hybrid systems theory offers a promising direction for verifying the behaviour of ANNs in more complex, real-world environments.

Regarding proof, future work will extend the proof method to support the proof of deadlock freedom with the physical platform in the loop. This will require hybrid theories. Equally interesting is the proof of deadlock freedom, taking into account the probabilistic and stochastic behaviour of RoboChart models.

Funded by
the European Union

Our work on uncertainty provides a general approach to identifying the uncertainty of robotics with LLMs. We foresee potential for LLMs to enhance decision-making and adaptive behaviour while handling uncertainty in robotics and other complex systems, particularly in complex and dynamic environments. Regarding uncertainty quantification, future work will explore novel methods for measuring and interpreting uncertainty in both deep learning models and traditional systems. This includes extending our framework to support multimodal inputs and enabling domain adaptation to enhance interpretability, robustness, and trustworthiness.

Funded by
the European Union

References

[ACW20] Z. Attala, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Comparison of Neural Network Tools for the Verification of Linear Specifications of ReLU Networks. In A. Albarghouthi, G. Katz, and N. Narodytska, editors, *3rd Workshop on Formal Methods for ML-Enabled Autonomous System*, pages 22–33, 2020.

[ACW23] Z. Attala, A. L.C. Cavalcanti, and J. C. P. Woodcock. Modelling and verifying robotic software that uses neural networks. In E. Ábrahám, C. Dubslaff, and S. L. T. Tarifa, editors, *Theoretical Aspects of Computing*, pages 15–35. Springer, 2023.

[ARS15] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 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*, pages 13–23, Florence, Italy, May 2015. IEEE.

[asm] The ASMETA toolset website.

[Aug24] Midhun T. Augustine. A survey on Universal Approximation Theorems. *CoRR*, abs/2407.12895, 2024.

[AYF+25] Ziggy Attala, Fang Yan, Simon Foster, Ana Cavalcanti, and Jim Woodcock. Process-Algebraic Semantics for Verifying Intelligent Robotic Control Software. In *Proceedings of The 17th NASA Formal Methods Symposium (NFM2025)*, 2025. to appear.

[BC19] James Baxter and Ana Cavalcanti. RoboStar demonstrator: a segway. Technical report, University of York, 2019. robostar.cs.york.ac.uk/publications/techreports/reports/segway.pdf.

[BCG+12] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and A. Vandin. A conceptual framework for adaptation. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 LNCS, pages 240–254, 2012.

[BCM22] W. Barnett, A. L. C. Cavalcanti, and A. Miyazawa. Architectural Modelling for Robotics: RoboArch and the CorteX example. *Frontiers of Robotics and AI*, 2022.

[BdGB+17] Tarek R. Besold, Artur S. d'Avila Garcez, Sebastian Bader, Howard Bowman, Pedro M. Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luís C. Lamb, Daniel Lowd, Priscila Machado Vieira Lima, Leo de Penning, Gadi Pinkas, Hoifung Poon, and Gerson Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. *CoRR*, abs/1711.03902, 2017.

[BFG+97] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

Funded by
the European Union

[BFW09] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009.

[BGJ+23] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, et al. Improving image generation with better captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, 2(3):8, 2023.

[BKPU16] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1), 2016.

[BS23] Achim D. Brucker and Amy Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods*, pages 427–444, Cham, 2023. Springer International Publishing.

[BSML18] Maryam Bagheri, Marjan Sirjani, Ali Movaghar, and Edward A Lee. Coordinated actor model of self-adaptive track-based traffic control systems. *The Journal of Systems & Software*, 143(September 2017):116–139, 2018.

[BvAK+25] James Baxter, Bert van Acker, Morten Kristensen, Thomas Wright, Ana Cavalcanti, and Cláudio Gomes. Formal architectural patterns for adaptive robotic software. In Artur Boronat and Gordon Fraser, editors, *Fundamental Approaches to Software Engineering*, pages 145–165, Cham, 2025. Springer Nature Switzerland.

[CBB+21] A. L. C. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, and A. C. A. Sampaio. *RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing*, pages 249–293. Springer International Publishing, 2021.

[CBC18] M. Camilli, C. Bellettini, and L. Capra. A high-level petri net-based formal model of Distributed Self-adaptive Systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, 2018.

[CfC20] European Commission and Directorate-General for Communication. *Circular economy action plan – For a cleaner and more competitive Europe*. Publications Office of the European Union, 2020.

[CMS13] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer, 2013.

[CSW03] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.

[CSW04] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement: An overview. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering,*

Funded by
the European Union

First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004, Revised Lectures, volume 3167 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2004.

[CSW05] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Softw. Syst. Model.*, 4(3):277–296, 2005.

[CW17] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017.

[CYA21] Ferhat Ozgur Catak, Tao Yue, and Shaukat Ali. Prediction surface uncertainty quantification in object detection models for autonomous driving. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 93–100. IEEE, 2021.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control. Signals Syst.*, 2(4):303–314, 1989.

[DAK+23] Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 102–120, New York, NY, USA, 2023. Association for Computing Machinery.

[DKA+24] Matthew L. Daggitt, Wen Kokke, Robert Atkey, Natalia Slusarz, Luca Arnaboldi, and Ekaterina Komendantskaya. Vehicle: Bridging the Embedding Gap in the Verification of Neuro-Symbolic Programs, 2024.

[DSM20] Rangan Das, Sagnik Sen, and Ujjwal Maulik. A survey on fuzzy deep neural networks. *ACM Comput. Surv.*, 53(3), may 2020.

[FCC+20] S. Foster, A. L. C. Cavalcanti, S. Canham, J. C. P. Woodcock, and F. Zeyda. Unifying theories of reactive design contracts. *Theoretical Computer Science*, 802:105 – 140, 2020.

[FZN+19] S. Foster, F. Zeyda, Y. Nemouchi, P. Ribeiro, and B. Wolff. Isabelle/UTP: Mechanised theory engineering for unifying theories of programming. *Archive of Formal Proofs*, 2019, 2019.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org, 2011.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

Funded by
the European Union

[GG16] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[GL23] Artur d'Avila Garcez and Luís C. Lamb. Neurosymbolic AI: The 3rd wave. *Artif. Intell. Rev.*, 56(11):12387–12406, mar 2023.

[GRABR14] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.

[GTCM20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of field robotics*, 37(3):362–386, 2020.

[Hai16] Tameru Hailesilassie. Rule extraction algorithm for deep neural networks: A review. *International Journal of Computer Science and Information Security (IJCSIS)*, 14:376–381, July 2016.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Heh84a] Eric C. R. Hehner. Predicative programming, Part I. *Commun. ACM*, 27(2):134–143, 1984.

[Heh84b] Eric C. R. Hehner. Predicative programming, Part II. *Commun. ACM*, 27(2):144–151, 1984.

[HJ98a] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HJ98b] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall, 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[HOL+24] Ahmed E. Hassan, Gustavo A. Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, and Jiang. Rethinking Software Engineering in the Foundation Model Era: From Task-Driven AI Copilots to Goal-Driven AI Pair Programmers, 2024.

[HW21] Eyke Hüllermeier and Willem Waegeman. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine learning*, 110(3):457–506, 2021.

[IBZK22] Omri Isac, Clark W. Barrett, M. Zhang, and Guy Katz. Neural Network Verification with Proof Production. *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pages 38–48, 2022.

[IJH+21] Radoslav Ivanov, Kishor Jothimurugan, Steve Hsu, Shaan Vaidya, Rajeev Alur, and Osbert Bastani. Compositional Learning and Verifica-

Funded by
the European Union

tion of Neural Network Controllers. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[K+19] Guy Katz et al. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 443–452. Springer, 2019.

[LB06] Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, pages 154–169, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[LBOM98] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[LCTJ23] Diego Manzanas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T. Johnson. Nnv 2.0: The neural network verification tool. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 397–412, Cham, 2023. Springer Nature Switzerland.

[LFA+24] Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I*, page 311–330, Berlin, Heidelberg, 2024. Springer-Verlag.

[LGM+17] Holden Lee, Rong Ge, Tengyu Ma, Andrej Risteski, and Sanjeev Arora. On the ability of neural nets to express distributions. In Satyen Kale and Ohad Shamir, editors, *Proceedings of the 30th Conference on Learning Theory, COLT 2017, Amsterdam, The Netherlands, 7-10 July 2017*, volume 65 of *Proceedings of Machine Learning Research*, pages 1271–1296. PMLR, 2017.

[LPB17] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems*, 30, 2017.

[LSV03] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[Mor94] Carroll Morgan. *Programming from specifications*. Prentice Hall International series in computer science. Prentice Hall, 2nd edition, 1994.

Funded by
the European Union

[MRL+19] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019.

[MRY+21] Alvaro Miyazawa, Pedro Ribeiro, Kangfeng Ye, Ana Cavalcanti, Wei Li, Jim Woodcock, and Jon Timmis. RoboChart Reference Manual, 2021.

[MS89] Carroll Morgan and J. W. Sanders. Laws of the logical calculi. Technical Report PRG-78, Programming Research Group, Oxford University Computing Laboratory, September 1989.

[MSF20] J. H. Y. Munive, G. Struth, and S. Foster. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In *18th International Conference on Relational and Algebraic Methods in Computer Science*, volume 12062 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2020.

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[Oli06] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs Using Circus*. PhD thesis, University of York, 2006.

[PDPB14] J.M.T. Portocarrero, F.C. Delicato, P.F. Pires, and T.V. Batista. Reference architecture for self-adaptive management in wireless sensor networks. In Abdelhamid Bouchachia, editor, *Adaptive and Intelligent Systems*, volume 8779 LNAI, pages 110–120, 2014.

[Pla10] A. Platzer. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer, 2010.

[PP24] Habeeb P and Pavithra Prabhakar. Approximate conformance verification of deep neural networks. In Nathaniel Benz, Divya Gopinath, and Nija Shi, editors, *NASA Formal Methods - 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Proceedings*, volume 14627 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2024.

[PW08a] Juan Ignacio Perna and Jim Woodcock. Mechanised wire-wise verification of Handel-C synthesis. In Patrícia D. L. Machado, editor, *Proceedings of the Eleventh Brazilian Symposium on Formal Methods, SBMF 2008, Salvador, Brazil, August 26-29, 2008*, volume 240 of *Electronic Notes in Theoretical Computer Science*, pages 201–219. Elsevier, 2008.

[PW08b] Juan Ignacio Perna and Jim Woodcock. UTP semantics for Handel-C. In Andrew Butterfield, editor, *Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, volume 5713 of *Lecture Notes in Computer Science*, pages 142–160. Springer, 2008.

Funded by
the European Union

[PWSI11] Juan Ignacio Perna, Jim Woodcock, Augusto Sampaio, and Juliano Iyoda. Correct hardware synthesis — An algebraic approach. *Acta Informatica*, 48(7-8):363–396, 2011.

[QT19] Meng Qu and Jian Tang. Probabilistic logic neural networks for reasoning. *CoRR*, abs/1906.08495, 2019.

[RBL+21] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2021.

[RJC12] Andres J Ramirez, Adam C Jensen, and Betty HC Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 99–108. IEEE, 2012.

[RNS+05] Steve Reeves, David A. Naumann, Margaret-Anne Storey, Jens Bendisposto, Christian Engelhardt, and Frank Hölzl. Czt: A framework for z tools. In *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *LNCS*, pages 65–84. Springer, 2005.

[Ros11] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[SdG16] Luciano Serafini and Artur S. d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *CoRR*, abs/1606.04422, 2016.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SJ22] Serena Serafina Serbinowska and Taylor T. Johnson. Behaverify: Verifying temporal logic specifications for behavior trees. In Bernd-Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods*, pages 307–323, Cham, 2022. Springer International Publishing.

[SKS19] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 147–156, New York, NY, USA, 2019. Association for Computing Machinery.

[SSS22] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Toward verified artificial intelligence. *Commun. ACM*, 65(7):46–55, jun 2022.

[TDVDWH19] Dustin Tran, Mike Dusenberry, Mark Van Der Wilk, and Danijar Hafner. Bayesian layers: A module for neural network uncertainty. *Advances in neural information processing systems*, 32, 2019.

[TWY20] Safouan Taha, Burkhart Wolff, and Lina Ye. The hol-csp refinement toolkit. *Arch. Formal Proofs*, 2020, 2020.

Funded by
the European Union

[VDDP18] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018(1):7068349, 2018.

[WC22] M. Windsor and A. L. C. Cavalcanti. RoboCert: Property Specification in Robotics. In A. Riesco and M. Zhang, editors, *International Conference on Formal Engineering Methods*, volume 13478 of *Lecture Notes in Computer Science*. Springer, 2022.

[WCF+19] Jim Woodcock, Ana Cavalcanti, Simon Foster, Alexandre Mota, and Kangfeng Ye. Probabilistic semantics for RoboChart — A weakest completion approach. In Pedro Ribeiro and Augusto Sampaio, editors, *Unifying Theories of Programming - 7th International Symposium, UTP 2019, Dedicated to Tony Hoare on the Occasion of His 85th Birthday, Porto, Portugal, October 8, 2019, Proceedings*, volume 11885 of *Lecture Notes in Computer Science*, pages 80–105. Springer, 2019.

[WCF+23] J. C. P. Woodcock, A. L. C. Cavalcanti, S. Foster, M. V. M. Oliveira, A. C. A. Sampaio, and F. Zeyda. UTP, *Circus*, and Isabelle. In J. Bowen, Q. Li, and Q. Xu, editors, *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*, pages 19–51. Springer, 2023.

[WCM+23a] Danny Weyns, Radu Calinescu, Raffaela Mirandola, Kenji Tei, Maribel Acosta, Nelly Bencomo, Amel Bennaceur, Nicolas Boltz, Tomas Bures, Javier Camara, et al. Towards a research agenda for understanding and managing uncertainty in self-adaptive systems. *ACM SIGSOFT Software Engineering Notes*, 48(4):20–36, 2023.

[WCM+23b] Danny Weyns, Radu Calinescu, Raffaela Mirandola, Kenji Tei, Maribel Acosta, Nelly Bencomo, Amel Bennaceur, Nicolas Boltz, Tomas Bures, Javier Camara, et al. Towards a research agenda for understanding and managing uncertainty in self-adaptive systems. *ACM SIGSOFT Software Engineering Notes*, 48(4):20–36, 2023.

[WD96a] J. C. P. Woodcock and J. Davies. *Using Z - Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[WD96b] Jim Woodcock and Jim Davies. *Using Z — Specification, refinement, and proof*. Prentice Hall international series in computer science. Prentice Hall, 1996.

[WDWF24] Thomas Wright, Louise A. Dennis, Jim Woodcock, and Simon Foster. Formal verification of BDI agents. To be published at isola 2024., Aarhus University, Denmark, 2024.

[Woo91a] Jim Woodcock. The Refinement Calculus. In Søren Prehn and W. J. Toetenel, editors, *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2: Tutorials*, volume

Funded by
the European Union

552 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 1991.

[Woo91b] Jim Woodcock. A tutorial on the Refinement Calculus. In Søren Prehn and W. J. Toetenel, editors, *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*, pages 79–140. Springer, 1991.

[Woo23] Jim Woodcock. Towards a unifying framework for uncertainty in cyber-physical systems. In Anne E. Haxthausen, Wen-ling Huang, and Markus Roggenbach, editors, *Applicable Formal Methods for Safe Industrial Products - Essays Dedicated to Jan Peleska on the Occasion of His 65th Birthday*, volume 14165 of *Lecture Notes in Computer Science*, pages 237–253. Springer, 2023.

[XWCL15] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[XWZ+17] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Yuyin Zhou, Lingxi Xie, and Alan Yuille. Adversarial examples for semantic segmentation and object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 1369–1378, 2017.

[YCF+22a] K. Ye, A. L. C. Cavalcanti, S. Foster, A. Miyazawa, and J. C. .P. Woodcock. Probabilistic modelling and verification using RoboChart and PRISM. *Software and Systems Modeling*, 21:667–716, 2022.

[YCF+22b] Kangfeng Ye, Ana Cavalcanti, Simon Foster, Alvaro Miyazawa, and Jim Woodcock. Probabilistic modelling and verification using RoboChart and PRISM. *Softw. Syst. Model.*, 21(2):667–716, 2022.

[YFW21] Kangfeng Ye, Simon Foster, and Jim Woodcock. Automated reasoning for probabilistic sequential programs with theorem proving. In Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter, editors, *Relational and Algebraic Methods in Computer Science - 19th International Conference, RAMiCS 2021, Marseille, France, November 2-5, 2021, Proceedings*, volume 13027 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2021.

[YLL+23] Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. The Dawn of LMMs: Preliminary Explorations with GPT-4V(ision), 2023.

[YW24] Kangfeng Ye and Jim Woodcock. RoboCertProb: Property specification for probabilistic RoboChart models. *CoRR*, abs/2403.08136, 2024.

[YWF23] Kangfeng Ye, Jim Woodcock, and Simon Foster. Probabilistic relations for modelling epistemic and aleatoric uncertainty: its semantics and automated reasoning with theorem proving. *CoRR*, abs/2303.09692, 2023.

Funded by
the European Union

[ZSCS14] Frank Zeyda, Thiago L. V. L. Santos, Ana Cavalcanti, and Augusto Sampaio. A modular theory of object orientation in higher-order UTP. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 627–642. Springer, 2014.

[ZWL+24] Dapeng Zhi, Peixin Wang, Si Liu, C.-H. Luke Ong, and Min Zhang. Unifying qualitative and quantitative safety verification of DNN-controlled systems. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 401–426, Cham, 2024. Springer Nature Switzerland.

Funded by
the European Union

Table 1: Selected *Circus* action operators. Here, we use A and B as metavariables to represent actions, cs to denote a set of channels, e to represent an event, i to represent an index, and T to represent a finite type. For the replicated (iterated) operators, $A(i)$ is an action identified i.

Symbol	**Name**	**Symbol**	**Name**
Skip	Skip	$e \longrightarrow A$	Prefix
$A \llbracket \mid cs \mid \rrbracket B$	Parallel Composition	$\llbracket cs \rrbracket i : T \bullet A(i)$	Replicated Parallel
$A \mathbin{;} Q$	Sequential Composition	$\fatsemi i : T \bullet A(i)$	Replicated Sequential Composition
$c?x \longrightarrow A$	Input	$c!e \longrightarrow A$	Output
$(c) \mathbin{\&} A$	Guarded Action	$A \setminus cs$	Hiding
$A[e := e1]$	Renaming	$A \bigtriangleup B$	Interrupt
$A \mid\mid\mid B$	Interleaving		

Funded by
the European Union

A Translation Rules from RoboArch to RoboChart for the MAPLEK pattern

Rule 1. Interfaces for a `MAPLEK` Layer (Rule 1 in Section 2.4.2)

MAPLEKLayerToInterfaces(ammkl : Layer) : Set(Interface) =

{

ammkl.name_Inputs	ammkl.name_Outputs
for ev **in** inputs ⚡ ev.name: ev.type **end for**	**for** ev **in** outputs ⚡ ev.name: ev.type **end for**

ammkl.name_InputVars	ammkl.name_InputFlags
for ev **in** inputs ✕ ev.name_data: ev.type **end for**	**for** ev **in** inputs ✕ ev.name_flag: boolean **end for**

}

∪ RecordedDataInterfaces(ammkl.pattern.monitor.recordedData, ammkl.pattern)

∪ ⋃{analyse : ammkl.pattern.analyse •
AnalysisResultsInterfaces(analyse.analysisResults, ammkl.pattern, ammkl.name)}

∪ ⋃{plan : ammkl.pattern.plan •
PlanDataInterfaces(plan.planData, ammkl.pattern, ammkl.name)}

∪ ⋃{legitimate : ammkl.pattern.legitimate •
VerificationInfoInterfaces(legitimate.verificationInfo, ammkl.pattern)}

where

inputs = ammkl.pattern.monitor.inputs

outputs = ammkl.pattern.execute.outputs

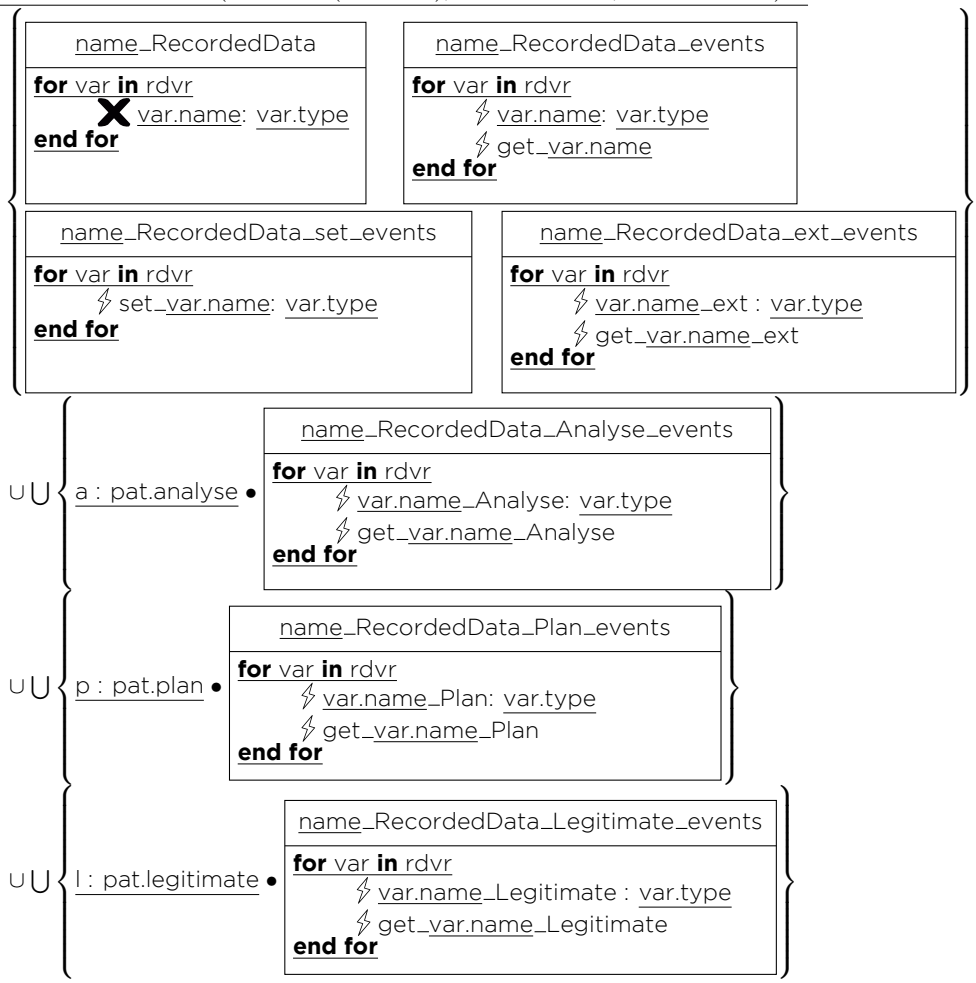
provided

ammkl.pattern **instanceof** MAPLEK

Funded by
the European Union

Rule 2. Interfaces for Recorded Data Variables (Rule 2 in Section 2.4.2)

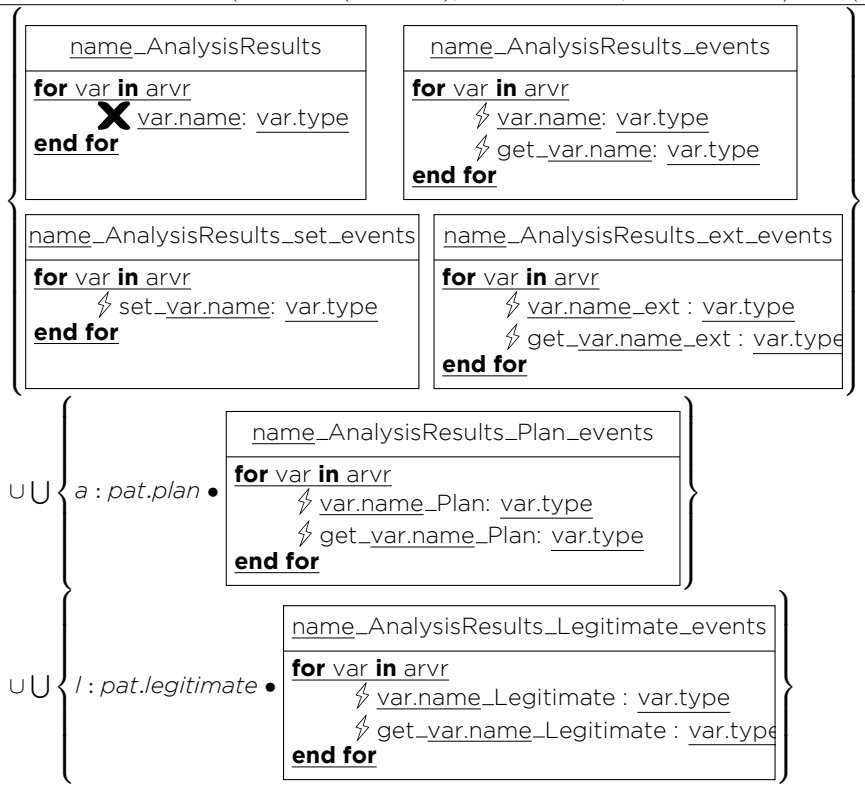
RecordedDataInterfaces(rdvr : Set(Variable), pat : MAPLEK, name : String) =



Funded by
the European Union

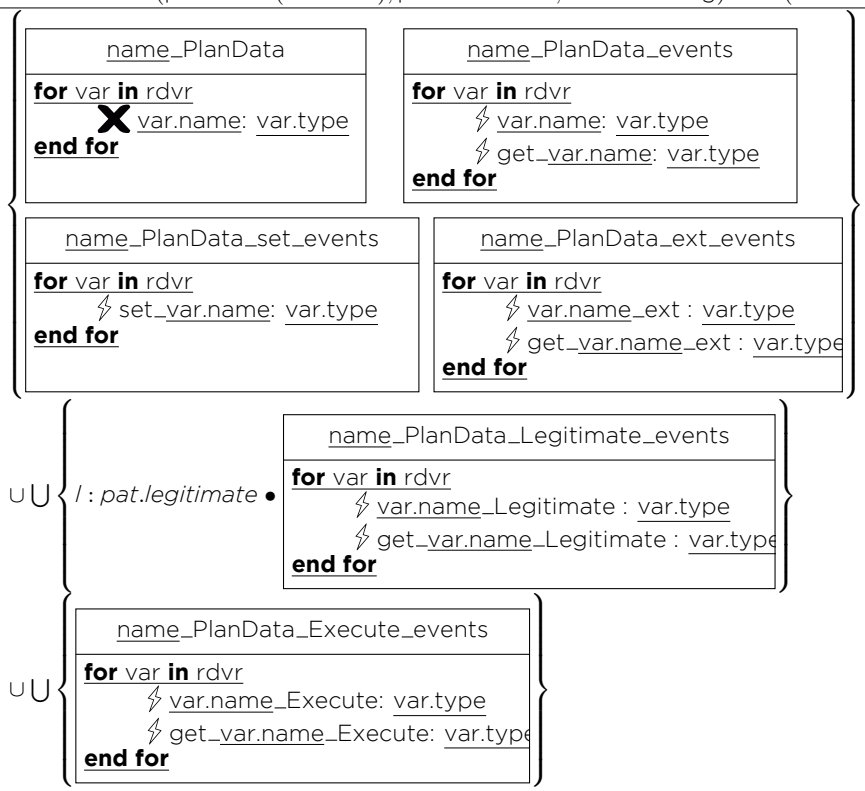
Rule 3. Interfaces for Analysis Results Variables

AnalysisResultsInterfaces(arvr : Set(Variable), pat : MAPLEK, name : String) : Set(Interface) =



Rule 4. Interfaces for Plan Data Variables

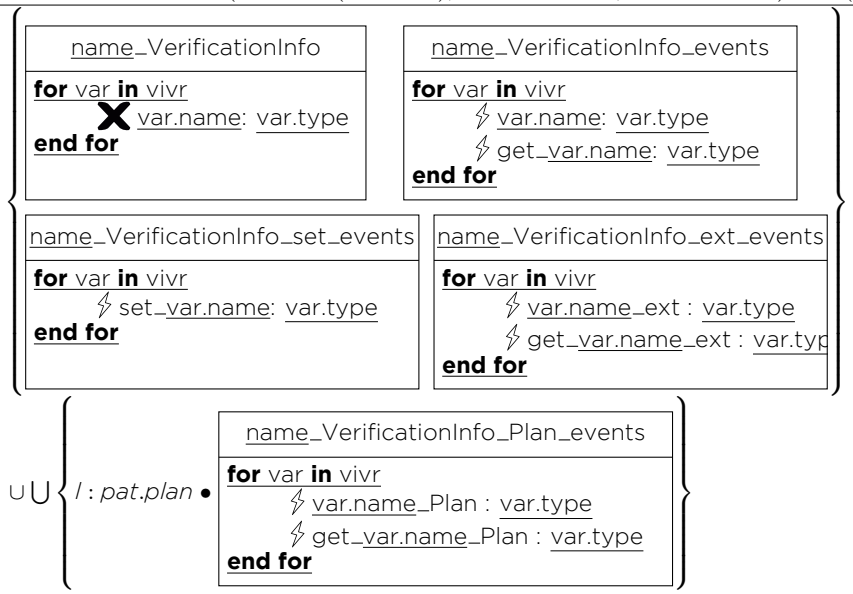
PlanDataInterfaces(pdvr : Set(Variable), pat : MAPLEK, name : String) : Set(Interface) =



Funded by
the European Union

Rule 5. Interfaces for Verification Info Variables

VerificationInfoInterfaces(vivr : Set(Variable), pat : MAPLEK, name : String) : Set(Interface) =



Rule 6. Machines and Connections for a `MAPLEK` layer (Rule 3 in Section 2.4.2)

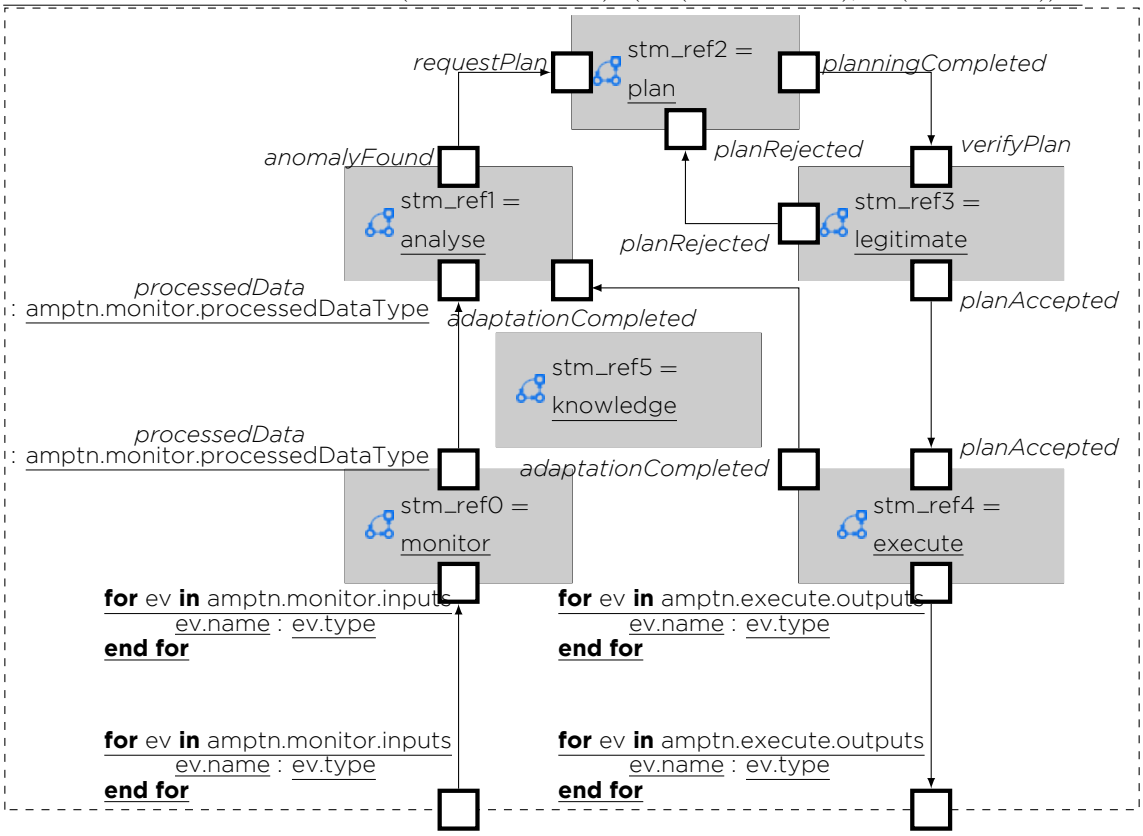
MAPLEKPatternToMachinesAndConnections(ammkl : Layer) : (Set(StateMachine), Set(Connection)) =

if # ammkl.pattern.legitimate = 0 **then**
  if # ammkl.pattern.plan = 0 **then**
    if # ammkl.pattern.analyse = 0 **then** MEKMachinesAndConnections(ammkl)
    else MAEKMachinesAndConnections(ammkl)
    end if
  else
    if # ammkl.pattern.analyse = 0 **then** MEPKMachinesAndConnections(ammkl)
    else MAPEKMachinesAndConnections(ammkl)
    end if
  end if
else
  if # ammkl.pattern.analyse = 0 **then** MPLEKMachinesAndConnections(ammkl)
  else MAPLEKMachinesAndConnections(ammkl)
  end if
end if
provided
  ammkl.pattern **instanceof** MAPLEK

Funded by
the European Union

Rule 7. Machines and Connections for a `MAPLEK` layer with all components (Rule 4 in Section 2.4.2)

MAPLEKMachinesAndConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =



∪ MAPLEKKnowledgeExternalConnections(amptn)
∪ MAPLEKMonitorKnowledgeConnections(amptn)
∪ MAPLEKAnalyseKnowledgeConnections(amptn)
∪ MAPLEKPlanKnowledgeConnections(amptn)
∪ MAPLEKLegitimateKnowledgeConnections(amptn)
∪ MAPLEKExecuteKnowledgeConnections(amptn)

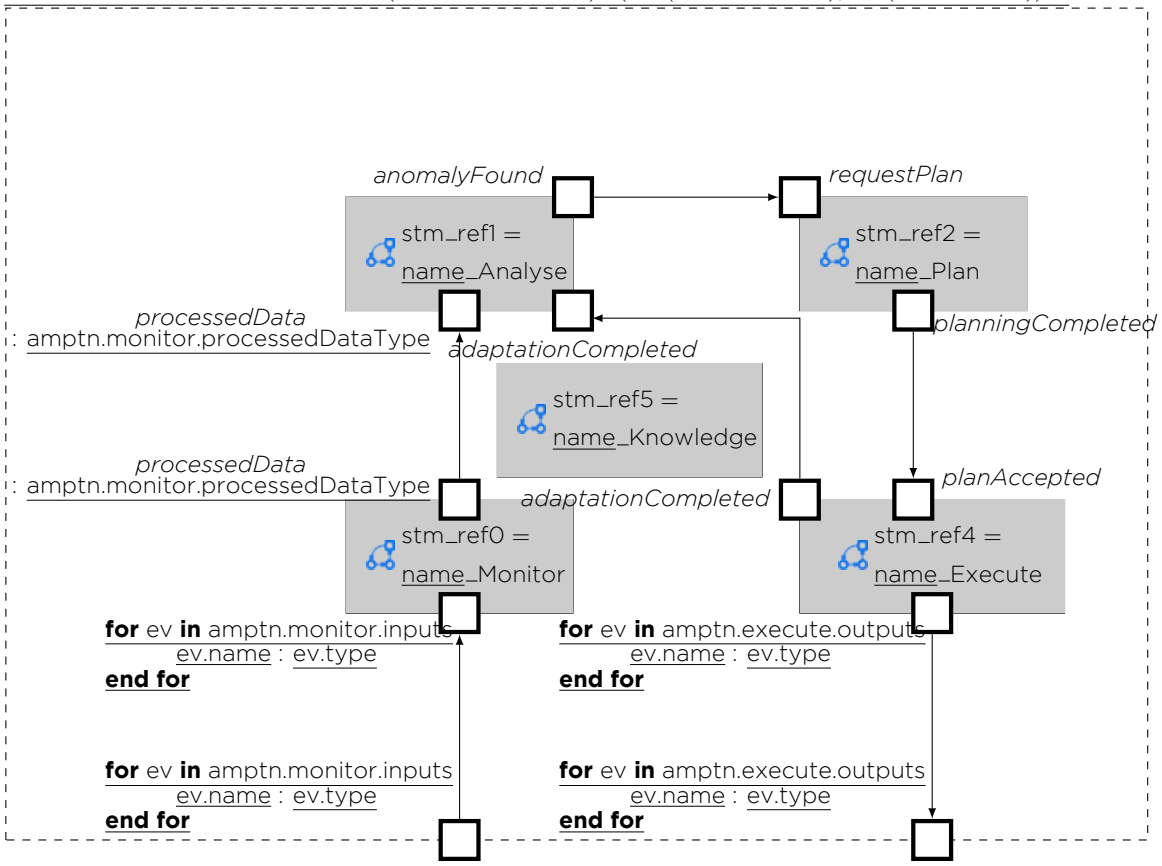
where

monitor = MonitorStateMachine(amlyr.pattern.monitor, amlyr.pattern, amlyr.name)
analyse = AnalyseStateMachine(amlyr.pattern.analyse, amlyr.pattern, amlyr.name)
plan = PlanStateMachine(amlyr.pattern.plan, amlyr.pattern, amlyr.name)
legitimate = LegitimateStateMachine(amlyr.pattern.legitimate, amlyr.pattern, amlyr.name)
execute = ExecuteStateMachine(amlyr.pattern.execute, amlyr.pattern, amlyr.name)
knowledge = KnowledgeStateMachine(amlyr.pattern, amlyr.name)

Funded by the European Union

Rule 8. Machines and Connections for a `MAPLEK` layer with no `LegitimateComponent`

MAPEKMachinesAndConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =

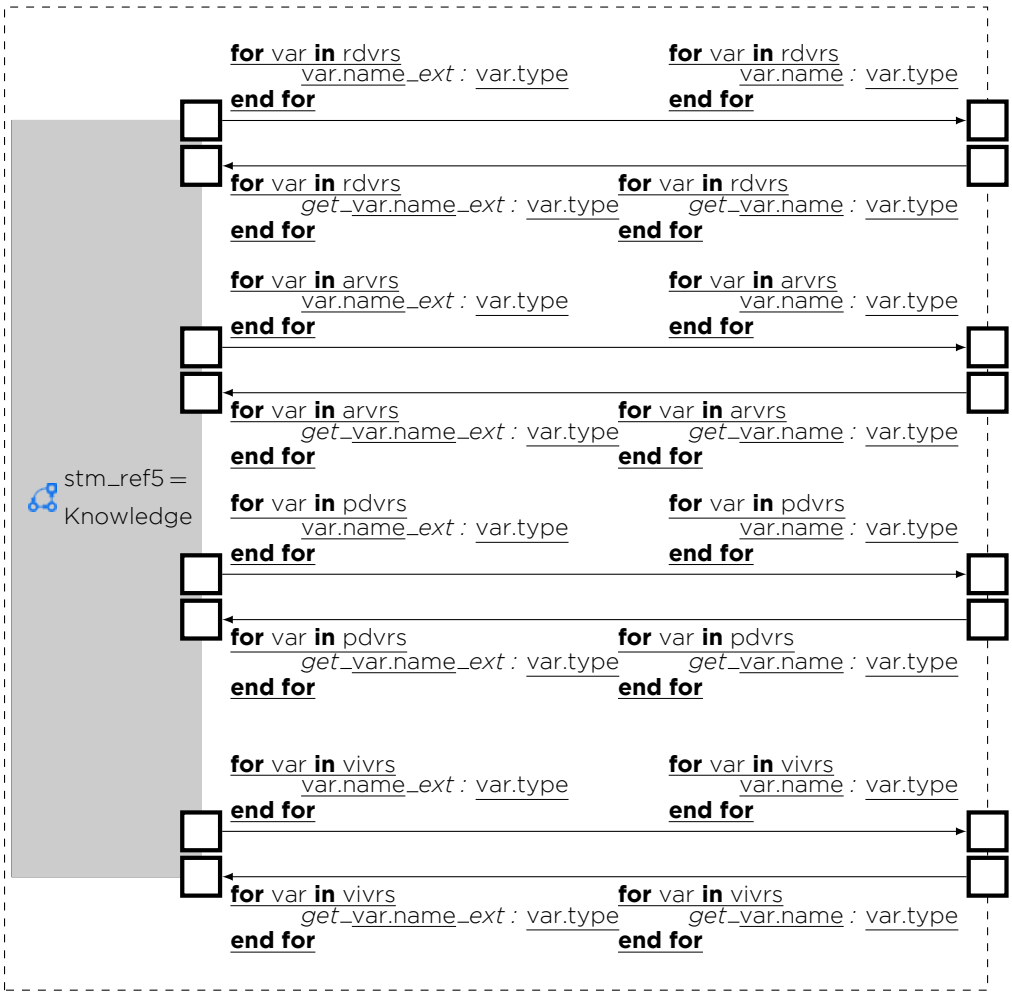


∪ MAPLEKKnowledgeExternalConnections(amptn)
∪ MAPLEKMonitorKnowledgeConnections(amptn)
∪ MAPLEKAnalyseKnowledgeConnections(amptn)
∪ MAPLEKPlanKnowledgeConnections(amptn)
∪ MAPLEKExecuteKnowledgeConnections(amptn)

Funded by
the European Union

Rule 9. Connections from the Knowledge State Machine to the Controller

MAPLEKKnowledgeExternalConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =



where

rdvrs = amptn.monitor.recordedData

arvrs = ⋃{a : amptn.analyse • a.analysisResults}

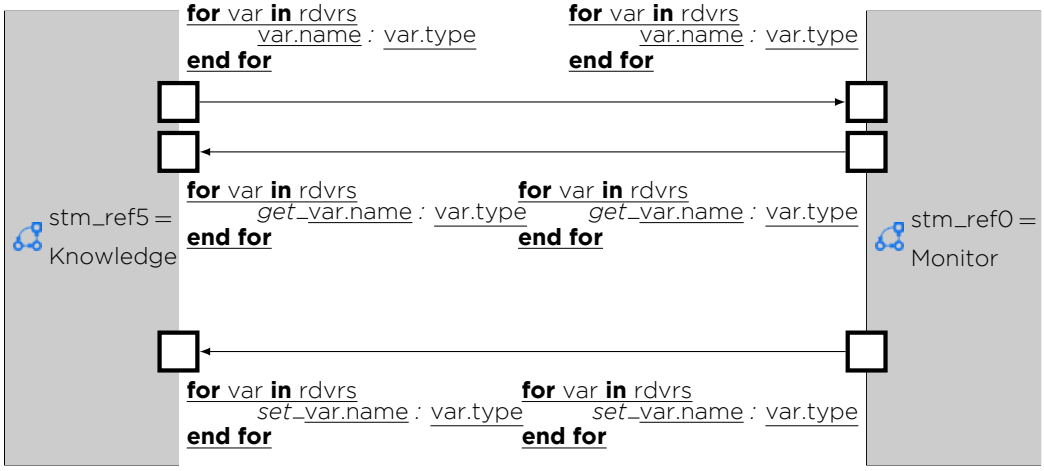
pdvrs = ⋃{p : amptn.plan • p.planData}

vivrs = ⋃{l : amptn.legitimate • l.verificationInfo}

Funded by the European Union

Rule 10. Connections between the Knowledge and Monitor State Machines

MAPLEKMonitorKnowledgeConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =



where

rdvrs = amptn.monitor.recordedData

Rule 11. Connections between the Knowledge and Analyse State Machines

MAPLEKAnalyseKnowledgeConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =

where

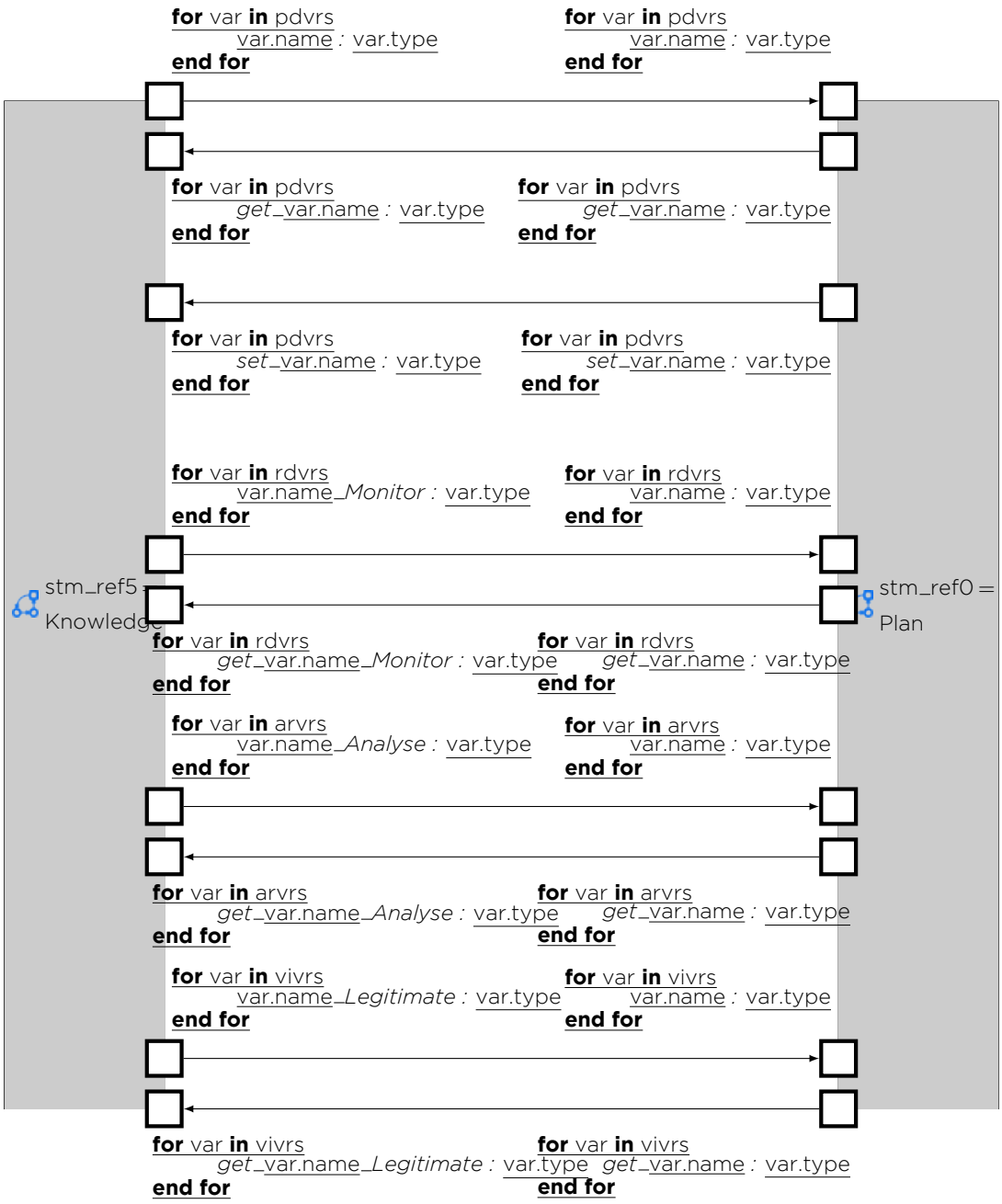
rdvrs = amptn.monitor.recordedData

arvrs = ⋃{a : amptn.analyse • a.analysisResults}

Funded by
the European Union

Rule 12. Connections between the Knowledge and Plan State Machines

MAPLEKPlanKnowledgeConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =



where

rdvrs = amptn.monitor.recordedData

arvrs = ⋃{a : amptn.analyse • a.analysisResults}

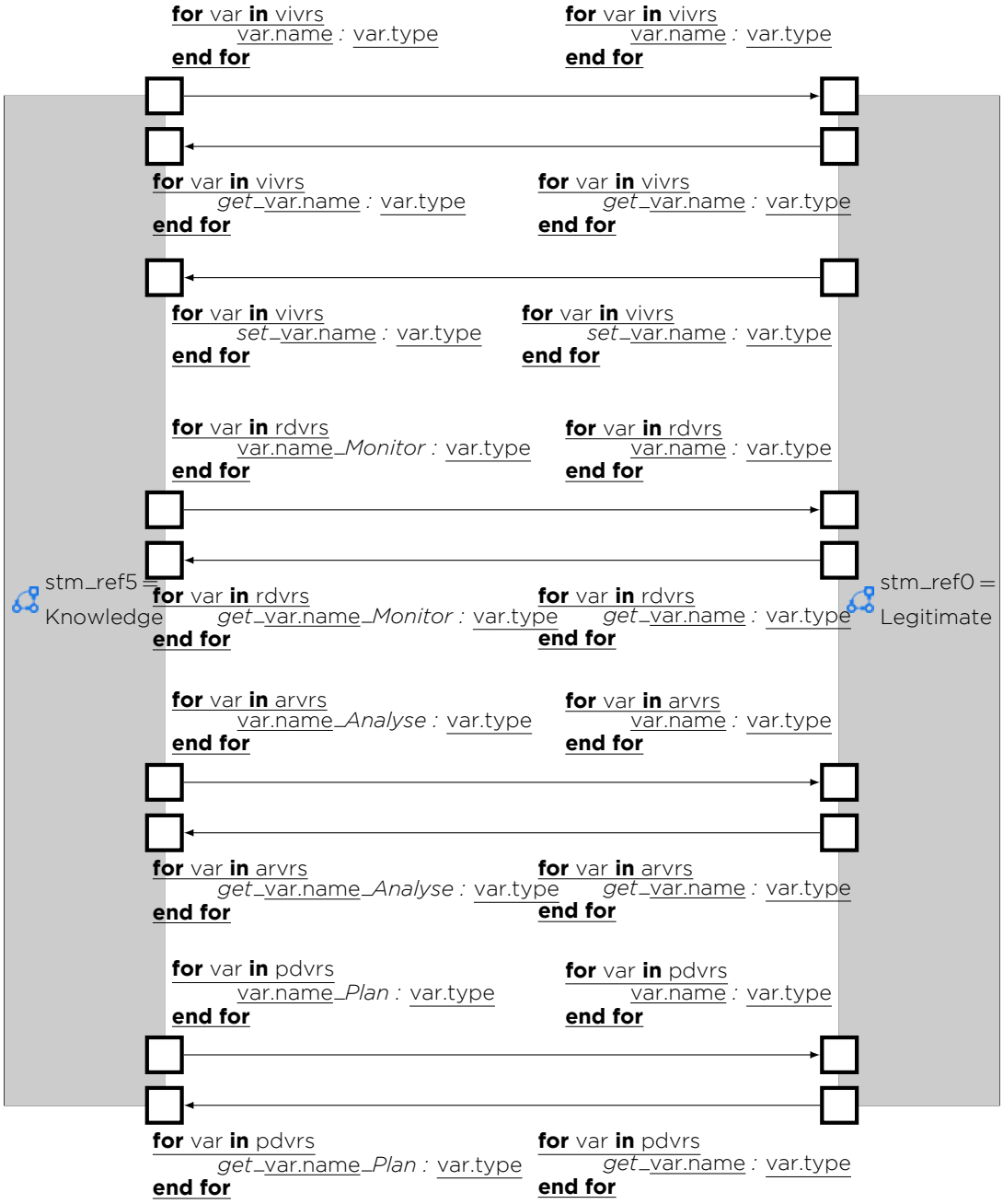
pdvrs = ⋃{p : amptn.plan • p.planData}

vivrs = ⋃{l : amptn.legitimate • l.verificationInfo}

Funded by
the European Union

Rule 13. Connections between the Knowledge and Legitimate State Machines

MAPLEKLegitimateKnowledgeConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =



where

rdvrs = amptn.monitor.recordedData

arvrs = ⋃{a : amptn.analyse • a.analysisResults}

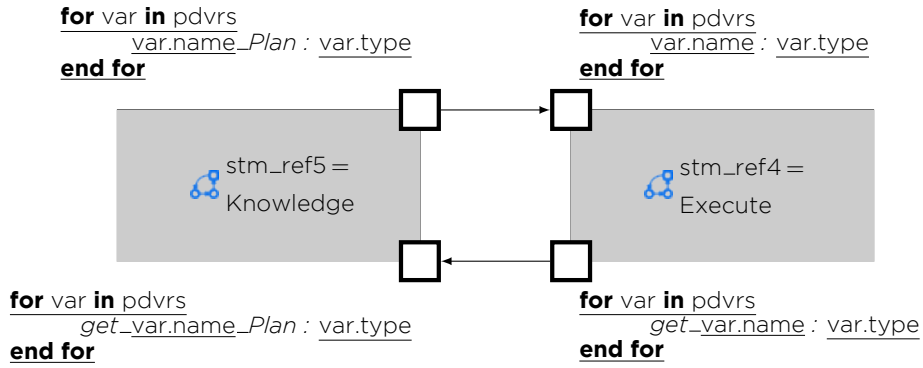
pdvrs = ⋃{p : amptn.plan • p.planData}

vivrs = ⋃{l : amptn.legitimate • l.verificationInfo}

Funded by the European Union

Rule 14. Connections between the Knowledge and Execute State Machines

MAPLEKExecuteKnowledgeConnections(amptn : MAPLEK) : (Set(StateMachine), Set(Connection)) =

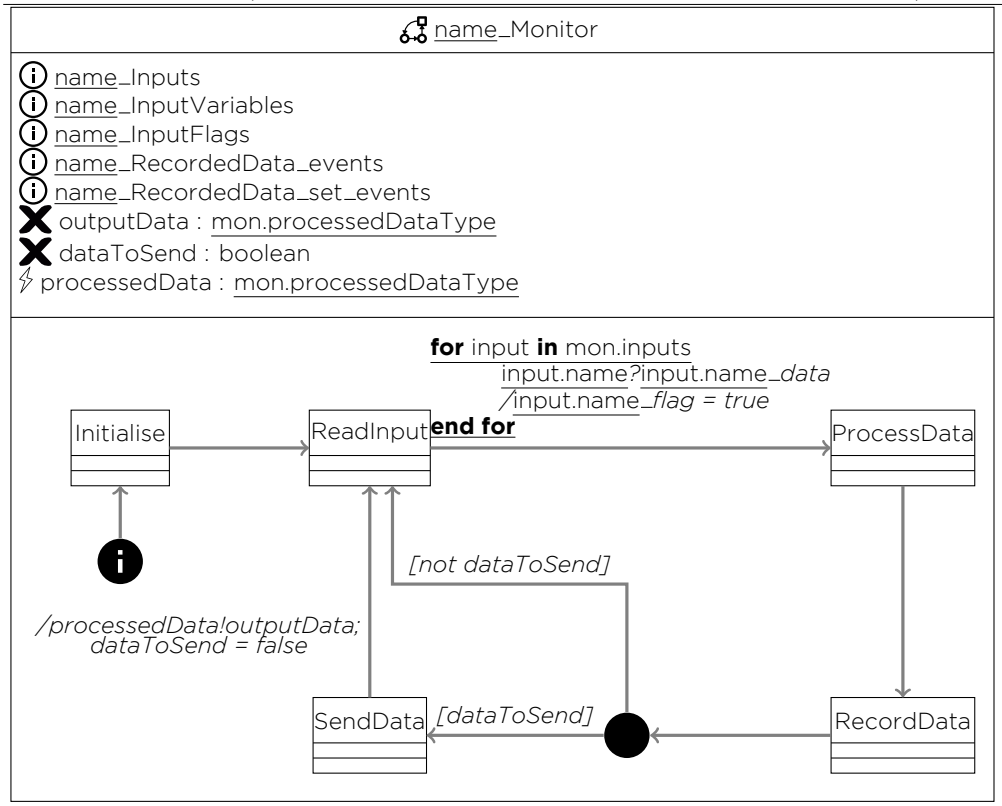


where

pdvrs = ⋃{p : amptn.plan • p.planData}

Rule 15. _Monitor State Machine (Rule 5 in Section 2.4.2)

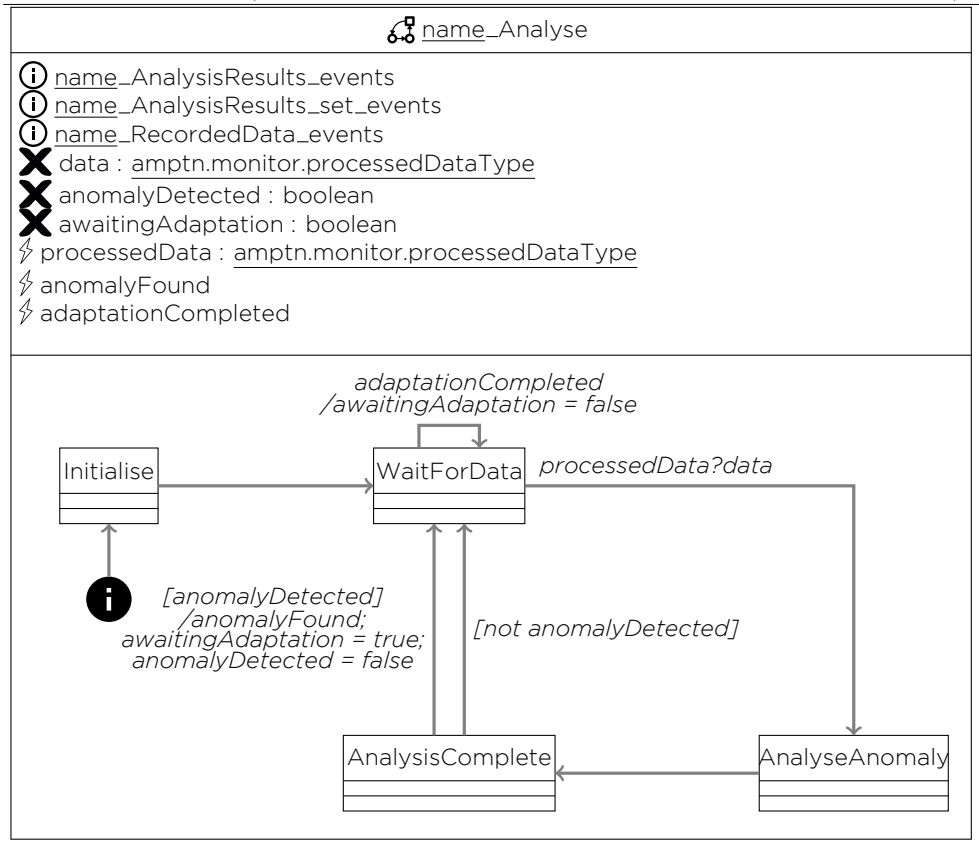
MonitorStateMachine(mon : MonitorComponent, amptn : MAPLEK, name : String) : StateMachine =



Funded by the European Union

Rule 16. _Analyse State Machine (Rule 6 in Section 2.4.2)

AnalyseStateMachine(analyse : AnalyseComponent, amptn : MAPLEK, name : String) : StateMachine =

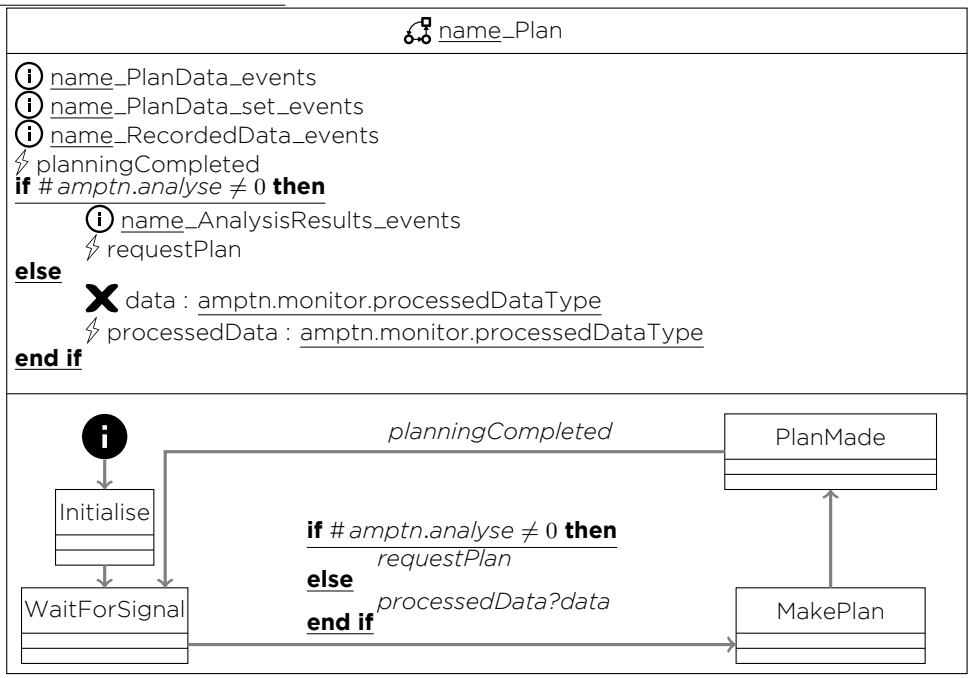


Funded by
the European Union

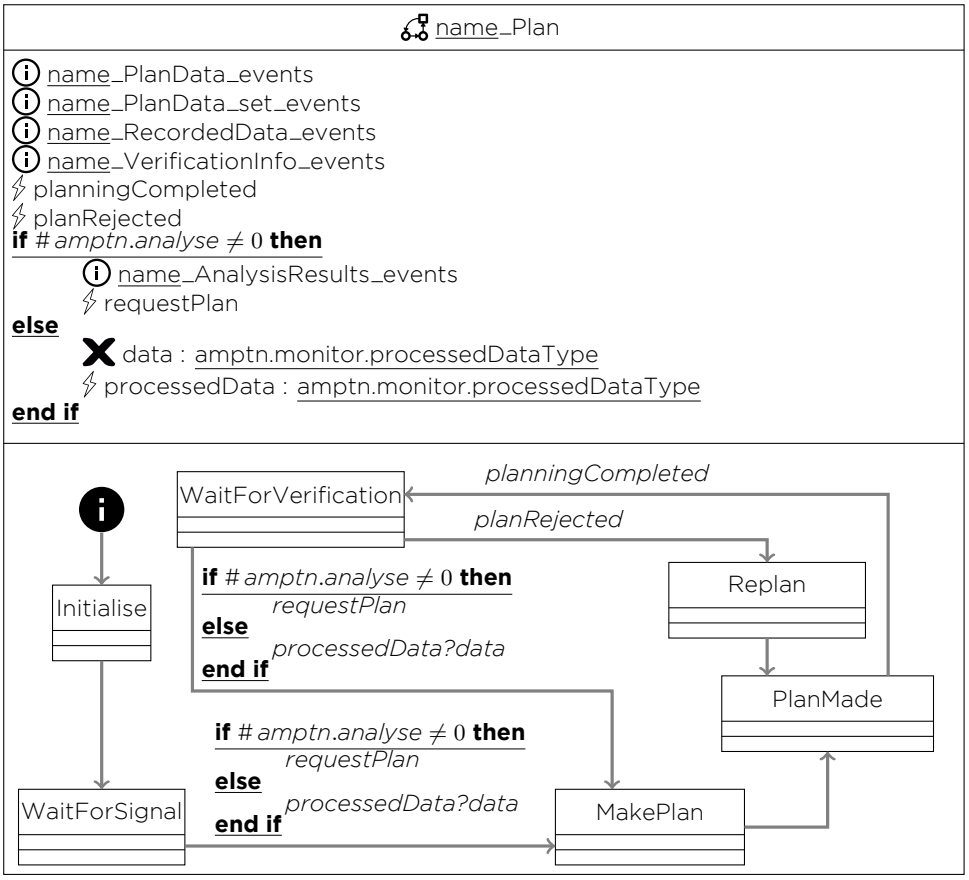
Rule 17. _Plan State Machine (Rule 7 in Section 2.4.2)

PlanStateMachine(plan : PlanComponent, amptn : MAPLEK, name : String) : StateMachine =

if # amptn.legitimate = 0 **then**



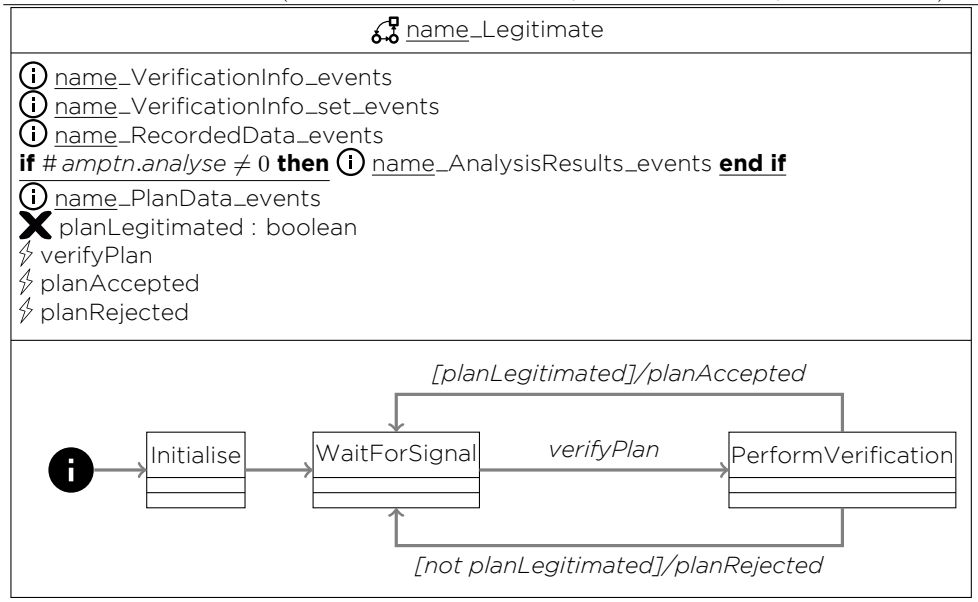
else



end if

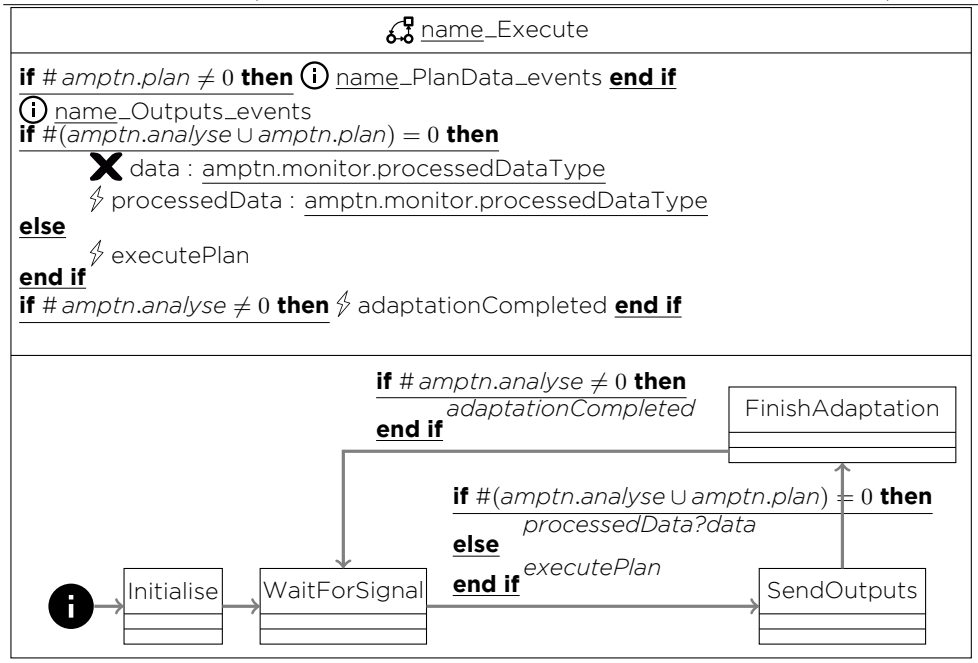
Rule 18. _Legitimate State Machine (Rule 8 in Section 2.4.2)

LegitimateStateMachine(l : LegitimateComponent, amptn : MAPLEK, name : String) : StateMachine =



Rule 19. _Execute State Machine (Rule 9 in Section 2.4.2)

ExecuteStateMachine(e : ExecuteComponent, amptn : MAPLEK, name : String) : StateMachine =



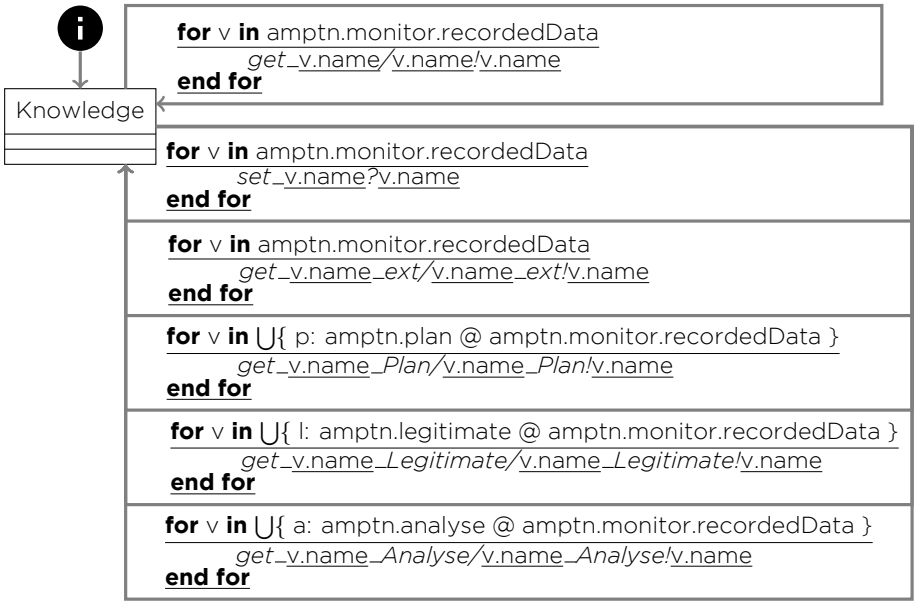
Funded by
the European Union

Rule 20. _Knowledge State Machine (Rule 10 in Section 2.4.2)

KnowledgeStateMachine(amptn : MAPLEK, name : String) : StateMachine =

name_Knowledge

ⓘ name_RecordedData
if # *amptn.analyse* ≠ 0 **then** ⓘ name_AnalysisResults **end if**
if # *amptn.plan* ≠ 0 **then** ⓘ name_PlanData **end if**
if # *amptn.legitimate* ≠ 0 **then** ⓘ name_VerificationInfo **end if**
ⓘ name_RecordedData_events
ⓘ name_RecordedData_set_events
ⓘ name_RecordedData_ext_events
if # *amptn.analyse* ≠ 0 **then** ⓘ name_RecordedData_Analyse_events **end if**
if # *amptn.plan* ≠ 0 **then** ⓘ name_RecordedData_Plan_events **end if**
if # *amptn.legitimate* ≠ 0 **then** ⓘ name_RecordedData_Legitimate_events **end if**
if # *amptn.analyse* ≠ 0 **then**
    ⓘ AnalysisResults_events
    ⓘ AnalysisResults_set_events
    ⓘ AnalysisResults_ext_events
    if # *amptn.plan* ≠ 0 **then** ⓘ AnalysisResults_Plan_events**end if**
    if # *amptn.legitimate* ≠ 0 **then** ⓘ AnalysisResults_Legitimate_events **end if**
end if
if # *amptn.plan* ≠ 0 **then**
    ⓘ PlanData_events
    ⓘ PlanData_set_events
    ⓘ PlanData_ext_events
    if # *amptn.legitimate* ≠ 0 **then** ⓘ PlanData_Legitimate_events **end if**
    ⓘ PlanData_Execute_events
end if
if # *amptn.legitimate* ≠ 0 **then**
    ⓘ VerificationInfo_events
    ⓘ VerificationInfo_set_events
    ⓘ VerificationInfo_ext_events
    if # *amptn.plan* ≠ 0 **then** ⓘ VerificationInfo_Plan_events **end if**
end if

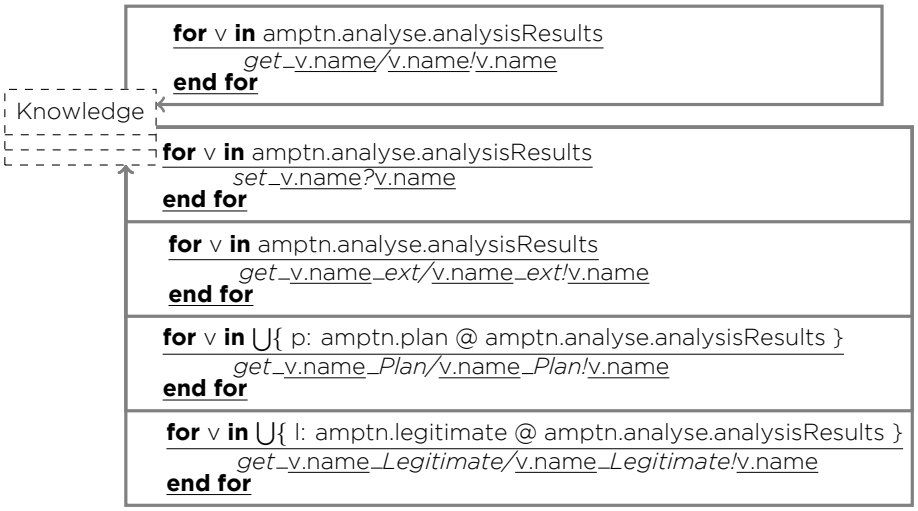


if # *amptn.analyse* ≠ 0 **then** KnowledgeAnalysisResultsTransitions(amptn, name) **end if**
if # *amptn.plan* ≠ 0 **then** KnowledgePlanDataTransitions(amptn, name) **end if**
if # *amptn.legitimate* ≠ 0 **then** KnowledgeVerificationInfoTransitions(amptn, name) **end if**

Funded by
the European Union

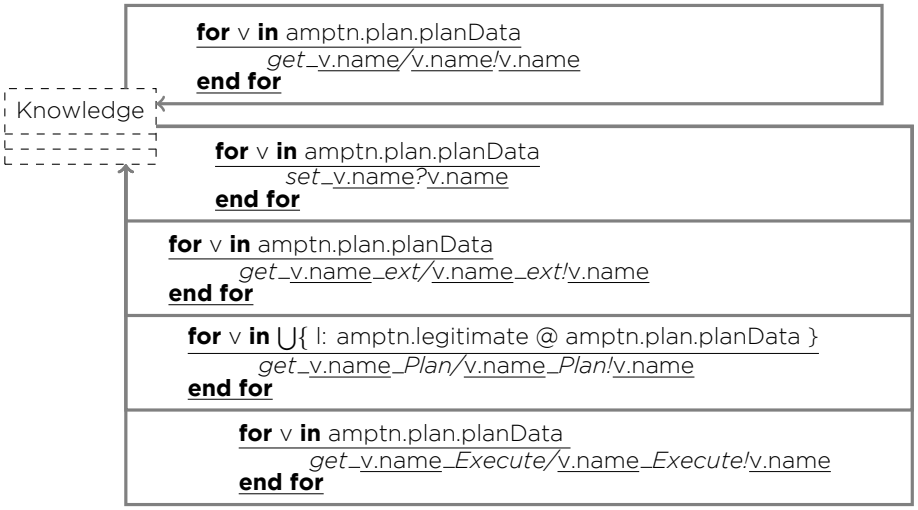
Rule 21. AnalysisResults Transitions for the _Knowledge State Machine

KnowledgeAnalysisResultsTransitions(amptn : MAPLEK, name : String) : Set(Transition) =



Rule 22. PlanData Transitions for the _Knowledge State Machine

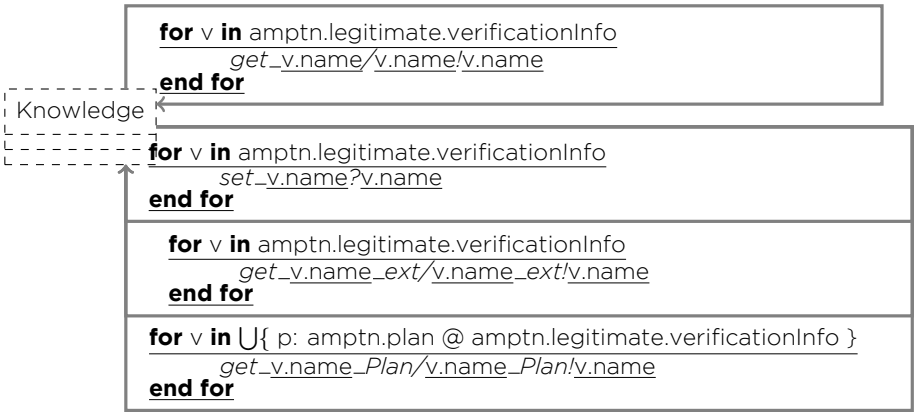
KnowledgePlanDataTransitions(amptn : MAPLEK, name : String) : Set(Transition) =



Funded by
the European Union

Rule 23. VerificationInfo Transitions for the _Knowledge State Machine

KnowledgeVerificationInfoTransitions(amptn : MAPLEK, name : String) : Set(Transition) =



Funded by
the European Union

B ANN Components in RoboChart

B.1 Metamodel and well-formedness conditions

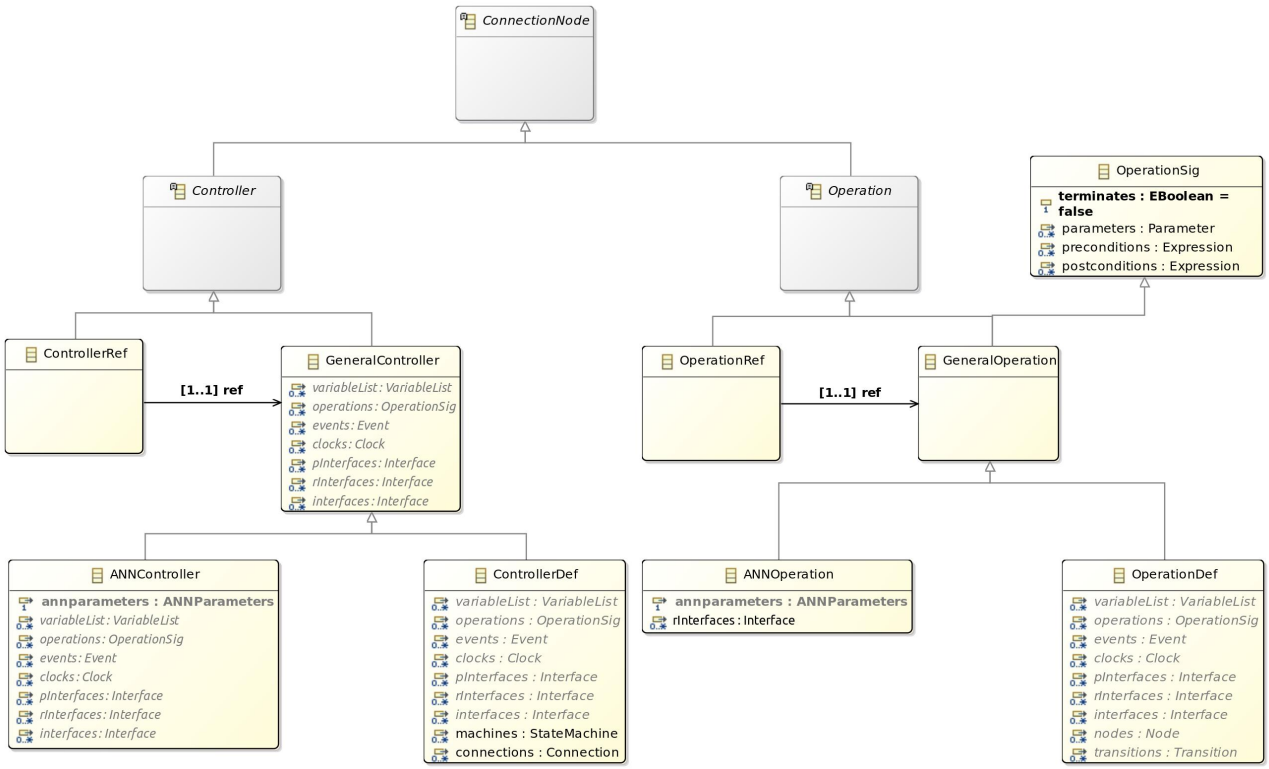


Figure 26: Our modified RoboChart connection metamodel including our new classes: *ANNController*, *ANNOperation*, *GeneralController*, and *GeneralOperation*.

Funded by
the European Union

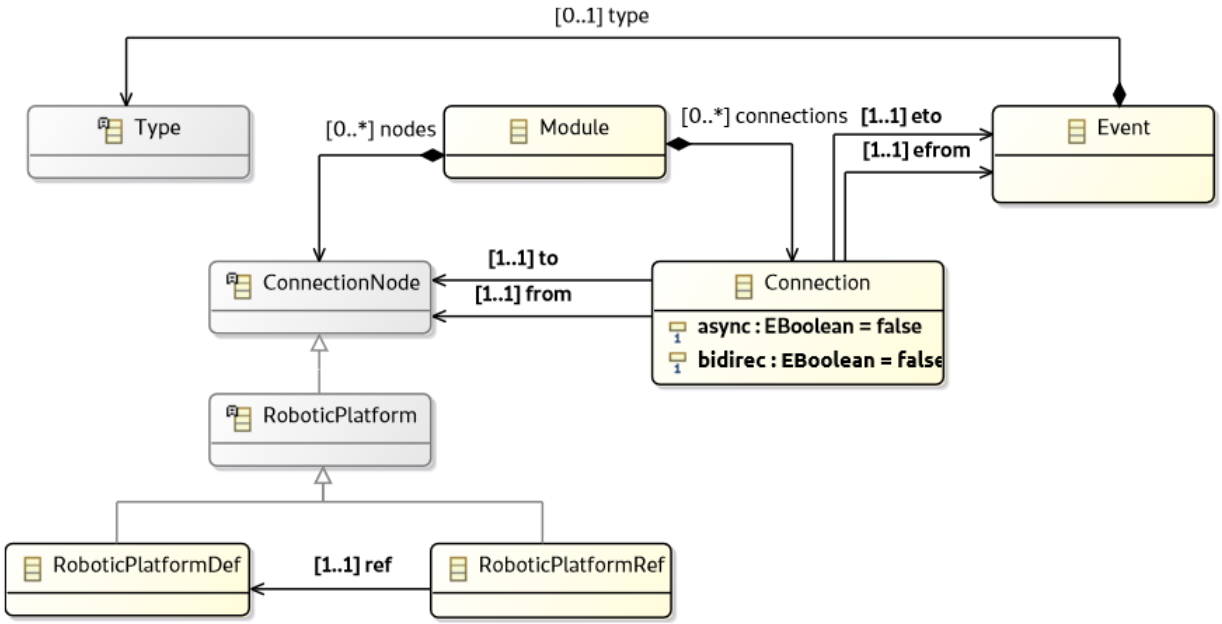


Figure 27: RoboChart module communication metamodel.

WF1	The events of inputContext and outputContext are both non-empty, and layerstructure, weights, inRanges, outRanges, annRange, and biases are non-empty and of the same size, if not null.
WF2	If filename is null, then weights, biases, inRanges, outRanges, annRange, and layerstructure are not.
WF3	Either layerstructure, weights, inRanges, outRanges, annRange, and biases are all null, in which case filename is not, or they are all different from null.
WF4	activationfunction is NOTSPECIFIED if, and only if, filename is not null.
WF5	For every *i*, the size of weights *i* and biases *i* is layerstructure *i*.
WF6	For every *i*, and for all *j*, the size of weights *i j* is layerstructure $(i - 1)$ when *i* is greater than 1, or the size of the events in its inputContext otherwise.
WF7	The connections to and from an ANNController match the nature of the events (inputs and outputs) in their directions and types.
WF8	An ANNController cannot define events, clocks, or variables.
WF9	An ANNController cannot define, provide, or require interfaces.
WF10	An ANNController's inputContext and outputContext's cannot provide or require interfaces.
WF11	An ANNController's inputContext and outputContext cannot define clocks or variables.
WF12	The connections to an ANNController must be to events in its input context, and connections from an ANNController must be from events in its output context.

Table 2: RoboChart ANN well-formedness conditions

Funded by
the European Union

WF13	The sequence inRanges is of length equal to the number of events in the inputContext.
WF14	The sequence outRanges is of length equal to the number of events in the outputContext.
WF15	All elements in inRanges must be pairs where the second element is strictly greater than the first.
WF16	All elements in outRanges must be pairs where the second element is strictly greater than the first.
WF17	The second element in annRange must be strictly greater than the first.
WF18	Each event defined in the inputContext and outputContext must be an OrderedEvent.
WF19	Each index of events defined in the inputContext and outputContext must be sequential and one-indexed.

Table 3: RoboChart ANN Normalisation well-formedness conditions

Funded by
the European Union

B.2 Semantics

$[Context]$
$ActivationFunction ::= RELU \mid LINEAR \mid NOTSPECIFIED$
$Value == \mathbb{A}$
$SeqExp ::= null\_seq \mid list\langle\!\langle \text{seq}\, \mathbb{N} \rangle\!\rangle \mid matrix\langle\!\langle \text{seq seq}\, Value \rangle\!\rangle \mid tensor\langle\!\langle \text{seq seq seq}\, Value \rangle\!\rangle$

ANNParameters

$layerstructure : SeqExp$
$weights : SeqExp$
$biases : SeqExp$
$activationfunction : ActivationFunction$
$inputContext : Context$
$outputContext : Context$
$inRanges : SeqExp$
$outRanges : SeqExp$
$annRange : (Value \times Value)$

ANN

$annparameters : ANNParameters$

ANNController

ANN

Figure 28: RoboChart types for ANN translations, specified in Z.

Funded by
the European Union

Rule 3. Function ANNChannelDecl
ANNChannelDecl(C : ANNController) : Program =

channel *layerRes* : $\mathbb{N} \times \mathbb{N} \times$ *Value*

channel *nodeOut* : $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times$ *Value*

for i in (**1**, # **InRanges**)

  channel inEvents(i)_in : *Value*

endfor

for i in (**1**, # **OutRanges**)

  channel outEvents(i)_out : *Value*

endfor

channel *terminate*

where

inEvents = order(allEvents(C.annparameters.inputContext))

outEvents = order(allEvents(C.annparameters.outputContext))

Funded by
the European Union

Rule 4. Function ANNConstants

ANNConstants(C : ANNController) : Par =

weights : seq seq seq *Value*;

biases : seq seq *Value*;

inRanges : seq(*Value* × *Value*);

outRanges : seq(*Value* × *Value*);

annRange : *Value* × *Value*;

norm : (*Value* × (*Value* × *Value*) × (*Value* × *Value*)) → *Value*;

normI : (ℕ × *Value*) → *Value*;

denormO : (ℕ × *Value*) → *Value*;

relu : *Value* → *Value*

layerInput : seq ℕ

⊢

weights = ((tensor ~) C.annparameters.weights) ∧

biases = ((matrix ~) C.annparameters.biases) ∣ n ∧

inRanges = ((list ~) C.annparameters.inRanges) ∧

outRanges = ((list ~) C.annparameters.outRanges) ∧

annRange = C.annparameters.annRange ∧

layerInput = LStructure(0)⌢

((list ~) C.annparameters.layerstructure) ∧

∀ *n* : ℕ •

∀ *r*, *r*′ : *Value* × *Value* ∣ *r*.2 > *r*.1 ∧ *r*′.2 > *r*′.1 •

∀ *x* : *Value* •

norm(*x*, *r*, *r*′) = (((*x* − *r*.1))/(*r*.2 − *r*.1)) ∗ (*r*′.2 − *r*′.1) + *r*′.1 ∧

normI(*n*, *x*) = *norm*(*x*, *inRanges*(*n*), *annRange*) ∧

denormO(*n*, *x*) = *norm*(*x*, *annRange*, *outRanges*(*n*)) ∧

(*x* < 0 ⇒ (*x*, 0) ∈ *relu*) ∧

(*x* ≥ 0 ⇒ (*x*, *x*) ∈ *relu*)

Funded by
the European Union

Rule 5. Function HiddenLayers
HiddenLayers(C : ANNController) : CSPAction =

HL(**l**) =

if(**l** == **1**)

then

(*HiddenLayer*(l, LStructure(C, l), LStructure(C, l − 1)))

else

(**HL**(**l** − **1**)

⟦ | {| *layerRes*.(l − 1) |} | ⟧

HiddenLayer(l, LStructure(C, l), LStructure(C, l − 1))

within

HL(**layerNo**(**C**) − **1**)

Rule 6. Function OutputLayer
OutputLayer(C : ANNController) : CSPAction =

⟦{| *layerRes*.(layerNo(C) − 1) |}⟧ *i* : 1 .. LStructure(C, layerNo(C)) •

Node(layerNo(C), *i*, LStructure(C, layerNo(C) − 1)))

Rule 7. Function CircANN
CircANN(C : ANNController) : CSPAction =

((*Interpreter* ⟦ | {| *layerRes*.0, *layerRes*.layerNo(C) |} | ⟧ *ANN*) \ {| *layerRes* |})

△ *terminate* ⟶ **Skip**

Funded by
the European Union

Rule 8. Function Interpreter
Interpreter(C : ANNController) : CSPAction) =

$(|||\ i : 1..\#\text{inRanges} \bullet \text{inEvents(i)\_in}?x \longrightarrow \mathit{layerRes}.0.\underline{i}!(\mathit{normI}(\underline{i}, x)) \longrightarrow \textbf{Skip})$;
$(|||\ i : 1..\#\text{outRanges} \bullet \mathit{layerRes}.\text{layerNo(C)}.\underline{i}?y \longrightarrow \text{outEvents(i)\_out}!(\mathit{denormO}(\underline{i}, y)) \longrightarrow \textbf{Skip})$;
Interpreter

where

inEvents = order(allEvents(C.annparameters.inputContext))

outEvents = order(allEvents(C.annparameters.outputContext))

Rule 9. Function LStructure
LStructure(C : ANNController, i : $\mathbb{N}$) : $\mathbb{N}$ =

if(**i** == **0**)

then

#allEvents(C.annparameters.inputContext)

else

((list ~) C.annparameters.layerstructure) i

Rule 10. Function layerNo (number of layers)
layerNo(C : ANNController) : $\mathbb{N}$ =

#((list ~) C.annparameters.layerstructure)

Funded by
the European Union

C Conformance Theory

C.1 Relations

This section recalls some basic constructs from the alphabetised relational calculus used in the Unifying Theories of Programming (UTP) framework. In UTP, predicates are interpreted as relations over variables drawn from a program's alphabet, which includes both before and after values of variables. These are distinguished syntactically: an unprimed variable (e.g., v) denotes its value before execution, while a primed variable (v') denotes its value after execution.

Definition C.1 (Relation theory:- identity)**.** *The identity relation over an alphabet A is defined as:*

$$\mathit{II}_A = (v' = v) \qquad \text{where } A = \{v, v'\}$$

where A is the set of variables $\{v, v'\}$. This relation states that the post-state is equal to the pre-state for every variable in the alphabet, and thus represents no change. It is used in UTP as the neutral element for sequential composition.

Definition C.2 (Relation theory:- sequence)**.** *Sequential composition of two relations P ; Q is defined by existentially quantifying over the intermediate state:*

$$P \,;\, Q = \exists x_0 \bullet P[x_0/x'] \land Q[x_0/x]$$

This definition formalises composition via state-passing: the post-state of P becomes the pre-state of Q. The notation $P[x_0/x']$ means that we substitute x_0 for x' in P, and similarly for Q. The fresh variable x_0 represents the intermediate state vector and x and x' represent pre- and post-state vectors.

Definition C.3 (Relation theory:- initialised block)**.** *Local variable initialisation is modelled by:*

$$x \text{ not free in } e \Rightarrow ((\textbf{var } x := e \bullet P) = (\exists x, x' \bullet P[e/x]))$$

This introduces a fresh variable x' initialised to e', where x is not free in e. The substitution $P[e/x]$ replaces all instances of x in P with e, and the quantification removes x and x' from the alphabet. This operation is used to model simulation blocks later in the paper.

Law C.4 (Relation theory:- leading assignment)**.** *Assignments are modelled relationally by substitution into the post-state:*

$$x := e \,;\, P = P[e/x]$$

This law states that executing $x := e$ followed by a process P is equivalent to substituting e for x in P, reflecting the semantics of instantaneous assignment in the relational model.

Definition C.5 (Relation theory:- alphabet extension)**.** *Alphabet extension adds a variable x to the alphabet without affecting behaviour:*

$$P_{+x} = P \land (x' = x)$$

This ensures that x remains unchanged during the execution of P, a useful construct for reasoning about processes in larger alphabets or under variable scoping.

Funded by
the European Union

These relational definitions form the foundation for the design and reactive theories presented in the subsequent sections. In particular, they enable a clean algebraic treatment of trace transformations, substitutions, and process approximations central to our theory of neural network conformance. These definitions underpin the process approximation semantics introduced in Section 5, and are essential for interpreting process simulations in terms of CSP traces and updates.

C.2 Designs

The UTP theory of designs provides a unified framework for modelling pre- and post-condition specifications. It forms the semantic foundation for describing total correctness properties of programs and extends naturally to reactive systems through healthiness conditions such as **R1**, **R2**, and **R3**. In this section, we present the basic theory of designs and their properties, which we later use to express and reason about neural network behaviours as reactive CSP processes.

Definition C.6 (Design theory:- design)**.** *A* design *is a UTP relation of the form:*

$$(P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q$$

This predicate specifies that if the program starts in a state where it is enabled (denoted by the boolean variable ok), and if the precondition P holds, then the program must terminate in a state where it is again enabled (denoted by the boolean variable ok′) and satisfies postcondition Q.

This formulation captures total correctness: if the design is invoked in a state satisfying P, then it will not fail, and will establish Q.

Several lemmas show how substitution for ok and ok' affects a design. We use the UTP community convention that $P^t = P[true/ok']$ and $P^f = P[false/ok']$.

Definition C.7 (Design theory:- termination substitution)**.**

$$P^b = P[b/ok'], \qquad P^t = P[true/ok'], \qquad P^f = P[false/ok']$$

Lemma 21 (Design theory:- $P[true/ok]^t$ simplification)**.** *The relation $P[true/ok]^t$ describes the behaviour of a design P that is properly started ($ok = true$) and properly terminates ($ok' = true$). These substitutions simplify a design to its implication form: if the precondition P holds before execution, then the postcondition Q will hold after execution.*

$$(P \vdash Q)[true/ok]^t = P \Rightarrow Q$$

Proof.

$$\begin{aligned}
& (P \vdash Q)[true/ok]^t \\
= & \quad \{ \text{ Definition 21: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& (ok \wedge P \Rightarrow ok' \wedge Q)[true/ok]^t \\
= & \quad \{ \text{ Logic theory:- substitution } \}
\end{aligned}$$

Funded by the European Union

$$
\begin{aligned}
& true \wedge P \Rightarrow true \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& P \Rightarrow Q
\end{aligned}
$$

□

Lemma 22 (Design theory:- $P[true/ok]^f$ simplification)**.** *Substituting true for ok and false for ok′ yields the negation of the precondition. That is, if the design is properly started (ok = true) buut fails to terminate (ok′ = false), then the precondition must have been violated (¬ P).*

$(P \vdash Q)[true/ok]^f = \neg P$

Proof.

$$
\begin{aligned}
& (P \vdash Q)[true/ok]^f \\
= & \quad \{ \text{ Definition 21: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \} \\
& (ok \wedge P \Rightarrow ok' \wedge Q)[true/ok]^f \\
= & \quad \{ \text{ Logic theory:- substitution } \} \\
& true \wedge P \Rightarrow false \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& \neg P
\end{aligned}
$$

□

Finally, we have two lemmas that show what happens Substituting false for ok yields true in both true and false branches, capturing vacuous correctness when a process is not enabled. The value of *ok* reflects whether the design is being observed in a context where it is allowed to start. So:

- If ($ok = true$): the design is executed, and both the precondition and postcondition matter.
- If ($ok = false$): the design is not properly started; thus, its behaviour is unconstrained: the precondition does not have to hold, and the postcondition is irrelevant.

This reflects **abort**, a divergent or undefined context in the observational semantics.

Lemma 23 (Design theory:- $P[false/ok]^t$ simplification)**.**

$(P \vdash Q)[false/ok]^t = true$

Funded by
the European Union

Proof.

$$
\begin{aligned}
& (P \vdash Q)[\mathit{false}/\mathit{ok}]^t \\
= & \quad \{ \text{ Definition 21: Design theory:- design: } (P \vdash Q) = \mathit{ok} \wedge P \Rightarrow \mathit{ok}' \wedge Q \ \} \\
& (\mathit{ok} \wedge P \Rightarrow \mathit{ok}' \wedge Q)[\mathit{false}/\mathit{ok}]^t \\
= & \quad \{ \text{ Logic theory:- substitution } \} \\
& \mathit{false} \wedge P \Rightarrow \mathit{true} \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& \mathit{true}
\end{aligned}
$$

□

Lemma 24 (Design theory:- $P[\mathit{false}/\mathit{ok}]^f$ simplification)**.**

$(P \vdash Q)[\mathit{false}/\mathit{ok}]^f = \mathit{true}$

Proof.

$$
\begin{aligned}
& (P \vdash Q)[\mathit{false}/\mathit{ok}]^f \\
= & \quad \{ \text{ Definition 21: Design theory:- design: } (P \vdash Q) = \mathit{ok} \wedge P \Rightarrow \mathit{ok}' \wedge Q \ \} \\
& (\mathit{ok} \wedge P \Rightarrow \mathit{ok}' \wedge Q)[\mathit{false}/\mathit{ok}]^f \\
= & \quad \{ \text{ Logic theory:- substitution } \} \\
& \mathit{false} \wedge P \Rightarrow \mathit{false} \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& \mathit{true}
\end{aligned}
$$

□

These simplification laws are used frequently in the refinement proofs throughout the rest of the paper, especially for computing conformance under approximation and simulation.

Designs admit case analysis and standard substitution principles. UTP uses the notation $[P]$ to den ote the universal closure of predicate P over its alphabet. If the alphabet contains the state vector v, then $[P] = \forall v \bullet P$. A design's alphabet contains the observational variables ok and ok', and possibly some programming variables. If P is a design, then $[P]$ is universally quantifying, in particular, ok'. This variable is,

Funded by
the European Union

of course, a boolean, so there are two cases: termination and nontermination. This is reflected in the following law.

Law C.8 (Design theory:- cases)**.**

$$[P] = [P^f] \wedge [P^t]$$

Law C.9 (Design theory:- abort)**.**

$$(P \vdash Q)^f = ok \Rightarrow P^f$$

Law C.10 (Design theory:- not abort)**.**

$$(P \vdash Q)^t = (ok \wedge P^t \vdash Q^t)$$

Law C.11 (Design theory:- existential design)**.** *Existential quantification over a design distributes in the same way as in the underlying implication.*

$$(\exists x \bullet (P \vdash Q)) = (\forall x \bullet P \vdash \exists x \bullet Q)$$

These laws facilitate reasoning about designs algebraically and are foundational in the reactive extensions in Section 4.

The UTP design theory includes algebraic laws for refinement and nondeterministic choice. Hoare and He define a correctness relation between relations specifying computations. They define Q to be correct with respect to P if, and only if, $Q \Rightarrow P$. So, all the behaviours of Q must also be behaviours of P. This notion applies both to single-predicate relations and to designs. Their Theorem 3.1.2 expresses this statement of correctness as an equivalent constraint on P and Q preconditions and postconditions.

Theorem 6 (Design theory:- design refinement)**.**

$$((P1 \vdash P2) \sqsubseteq (Q1 \vdash Q2)) = [P1 \Rightarrow Q1] \wedge [P1 \wedge Q2 \Rightarrow P2]$$

Proof. See Hoare & He, Theorem 3.1.2.

□

As they point out, the message of this theorem is that $(Q1 \vdash Q2)$ is stronger because it has a weaker assumption $Q1$, and so it can be used more widely; furthermore, in all circumstances where $(P1 \vdash P2)$ can be used, $(Q1 \vdash Q2)$ has a stronger commitment, so its behaviour can be more readily predicted and controlled.

Nondeterministic choice between relations is simply disjunction. But the added richness of designs means that we can understand nondeterministic behaviour in terms of assumptions and commitments.

Theorem 7 (Design theory:- design choice)**.**

$$(P1 \vdash P2) \sqcap (Q1 \vdash Q2) = (P1 \wedge Q1 \vdash P2 \vee Q2)$$

Funded by
the European Union

Proof. See Hoare & He, Theorem 3.1.4.

□

We cannot control the choice in the nondeterministic design $(P1 \vdash P2) \sqcap (Q1 \vdash Q2)$. It may abort if $(P1 \vdash P2)$ is chosen and $P1$ does not hold; but may also abort if $(Q1 \vdash Q2)$ is chosen and $Q1$ does not hold. The effect of the nondeterministic choice is certainly defined either by $P2$ or by $Q2$.

To support reactive processes, we extend the design theory with healthiness conditions, starting with **R1**, which enforces monotonicity of the trace.

Definition C.12 (Design theory:- **R1**)**.**

$$\mathbf{R1}(P) = P \wedge (tr \leq tr')$$

This ensures that events are never removed from the trace as the process evolves.

While designs are sufficient to model sequential behaviour, reactive systems such as neural controllers require explicit treatment of time and interaction. The **R1** law enforces trace monotonicity, forming the foundation of CSP-style reactivity.We define an **R1**-design as:

Definition C.13 (Design theory:- **R1**-design)**.**

$$(P \models Q) = \mathbf{R1}(P \vdash Q)$$

We use **R1**-designs as the foundation for reactive processes and show in Section 4 how they are further extended with **R2** and **R3** to model CSP semantics faithfully.

The next six results are used to simplify designs and **R1**-designs.

Lemma 25 (Design theory:- design *ok* pre)**.**

$$(P \vdash Q) = (ok \wedge P \vdash Q)$$

Proof.

$$\begin{aligned}
& P \vdash Q \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& ok \wedge P \Rightarrow ok' \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& ok \wedge ok \wedge P \Rightarrow ok' \wedge Q \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& ok \wedge P \vdash Q
\end{aligned}$$

□

Funded by the European Union

Corollary C.14 (Design theory:- **R1**-design *ok* pre)**.**

$(P \models Q) = (ok \wedge P \models Q)$

Proof. From Lemma 25: Design theory:- design *ok* pre:

$(P \vdash Q) = (ok \wedge P \vdash Q)$

□

Lemma 26 (Design theory:- design precondition simp)**.**

$(P \vdash P \wedge Q) = (P \vdash Q)$

Proof.

$$
\begin{aligned}
& P \vdash Q \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& ok \wedge P \Rightarrow ok' \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& ok \wedge P \Rightarrow ok' \wedge P \wedge Q \\
= & \quad \{ \text{ Definition: C.6: Design theory:- design}(P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& P \vdash P \wedge Q
\end{aligned}
$$

□

Corollary C.15 (Design theory:- **R1**-design precondition simp)**.**

$(P \models P \wedge Q) = (P \models Q)$

Proof. From Lemma 26: Design theory:- design precondition simp:

$(P \vdash P \wedge Q) = (P \vdash Q)$

□

Lemma 27 (Design theory:- design *ok* post)**.**

$(P \vdash Q) = (P \vdash ok \wedge Q)$

Funded by
the European Union

Proof.

$$
\begin{aligned}
& P \vdash Q \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& ok \wedge P \Rightarrow ok' \wedge Q \\
= & \quad \{ \text{ Logic theory:- propositions } \} \\
& ok \wedge P \Rightarrow ok' \wedge ok \wedge Q \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& P \vdash ok \wedge Q
\end{aligned}
$$

□

Corollary C.16 (Design theory:- **R1**-design *ok* post)**.**

$(P \models Q) = (P \models ok \wedge Q)$

Proof. From Lemma 27: Design theory:- design *ok* post:

$(P \vdash Q) = (P \vdash ok \wedge Q)$

□

The following lemma and its corollary show how substitution distributes through a design and an **R1** design.

Lemma 28 (Design theory:- design-substitution)**.**

x *isn't* $\{ok, ok'\} \Rightarrow (P \vdash Q)[e/x] = (P[e/x] \vdash Q[e/x])$

Proof.

$$
\begin{aligned}
& (P \vdash Q)[e/x] \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& (ok \wedge P \Rightarrow ok' \wedge Q)[e/x] \\
= & \quad \{ \text{ Logic theory:- substitution : } x \text{ isn't } \{ok, ok'\} \ \} \\
& ok \wedge P[e/x] \Rightarrow ok' \wedge Q[e/x] \\
= & \quad \{ \text{ Definition C.6: Design theory:- design: } (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \ \} \\
& P[e/x] \vdash Q[e/x]
\end{aligned}
$$

□

Funded by
the European Union

Corollary C.17 (Design theory:- **R1**-design-substitution)**.**

x *isn't* $\{ok, ok', tr, tr'\} \Rightarrow (P \models Q)[e/x] = (P[e/x] \models Q[e/x])$

Proof.

$$
\begin{array}{ll}
 & (P \models Q)[e/x] \\
= & \quad \left\{ \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design: } (P \models Q) = \mathbf{R1}(P \vdash Q) \right\} \\
 & (\mathbf{R1}(P \vdash Q))[e/x] \\
= & \quad \left\{ \text{Logic theory:- substitution : } x \text{ isn't } \{tr, tr'\} \right\} \\
 & \mathbf{R1}((P \vdash Q)[e/x]) \\
= & \quad \left\{ \begin{array}{l} \text{Lemma 28: Design theory:- design-substitution:} \\ x \text{ isn't } \{ok, ok'\} \Rightarrow (P \vdash Q)[e/x] = (P[e/x] \vdash Q[e/x]) \end{array} \right\} \\
 & \mathbf{R1}(P[e/x] \vdash Q[e/x]) \\
= & \quad \left\{ \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design: } (P \models Q) = \mathbf{R1}(P \vdash Q) \right\} \\
 & P[e/x] \models Q[e/x]
\end{array}
$$

□

The **R1** healthiness function distributes through disjunction, since it is conjunctive.

Lemma 29 (Design theory:- **R1** disjunctive)**.**

$\mathbf{R1}(P \vee Q) = \mathbf{R1}(P) \vee \mathbf{R1}(Q)$

Proof.

$$
\begin{array}{ll}
 & \mathbf{R1}(P \vee Q) \\
= & \quad \left\{ \text{Definition C.12: Design theory:- } \mathbf{R1}\text{: } \mathbf{R1}(P) = P \wedge (tr \leq tr') \right\} \\
 & (P \vee Q) \wedge (tr \leq tr') \\
= & \quad \left\{ \text{Logic theory:- propositions} \right\} \\
 & (P \wedge (tr \leq tr')) \vee (Q \wedge (tr \leq tr')) \\
= & \quad \left\{ \text{Definition C.12: Design theory:- } \mathbf{R1}\text{: } \mathbf{R1}(P) = P \wedge (tr \leq tr') \right\} \\
 & \mathbf{R1}(P) \vee \mathbf{R1}(Q)
\end{array}
$$

□

Funded by
the European Union

C.3 CSP

In order to model the behaviour of reactive neural systems, we extend the UTP theory of designs to incorporate the reactive semantics of Communicating Sequential Processes (CSP). Reactive systems are those that maintain ongoing interaction with their environment, and thus require a more expressive model than traditional sequential programs. The core idea is to represent system execution not only in terms of initial and final states but also in terms of observable behaviours: most notably, event traces and intermediate states.

We add two new observations to the alphabet of **R1**-designs: the boolean $wait'$ that describes a situation when a program is waiting for interaction with its environment. This is the essence of reactivity, so we call such programs reactive processes. This variable lets us distinguish intermediate observations from final ones. If $wait'$ is true, then all the other dashed variables are also intermediate.

To reason about such intermediate states, we introduce a substitution notation:

Definition C.18 (Design theory:- waiting)**.**

$$P_b = P[b/wait']$$

The next lemma shows that the treatment of intermediate states respects the substitutional structure of the R1-design, and helps reduce reactive designs to their non-waiting form for analysis.

Lemma 30 (Design theory:- **R1** design not waiting)**.**

$$(P \models Q)_f = (P_f \models Q_f)$$

Proof. From Lemma C.17: Design theory:- **R1**-design-substitution:

$$x \text{ isn't } \{ok, ok', tr, tr'\} \Rightarrow (P \models Q)[e/x] = (P[e/x] \models Q[e/x])$$

□

We lift the relational identity to reactive processes. The relation II_{rea} describes an identity that does not modify the trace and preserves the waiting condition. It allows both initial and final observations: either the process aborts ($\neg\, ok$) and the trace is unchanged, or it terminates successfully (ok') with no change to trace or waiting.

Definition C.19 (CSP theory:- identity)**.**

$$\mathit{II}_{rea} = \neg\, ok \wedge (tr \leq tr) \vee ok' \wedge \mathit{II}$$

The **R3** healthiness condition ensures that a process either behaves as the identity when $wait$ is true (during interaction), or behaves as P when $wait$ is false (when no interaction is expected).

Definition C.20 (CSP theory:- **R3** healthiness)**.**

$$\mathbf{R3}(P) = \mathit{II}_{rea} \lhd wait \rhd P$$

Funded by the European Union

The conditional expression of **R3** captures the reactive essence of CSP: a process is either quiescent and awaiting input (*wait* is true), or active and progressing (*wait* is false).

In the UTP, the **R2** healthiness condition is one of the foundational constraints on reactive processes. Reactive processes are models of systems that interact with their environment over time, maintaining an ongoing dialogue rather than producing a single output and terminating. The **R2** condition captures the idea of history independence: the behaviour of a reactive process should not depend on the trace of past events, but only on the extension to that trace made by the current interaction. This reflects a modular, stepwise view of computation in reactive systems. The **R2** healthiness condition can be defined as follows:

Definition C.21 (CSP theory:- **R2**)**.**

$$\mathbf{R2}(P(tr, tr')) = \bigsqcap s \bullet P(s, s \frown (tr' - tr))$$

This formulation ensures that P is invariant under changes to the trace prefix, provided the suffix remains constant. That is, if P behaves a certain way when extending trace tr with e, it must behave identically when extending any other trace s with the same suffix e. **R2** can also be presented as a Galois connection between trace extensions and prefix invariance.

We simplify the following presentation by assuming that the predicates we consider as reactive processes are all **R2**-healthy. This assumption lets us omit the explicit application of **R2**, simplifying derivations.

We now formalise the closure properties that reactive processes must satisfy. This theorem states that a reactive process can be constructed from its behaviour in the non-waiting case by combining the false and true termination branches. It parallels Hoare and He's Theorem 8.2.2 and is a foundational result for defining CSP processes within UTP. It states that every CSP process can be expressed as an **R3**-healthy **R1**-design. The substitutions extract the divergent behaviour (P^f_f) and the postcondition (P^f_f).

Theorem 8 (CSP theory:- CSP reactive design closure)**.**

$$P = R(\neg\, P^f_f \vdash P^t_f)$$

Our assumption about **R2** healthiness leads to the following corollary.

Corollary C.22 (CSP theory:- **R3**-**R1**-design process)**.**

$$P \textit{ is } \mathbf{R2}\textit{-healthy} \;\Rightarrow P = \mathbf{R3}(\neg\, P^f_f \models P^t_f)$$

Proof.

$$\begin{array}{ll} & P \\ = & \left\{ \text{Theorem 8:- CSP theory:- CSP reactive design closure : } P = R(\neg\, P^f_f \vdash P^t_f) \right\} \\ & \mathbf{R}(\neg\, P^f_f \vdash P^t_f) \end{array}$$

Funded by
the European Union

$$
\begin{array}{ll}
= & \left\{ \ \mathbf{R} = \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1} \ \right\} \\
& \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1}(\neg\, P_f^f \vdash P_f^t) \\
= & \left\{ \ \text{assumption: } P \text{ is } \mathbf{R2}\text{-healthy} \ \right\} \\
& \mathbf{R3} \circ \mathbf{R1}(\neg\, P_f^f \vdash P_f^t) \\
= & \left\{ \ \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design: } (P \models Q) = \mathbf{R1}(P \vdash Q) \ \right\} \\
& \mathbf{R3}(\neg\, P_f^f \models P_f^t)
\end{array}
$$

□

There now follow 10 lemmas about substitution in reactive designs.

Lemma 31 (CSP theory:- **R3** not waiting)**.**

$(\mathbf{R3}(P))_f = P_f$

Proof.

$$
\begin{array}{ll}
& (\mathbf{R3}(P))_f \\
= & \left\{ \ \text{Definition C.20: CSP theory:- } \mathbf{R3} \text{ healthiness: } \mathbf{R3}(P) = \varPi_{rea} \lhd wait \rhd P \ \right\} \\
& (\varPi_{rea} \lhd wait \rhd P)_f \\
= & \left\{ \ \text{Logic theory:- substitution} \ \right\} \\
& \varPi_{reaf} \lhd wait_f \rhd P_f \\
= & \left\{ \ \text{Logic theory:- substitution : } P_b = P[b/wait'] \ \right\} \\
& \varPi_{reaf} \lhd false \rhd P_f \\
= & \left\{ \ \text{conditional} \ \right\} \\
& P_f
\end{array}
$$

□

Lemma 32 (CSP theory:- **R3** Simplification)**.**

$(\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok]$

Proof.

$(\mathbf{R3}(P))^c_a[b/ok]$

Funded by the European Union

$$
\begin{array}{ll}
= & \left\{ \text{Definition C.20: CSP theory:- } \mathbf{R3} \text{ healthiness: } \mathbf{R3}(P) = \mathit{II}_{rea} \lhd \mathit{wait} \rhd P \right\} \\
& (\mathit{II}_{rea} \lhd \mathit{wait} \rhd P)^{c}_{a}[b/ok] \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
& \mathit{II}_{rea}{}^{c}_{a}[b/ok] \lhd \mathit{wait}_a \rhd P^{c}_{a}[b/ok] \\
= & \left\{ \begin{array}{l} \text{Definition C.19: CSP theory:- identity:} \\ \mathit{II}_{rea} = \neg\, ok \wedge (tr \leq tr) \vee ok' \wedge \mathit{II} \end{array} \right\} \\
& (\neg\, ok \wedge (tr \leq tr) \vee ok' \wedge (tr' = tr) \wedge (\mathit{wait}' = \mathit{wait}))^{c}_{a}[b/ok] \lhd \mathit{wait}_a \rhd P^{c}_{a}[b/ok] \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
& \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (\mathit{wait}' = a) \lhd a \rhd P^{c}_{a}[b/ok]
\end{array}
$$

□

Lemma 33 (CSP theory:- $\mathbf{R3}(P)_t[true/ok]^t$ simplification)**.**

$(\mathbf{R3}(P))_t[true/ok]^t = (tr' = tr) \wedge \mathit{wait}'$

Proof.

$$
\begin{array}{ll}
& (\mathbf{R3}(P))_t[true/ok]^t \\
= & \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^{c}_{a}[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (\mathit{wait}' = a) \lhd a \rhd P^{c}_{a}[b/ok] \end{array} \right\} \\
& \neg\, true \wedge (tr \leq tr) \vee true \wedge (tr' = tr) \wedge (\mathit{wait}' = true) \lhd true \rhd P_t[true/ok]^t \\
= & \left\{ \text{Logic theory:- propositions} \right\} \\
& (tr' = tr) \wedge \mathit{wait}'
\end{array}
$$

□

Lemma 34 (CSP theory:- $\mathbf{R3}(P)_t[true/ok]^f$ simplification)**.**

$(\mathbf{R3}(P))_t[true/ok]^f = false$

Proof.

$$
\begin{array}{ll}
& (\mathbf{R3}(P))_t[true/ok]^f \\
= & \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^{c}_{a}[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (\mathit{wait}' = a) \lhd a \rhd P^{c}_{a}[b/ok] \end{array} \right\}
\end{array}
$$

Funded by
the European Union

$$\neg\, true \wedge (tr \leq tr) \vee false \wedge (tr' = tr) \wedge (wait' = true) \lhd true \rhd P_t[true/ok]^f$$

$$= \left\{ \text{Logic theory:- propositions} \right\}$$

$$false$$

□

Lemma 35 (CSP theory:- $\mathbf{R3}(P)_t[false/ok]^t$ simplification)**.**

$(\mathbf{R3}(P))_t[false/ok]^t = (tr \leq tr)$

Proof.

$$(\mathbf{R3}(P))_t[false/ok]^t$$

$$= \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok] \end{array} \right\}$$

$$\neg\, false \wedge (tr \leq tr) \vee true \wedge (tr' = tr) \wedge (wait' = true) \lhd true \rhd P_t[false/ok]^t$$

$$= \left\{ \text{Logic theory:- propositions} \right\}$$

$$(tr \leq tr) \vee (tr' = tr) \wedge (wait' = true)$$

$$= \left\{ \text{Logic theory:- propositions} \right\}$$

$$(tr \leq tr)$$

□

Lemma 36 (CSP theory:- $\mathbf{R3}(P)_t[false/ok]^f$ simplification)**.**

$(\mathbf{R3}(P))_t[false/ok]^f = (tr \leq tr)$

Proof.

$$(\mathbf{R3}(P))_t[false/ok]^f$$

$$= \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok] \end{array} \right\}$$

$$\neg\, false \wedge (tr \leq tr) \vee false \wedge (tr' = tr) \wedge (wait' = true) \lhd true \rhd P_t[false/ok]^f$$

$$= \left\{ \text{Logic theory:- propositions} \right\}$$

$$(tr \leq tr)$$

Funded by
the European Union

□

Lemma 37 (CSP theory:- $\mathbf{R3}(P)_f[true/ok]^t$ simplification)**.**

$(\mathbf{R3}(P))_f[true/ok]^t = P_f[true/ok]^t$

Proof.

$$
\begin{aligned}
& (\mathbf{R3}(P))_f[true/ok]^t \\
= & \quad \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok] \end{array} \right\} \\
& \neg\, true \wedge (tr \leq tr) \vee true \wedge (tr' = tr) \wedge (wait' = false) \lhd false \rhd P_f[true/ok]^t \\
= & \quad \left\{ \text{Logic theory:- propositions} \right\} \\
& P_f[true/ok]^t
\end{aligned}
$$

□

Lemma 38 (CSP theory:- $\mathbf{R3}(P)_f[true/ok]^f$ simplification)**.**

$(\mathbf{R3}(P))_f[true/ok]^f = P_f[true/ok]^f$

Proof.

$$
\begin{aligned}
& (\mathbf{R3}(P))_f[true/ok]^f \\
= & \quad \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok] \end{array} \right\} \\
& \neg\, true \wedge (tr \leq tr) \vee false \wedge (tr' = tr) \wedge (wait' = false) \lhd false \rhd P_f[true/ok]^f \\
= & \quad \left\{ \text{Logic theory:- propositions} \right\} \\
& P_f[true/ok]^f
\end{aligned}
$$

□

Lemma 39 (CSP theory:- $\mathbf{R3}(P)_f[false/ok]^t$ simplification)**.**

$(\mathbf{R3}(P))_f[false/ok]^t = P_f[false/ok]^t$

Proof.

$(\mathbf{R3}(P))_f[false/ok]^t$

Funded by the European Union

$$
\begin{aligned}
&= \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok] \end{array} \right\} \\
&\quad \neg\, false \wedge (tr \leq tr) \vee true \wedge (tr' = tr) \wedge (wait' = false) \lhd false \rhd P_f[false/ok]^t \\
&= \left\{ \text{Logic theory:- propositions} \right\} \\
&\quad P_f[false/ok]^t
\end{aligned}
$$

□

Lemma 40 (CSP theory:- $\mathbf{R3}(P)_f[false/ok]^f$ simplification)**.**

$(\mathbf{R3}(P))_f[false/ok]^f = P_f[false/ok]^f$

Proof.

$$
\begin{aligned}
&\quad (\mathbf{R3}(P))_f[false/ok]^f \\
&= \left\{ \begin{array}{l} \text{Lemma 32:- CSP theory:- } \mathbf{R3} \text{ Simplification:} \\ (\mathbf{R3}(P))^c_a[b/ok] = \neg\, b \wedge (tr \leq tr) \vee c \wedge (tr' = tr) \wedge (wait' = a) \lhd a \rhd P^c_a[b/ok] \end{array} \right\} \\
&\quad \neg\, false \wedge (tr \leq tr) \vee false \wedge (tr' = tr) \wedge (wait' = false) \lhd false \rhd P_f[false/ok]^f \\
&= \left\{ \text{Logic theory:- propositions} \right\} \\
&\quad P_f[false/ok]^f
\end{aligned}
$$

□

The next theorem establishes a key equivalence between refinement of reactive designs and refinement of their underlying designs. Specifically, it states that two reactive processes $\mathbf{R3}(P1 \models P2)$ and $\mathbf{R3}(Q1 \models Q2)$ satisfy the refinement relation if, and only if, the corresponding designs $P1 \models P2$ and $Q1 \models Q2$ do. This result is important because it allows us to reduce reasoning about reactive refinement to simpler relational reasoning in the design theory. It hinges on the fact that both reactive designs are **R3**-healthy, and by assumption also **R2**-healthy, so we can decompose and simplify their semantics using substitutions for termination and divergence cases. The case analysis in the proof breaks down the refinement condition across all possible combinations of ok and ok' (i.e., true and false), exploiting earlier simplification lemmas. In doing so, the theorem shows that the observable reactive behaviour of a process is fully determined by its design-level assumptions and commitments. This provides a powerful tool for verifying refinement between CSP-modeled neural networks within the UTP framework.

The proof proceeds by full case analysis over the observational conditions ($wait'$, ok, and ok'). since these are all boolean, there are $2^3 = 8$ cases: $(wait', ok, ok') \in$

Funded by the European Union

$\{(t,t,t),(t,t,f),(t,f,t),(t,f,f),(f,t,t),(f,t,f),(f,f,t),(f,f,f)\}$. We use the substitution expression $P_a[b/ok]^c$ for case (a,b,c) of P.The simplification lemmas established earlier ensure that all relevant cases reduce to standard forms, enabling the use of the design refinement theorem.

Theorem 9 (CSP theory:- **R2**-CSP refinement)**.**

$$(\mathbf{R3}(P1 \models P2) \sqsubseteq \mathbf{R3}(Q1 \models Q2)) = ((P1 \vdash P2) \sqsubseteq (Q1 \vdash Q2))$$

Proof.

$$\begin{aligned}
& \mathbf{R3}(P) \sqsubseteq \mathbf{R3}(Q) \\
= & \quad \{ \text{ case analysis } \} \\
& (\mathbf{R3}(P))_t[true/ok]^t \sqsubseteq (\mathbf{R3}(Q))_t[true/ok]^t \\
& \wedge (\mathbf{R3}(P))_t[true/ok]^f \sqsubseteq (\mathbf{R3}(Q))_t[true/ok]^f \\
& \wedge (\mathbf{R3}(P))_t[false/ok]^t \sqsubseteq (\mathbf{R3}(Q))_t[false/ok]^t \\
& \wedge (\mathbf{R3}(P))_t[false/ok]^f \sqsubseteq (\mathbf{R3}(Q))_t[false/ok]^f \\
& \wedge (\mathbf{R3}(P))_f[true/ok]^t \sqsubseteq (\mathbf{R3}(Q))_f[true/ok]^t \\
& \wedge (\mathbf{R3}(P))_f[true/ok]^f \sqsubseteq (\mathbf{R3}(Q))_f[true/ok]^f \\
& \wedge (\mathbf{R3}(P))_f[false/ok]^t \sqsubseteq (\mathbf{R3}(Q))_f[false/ok]^t \\
& \wedge (\mathbf{R3}(P))_f[false/ok]^f \sqsubseteq (\mathbf{R3}(Q))_f[false/ok]^f
\end{aligned}$$

Funded by
the European Union

$$
= \left\{
\begin{array}{l}
\text{Lemma 33: CSP theory:- } \mathbf{R3}(P)_t[true/ok]^t \text{ simplification:} \\
(\mathbf{R3}(P))_t[true/ok]^t = (tr' = tr) \wedge wait' \\
\text{Lemma 34: CSP theory:- } \mathbf{R3}(P)_t[true/ok]^f \text{ simplification:} \\
(\mathbf{R3}(P))_t[true/ok]^f = false \\
\text{Lemma 35: CSP theory:- } \mathbf{R3}(P)_t[false/ok]^t \text{ simplification:} \\
(\mathbf{R3}(P))_t[false/ok]^t = (tr \leq tr) \\
\text{Lemma 36: CSP theory:- } \mathbf{R3}(P)_t[false/ok]^f \text{ simplification:} \\
(\mathbf{R3}(P))_t[false/ok]^f = (tr \leq tr) \\
\text{Lemma 37: CSP theory:- } \mathbf{R3}(P)_f[true/ok]^t \text{ simplification:} \\
(\mathbf{R3}(P))_f[true/ok]^t = P_f[true/ok]^t \\
\text{Lemma 38: CSP theory:- } \mathbf{R3}(P)_f[true/ok]^f \text{ simplification:} \\
(\mathbf{R3}(P))_f[true/ok]^f = P_f[true/ok]^f \\
\text{Lemma 39: CSP theory:- } \mathbf{R3}(P)_f[false/ok]^t \text{ simplification:} \\
(\mathbf{R3}(P))_f[false/ok]^t = P_f[false/ok]^t \\
\text{Lemma 40: CSP theory:- } \mathbf{R3}(P)_f[false/ok]^f \text{ simplification:} \\
(\mathbf{R3}(P))_f[false/ok]^f = P_f[false/ok]^f
\end{array}
\right\}
$$

$$
\begin{array}{l}
(tr' = tr) \wedge wait' \sqsubseteq (tr' = tr) \wedge wait' \\
\wedge\ false \sqsubseteq false \\
\wedge\ (tr \leq tr) \sqsubseteq (tr \leq tr) \\
\wedge\ (tr \leq tr) \sqsubseteq (tr \leq tr) \\
\wedge\ P_f[true/ok]^t \sqsubseteq Q_f[true/ok]^t \\
\wedge\ P_f[true/ok]^f \sqsubseteq Q_f[true/ok]^f \\
\wedge\ P_f[false/ok]^t \sqsubseteq Q_f[false/ok]^t \\
\wedge\ P_f[false/ok]^f \sqsubseteq Q_f[false/ok]^f
\end{array}
$$

$$
= \left\{ \text{reflexivity of refinement} \right\}
$$

Funded by the European Union

$$
\begin{array}{ll}
 & true \\
 & \wedge\ true \\
 & \wedge\ true \\
 & \wedge\ true \\
 & \wedge\ P_f[true/ok]^t \sqsubseteq Q_f[true/ok]^t \\
 & \wedge\ P_f[true/ok]^f \sqsubseteq Q_f[true/ok]^f \\
 & \wedge\ P_f[false/ok]^t \sqsubseteq Q_f[false/ok]^t \\
 & \wedge\ P_f[false/ok]^f \sqsubseteq Q_f[false/ok]^f \\
= & \quad \{\ \text{propositional calculus}\ \} \\
 & P_f[true/ok]^t \sqsubseteq Q_f[true/ok]^t \\
 & \wedge\ P_f[true/ok]^f \sqsubseteq Q_f[true/ok]^f \\
 & \wedge\ P_f[false/ok]^t \sqsubseteq Q_f[false/ok]^t \\
 & \wedge\ P_f[false/ok]^f \sqsubseteq Q_f[false/ok]^f
\end{array}
$$

Taking $P1 \vdash P2$ for P and $Q1 \vdash Q2$ for Q, we have

$$
\begin{array}{ll}
 & (P1 \vdash P2)_f[true/ok]^t \sqsubseteq (Q1 \vdash Q2)_f[true/ok]^t \\
 & \wedge\ (P1 \vdash P2)_f[true/ok]^f \sqsubseteq (Q1 \vdash Q2)_f[true/ok]^f \\
 & \wedge\ (P1 \vdash P2)_f[false/ok]^t \sqsubseteq (Q1 \vdash Q2)_f[false/ok]^t \\
 & \wedge\ (P1 \vdash P2)_f[false/ok]^f \sqsubseteq (Q1 \vdash Q2)_f[false/ok]^f \\
= & \quad \left\{ \begin{array}{l} \text{Lemma 23: Design theory:- } P[false/ok]^t \text{ simplification: } (P \vdash Q)[false/ok]^t = true \\ \text{Lemma 24: Design theory:- } P[false/ok]^f \text{ simplification: } (P \vdash Q)[false/ok]^f = true \end{array} \right\} \\
 & (P1 \vdash P2)_f[true/ok]^t \sqsubseteq (Q1 \vdash Q2)_f[true/ok]^t \\
 & \wedge\ (P1 \vdash P2)_f[true/ok]^f \sqsubseteq (Q1 \vdash Q2)_f[true/ok]^f \\
 & \wedge\ true \sqsubseteq true \\
 & \wedge\ true \sqsubseteq true \\
= & \quad \{\ \text{reflexivity of refinement}\ \} \\
 & (P1 \vdash P2)_f[true/ok]^t \sqsubseteq (Q1 \vdash Q2)_f[true/ok]^t \\
 & \wedge\ (P1 \vdash P2)_f[true/ok]^f \sqsubseteq (Q1 \vdash Q2)_f[true/ok]^f
\end{array}
$$

Funded by
the European Union

$$= \quad \left\{ \text{Theorem 3.1.2: Hoare \& He p.77} \right\}$$
$$(P1_f \vdash P2_f) \sqsubseteq (Q1_f \vdash Q2_f)$$

□

This theorem characterises refinement between reactive designs in terms of refinement between their underlying pre- and postconditions. The key insight is that if two reactive processes are both **R2** and **R3**-healthy, then refinement at the level of reactive behaviours can be reduced to refinement of their designs. This reduction simplifies reasoning significantly: rather than reasoning directly about reactive observations such as traces or refusals, we can operate on the design components directly. The proof makes this precise by considering the substitution instances of the ok and ok' variables and applying simplification lemmas derived from the healthiness conditions. The result supports modular verification, as one can verify refinement of reactive processes by checking the corresponding design-level refinements.

This corollary expresses the monotonicity of the **R3** operator with respect to refinement. Specifically, it states that if one design P refines another Q, then applying the **R3** healthiness condition to both preserves that refinement order. Since refinement is preserved through the application of **R3**, this healthiness operator is monotonic with respect to the refinement ordering. This property supports compositional reasoning.

Corollary C.23 (CSP theory:- **R3**-monotonicity)**.**

$$(P \sqsubseteq Q) \Rightarrow (\mathbf{R3}(P) \sqsubseteq \mathbf{R3}(Q))$$

Proof. Directly from Theorem 9: CSP theory:- **R2**-CSP refinement:

$$(\mathbf{R3}(P1 \models P2) \sqsubseteq \mathbf{R3}(Q1 \models Q2)) = ((P1 \vdash P2) \sqsubseteq (Q1 \vdash Q2))$$

□

This property is important because it allows compositional reasoning in the UTP framework. It ensures that once refinement is established at the design level, lifting to reactive processes via **R3** preserves that relationship. Consequently, designers can verify and refine system components in terms of their logical assumptions and commitments, confident that reactive interpretations will maintain these guarantees.

Now we turn to CSP's internal choice operator. First, **R3** distributes over choice (in the jargon, **R3** is *disjunctive*). Internal (nondeterministic) choice is interpreted in UTP and CSP via disjunction. The **R3** healthiness condition respects this structure.

Lemma 41 (CSP theory:- **R3** disjunctive)**.**

$$\mathbf{R3}(P \sqcap Q) = \mathbf{R3}(P) \sqcap \mathbf{R3}(Q)$$

Funded by the European Union

Proof.

$$
\begin{array}{ll}
& \mathbf{R3}((P1 \vdash P2) \sqcap (Q1 \vdash Q2)) \\
= & \quad \{ \text{ relational semantics of internal choice } \} \\
& \mathbf{R3}((P1 \vdash P2) \vee (Q1 \vdash Q2)) \\
= & \quad \{ \text{ R3 disjunctivity } \} \\
& \mathbf{R3}(P1 \vdash P2) \vee \mathbf{R3}(Q1 \vdash Q2) \\
= & \quad \{ \text{ relational semantics of internal choice } \} \\
& \mathbf{R3}(P1 \vdash P2) \sqcap \mathbf{R3}(Q1 \vdash Q2)
\end{array}
$$

□

Next, we define internal choice as a reactive-design operator. As we said before, the operator is simply disjunction. Here, we elaborate on that to explain the composition of preconditions and postconditions.

Lemma 42 (CSP theory internal choice reactive design)**.**

$$\mathbf{R3}(P1 \models P2) \sqcap \mathbf{R3}(Q1 \models Q2) \;=\; \mathbf{R3}(P1 \wedge Q1 \models P2 \vee Q2)$$

Proof.

$$
\begin{array}{ll}
& \mathbf{R3}(P1 \models P2) \sqcap \mathbf{R3}(Q1 \models Q2) \\
= & \quad \{ \text{ Lemma 41: CSP theory:- } \mathbf{R3} \text{ disjunctive: } \mathbf{R3}(P \sqcap Q) = \mathbf{R3}(P) \sqcap \mathbf{R3}(Q) \} \\
& \mathbf{R3}((P1 \models P2) \sqcap (Q1 \models Q2)) \\
= & \quad \{ \text{ Definition C.13: Design theory:- } \mathbf{R1}\text{-design: } (P \models Q) = \mathbf{R1}(P \vdash Q) \} \\
& \mathbf{R3}((P1 \vdash \mathbf{R1}(P2)) \sqcap (Q1 \vdash \mathbf{R1}(Q2))) \\
= & \quad \{ \text{ Theorem 7: Design theory:- design choice: } (P1 \vdash P2) \sqcap (Q1 \vdash Q2) = (P1 \wedge Q1 \vdash P2 \vee Q2) \} \\
& \mathbf{R3}(P1 \wedge Q1 \vdash \mathbf{R1}(P2) \vee \mathbf{R1}(Q2)) \\
= & \quad \{ \text{ Lemma 29: Design theory:- } \mathbf{R1} \text{ disjunctive: } \mathbf{R1}(P \vee Q) = \mathbf{R1}(P) \vee \mathbf{R1}(Q) \} \\
& \mathbf{R3}(P1 \wedge Q1 \vdash \mathbf{R1}(P2 \vee Q2))
\end{array}
$$

Funded by
the European Union

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design:} \\ (P \models Q) = \mathbf{R1}(P \vdash Q) \end{array} \right\} \\
& \mathbf{R3}(P1 \wedge Q1 \models P2 \vee Q2)
\end{array}
$$

□

The next result is important. It reduces refinement of reactive designs to a familiar pair of proof obligations: weakening preconditions and strengthening postconditions.

Lemma 43 (CSP theory:- reactive design refinement)**.**

$$P \sqsubseteq Q = [\neg P_f^f \Rightarrow \neg Q_f^f] \wedge [\neg P_f^f \wedge \mathbf{R1}(Q_f^t) \Rightarrow \mathbf{R1}(P_f^t)]$$

Proof.

$$
\begin{array}{ll}
& P \sqsubseteq Q \\
= & \left\{ \begin{array}{l} \text{Theorem C.22: CSP theory:- } \mathbf{R3}\text{-}\mathbf{R1}\text{-design process:} \\ P \text{ is } \mathbf{R2}\text{-healthy } \Rightarrow P = \mathbf{R3}(\neg P_f^f \models P_f^t) \end{array} \right\} \\
& \mathbf{R3}(\neg P_f^f \models P_f^t) \sqsubseteq \mathbf{R3}(\neg Q_f^f \models Q_f^t) \\
\Leftarrow & \left\{ \begin{array}{l} \text{Theorem C.23: CSP theory:- } \mathbf{R3}\text{-monotonicity:} \\ (P \sqsubseteq Q) \Rightarrow (\mathbf{R3}(P) \sqsubseteq \mathbf{R3}(Q)) \end{array} \right\} \\
& (\neg P_f^f \models P_f^t) \sqsubseteq (\neg Q_f^f \models Q_f^t) \\
= & \left\{ \begin{array}{l} \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design:} \\ (P \models Q) = \mathbf{R1}(P \vdash Q) \end{array} \right\} \\
& (\neg P_f^f \vdash \mathbf{R1}(P_f^t)) \sqsubseteq (\neg Q_f^f \mid \mathbf{R1}(Q_f^t)) \\
= & \left\{ \begin{array}{l} \text{Theorem 6: Design theory:- design refinement:} \\ ((P1 \vdash P2) \sqsubseteq (Q1 \vdash Q2)) = [P1 \Rightarrow Q1] \wedge [P1 \wedge Q2 \Rightarrow P2] \end{array} \right\} \\
& [\neg P_f^f \Rightarrow \neg Q_f^f] \wedge [\neg P_f^f \wedge \mathbf{R1}(Q_f^t) \Rightarrow \mathbf{R1}(P_f^t)]
\end{array}
$$

□

Lemma 44 (CSP theory:- reactive design precondition)**.**

$$\neg\, (\mathbf{R3}(P \models Q))_f^f = ok \wedge P_f^f$$

Proof.

$$\neg\, (\mathbf{R3}(P \models Q))_f^f$$

Funded by the European Union

$$
\begin{array}{ll}
 & \neg\ (P \models Q)^{f}_{f} \\
= & \left\{ \begin{array}{l} \text{Lemma 31: CSP theory:- } \mathbf{R3} \text{ not waiting:} \\ (\mathbf{R3}(P))_f = P_f \end{array} \right\} \\
 & \neg\ (P \models Q)^{f}_{f} \\
= & \left\{ \begin{array}{l} \text{Lemma C.17: Design theory:- } \mathbf{R1}\text{-design-substitution:} \\ x \text{ isn't } \{ok, ok', tr, tr'\} \Rightarrow (P \models Q)[e/x] = (P[e/x] \models Q[e/x]) \end{array} \right\} \\
 & \neg\ (P_f \models Q_f)^{f} \\
= & \left\{ \begin{array}{l} \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design:} \\ (P \models Q) = \mathbf{R1}(P \vdash Q) \end{array} \right\} \\
 & \neg\ (P_f \vdash \mathbf{R1}(Q_f))^{f} \\
= & \left\{ \begin{array}{l} \text{Definition C.6: Design theory:- design:} \\ (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \end{array} \right\} \\
 & \neg\ (ok \wedge P_f \Rightarrow ok' \wedge \mathbf{R1}(Q_f))^{f} \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
 & \neg\ (ok \wedge P^{f}_{f} \Rightarrow \mathit{false} \wedge \mathbf{R1}(Q_f)^{f}) \\
= & \left\{ \text{Logic theory:- propositions:} \right\} \\
 & \neg\ \neg\ (ok \wedge P^{f}_{f}) \\
= & \left\{ \text{Logic theory:- propositions:} \right\} \\
 & ok \wedge P^{f}_{f}
\end{array}
$$

□

Lemma 45 (CSP theory:- reactive design postcondition).

$(\mathbf{R3}(P \models Q))^{t}_{f} = ok \wedge P^{t}_{f} \Rightarrow \mathbf{R1}(Q_f)^{t}$

Proof.

$$
\begin{array}{ll}
 & (\mathbf{R3}(P \models Q))^{t}_{f} \\
= & \left\{ \begin{array}{l} \text{Lemma 31: CSP theory:- } \mathbf{R3} \text{ not waiting:} \\ (\mathbf{R3}(P))_f = P_f \end{array} \right\} \\
 & (P \models Q)^{t}_{f} \\
= & \left\{ \begin{array}{l} \text{Lemma C.17: Design theory:- } \mathbf{R1}\text{-design-substitution:} \\ x \text{ isn't } \{ok, ok', tr, tr'\} \Rightarrow (P \models Q)[e/x] = (P[e/x] \models Q[e/x]) \end{array} \right\}
\end{array}
$$

Funded by
the European Union

$$
\begin{array}{ll}
 & (P_f \models Q_f)^t \\
= & \left\{ \begin{array}{l} \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design:} \\ (P \models Q) = \mathbf{R1}(P \vdash Q) \end{array} \right\} \\
 & (P_f \vdash \mathbf{R1}(Q_f))^t \\
= & \left\{ \begin{array}{l} \text{Definition C.6: Design theory:- design:} \\ (P \vdash Q) = ok \wedge P \Rightarrow ok' \wedge Q \end{array} \right\} \\
 & (ok \wedge P_f \Rightarrow ok' \wedge \mathbf{R1}(Q_f))^t \\
= & \left\{ \text{Logic theory:- substitution :} \right\} \\
 & ok \wedge P_f^t \Rightarrow true \wedge \mathbf{R1}(Q_f)^t \\
= & \left\{ \text{Logic theory:- propositions:} \right\} \\
 & ok \wedge P_f^t \Rightarrow \mathbf{R1}(Q_f)^t
\end{array}
$$

□

Lemma 46 (CSP theory:- reactive design refinement 2)**.** *for* **R2***-healthy* $P1$*,* $P2$*,* $Q1$*,* $Q2$

$$\mathbf{R3}(P1 \models P2) \sqsubseteq \mathbf{R3}(Q1 \models Q2) = [P1 \Rightarrow Q1] \wedge [P1 \wedge \mathbf{R1}(Q2) \Rightarrow P2]$$

Proof.

$$
\begin{array}{ll}
 & \mathbf{R3}(P1 \models P2) \sqsubseteq \mathbf{R3}(Q1 \models Q2) \\
= & \left\{ \begin{array}{l} \text{Theorem 9: CSP theory:- } \mathbf{R2}\text{-CSP refinement:} \\ (\mathbf{R3}(P1 \models P2) \sqsubseteq \mathbf{R3}(Q1 \models Q2)) = ((P1 \vdash P2) \sqsubseteq (Q1 \vdash Q2)) \end{array} \right\} \\
 & [\neg\, (\mathbf{R3}(P1 \vdash \mathbf{R1}(P2)))_f^f \Rightarrow \neg\, (\mathbf{R3}(Q1 \vdash \mathbf{R1}(Q2)))_f^f] \\
 & \wedge \\
 & [\neg\, (\mathbf{R3}(P1 \vdash \mathbf{R1}(P2)))_f^f \wedge \mathbf{R1}((\mathbf{R3}(Q1 \vdash \mathbf{R1}(Q2)))_f^t) \Rightarrow \mathbf{R1}((\mathbf{R3}(P1 \vdash \mathbf{R1}(P2)))_f^t)] \\
= & \left\{ \begin{array}{l} \text{Lemma 44: CSP theory:- reactive design precondition:} \\ \neg\, (\mathbf{R3}(P \models Q))_f^f = ok \wedge P_f^f \end{array} \right\} \\
 & [ok \wedge P1_f^f \Rightarrow ok \wedge Q1_f^f] \wedge [ok \wedge P1_f^f \wedge \mathbf{R1}((\mathbf{R3}(Q1 \vdash \mathbf{R1}(Q2)))_f^t) \Rightarrow \mathbf{R1}((\mathbf{R3}(P1 \vdash \mathbf{R1}(P2)))_f^t)] \\
= & \left\{ \begin{array}{l} \text{CSP theory:- reactive design postcondition :} \\ (\mathbf{R3}(P \models Q))_f^t = ok \wedge P_f^t \Rightarrow \mathbf{R1}(Q_f)^t \end{array} \right\} \\
 & [ok \wedge P1_f^f \Rightarrow ok \wedge Q1_f^f] \wedge [ok \wedge P1_f^f \wedge (ok \wedge Q1_f^t \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow (ok \wedge P1_f^t \Rightarrow \mathbf{R1}(P2_f)^t)]
\end{array}
$$

Funded by
the European Union

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{Law C.8: Design theory:- cases:} \\ [P] = [P^f] \wedge [P^t] \end{array} \right\} \\
& [(ok \wedge P1^f_f \Rightarrow ok \wedge Q1^f_f)[false/ok]] \\
& \wedge \\
& [(ok \wedge P1^f_f \Rightarrow ok \wedge Q1^f_f)[true/ok]] \\
& \wedge \\
& [(ok \wedge P1^f_f \wedge (ok \wedge Q1^t_f \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow (ok \wedge P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t))[fslse/ok]] \\
& \wedge \\
& [(ok \wedge P1^f_f \wedge (ok \wedge Q1^t_f \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow (ok \wedge P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t))[true/ok]] \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
& [false \wedge P1^f_f \Rightarrow false \wedge Q1^f_f] \\
& \wedge \\
& [true \wedge P1^f_f \Rightarrow true \wedge Q1^f_f] \\
& \wedge \\
& [false \wedge P1^f_f \wedge (false \wedge Q1^t_f \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow false \wedge P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t] \\
& \wedge \\
& [true \wedge P1^f_f \wedge (true \wedge Q1^t_f \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow true \wedge P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t] \\
= & \left\{ \text{Logic theory:- propositions} \right\} \\
& [true] \wedge [P1^f_f \Rightarrow Q1^f_f] \wedge [true] \wedge [P1^f_f \wedge (Q1^t_f \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t] \\
= & \left\{ \text{Logic theory:- predicates} \right\} \\
& [P1^f_f \Rightarrow Q1^f_f] \wedge [P1^f_f \wedge (Q1^t_f \Rightarrow \mathbf{R1}(Q2_f)^t) \Rightarrow P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t] \\
= & \left\{ \text{Logic theory:- propositions} \right\} \\
& [P1^f_f \Rightarrow Q1^f_f] \wedge [P1^f_f \wedge \mathbf{R1}(Q2_f)^t \Rightarrow P1^t_f \Rightarrow \mathbf{R1}(P2_f)^t] \\
= & \left\{ \begin{array}{l} \text{assumption:} \\ [P1^f \Rightarrow P1^t] \end{array} \right\} \\
& [P1^f_f \Rightarrow Q1^f_f] \wedge [P1^f_f \wedge \mathbf{R1}(Q2_f)^t \Rightarrow \mathbf{R1}(P2_f)^t]
\end{array}
$$

□

C.4 Approximations

In this section, we introduce a theory of approximate conformance for neural systems based on observational semantics. Unlike classical refinement, which requires

Funded by the European Union

exact correspondence of behaviour, approximate conformance tolerates bounded differences—an essential property for reasoning about real-valued systems such as neural networks, which exhibit numerical imprecision due to learning, noise, and rounding. Our approach is compositional and layered. We begin with an approximate relation on values, lift it to sequences, and then extend it to reactive processes. At each level, we show how approximation respects nondeterministic choice and preserves key semantic properties.

Example C.24 (Approximation in ANNs)**.** *Suppose a neural network produces the output trace* $[0.9, 0.1, 0.8]$*, and we choose a tolerance* $\varepsilon = 0.2$*. Then the pointwise approximation intervals are:*

$$\begin{aligned} \mathit{Step}1: &\quad \approx (0.2)(0.9) = \langle 0.7, 1.1\rangle \\ \mathit{Step}2: &\quad \approx (0.2)(0.1) = \langle -0.1, 0.3\rangle \\ \mathit{Step}3: &\quad \approx (0.2)(0.8) = \langle 0.6, 1.0\rangle \end{aligned}$$

Thus, the set of ε*-approximating traces includes all sequences of the form* $\langle x_1, x_2, x_3\rangle$ *where:*

$$x_1 \in \langle 0.7, 1.1\rangle, \quad x_2 \in \langle -0.1, 0.3\rangle, \quad x_3 \in \langle 0.6, 1.0\rangle$$

For example, the following traces conform within ε *to the original trace:*

$$\langle 0.85, 0.2, 0.95\rangle, \quad \langle 0.7, -0.05, 0.6\rangle, \quad \langle 1.0, 0.0, 0.8\rangle$$

These traces are all members of the set $\mathit{seq\_approx}(\varepsilon)(\langle 0.9, 0.1, 0.8\rangle)$*, and so would be permitted as observations in the process* $\mathit{Approx}(\varepsilon)(P)$*, where* P *is the original network.*

This illustrates how the simulation

$$\mathit{Approx}(\varepsilon)(P) = \textbf{var}\ t := tr \bullet P + t\ ;\ \mathit{upd\_tr}(\varepsilon)$$

captures bounded behavioural variation. The update $\mathit{upd\_tr}(\varepsilon)$ *allows any* tr' *such that the added suffix* $tr' - t$ *is an* ε*-approximation of the original suffix* $tr_0 - t$*, where* tr_0 *satisfies* P*.*

We start our formal account of conformance for ANNs by defining the simple idea of one value approximating another within a tolerance. The fundamental idea in our theory is that value approximation is defined using intervals. The binary relation $\mathit{approx}(\varepsilon)$ captures ε-bounded approximation between values.

Definition C.25 (Value Approximation)**.** *We approximate* x *within* ε *as*

$$\mathit{approx}(\varepsilon)(x) \ \hat{=}\ [x - \varepsilon, x + \varepsilon] \qquad \textit{provided}\ \varepsilon \geq 0$$

This function defines the atomic unit of deviation our system can tolerate. The proviso avoids an empty interval, which does not interest us. The function has some obvious properties. First, approximation defines a range of values:

Funded by
the European Union

Law C.26 (Value approximation: interval)**.**

$x \in approx(\varepsilon)(y) = y - \varepsilon \leq x \leq y + \varepsilon$

Every value approximates itself:

Law C.27 (Value self-approximation)**.**

$\varepsilon \geq 0 \Rightarrow x \in approx(\varepsilon)(x)$

Next, every value is a 0-approximation of itself:

Law C.28 (Value trivial approximation)**.**

$approx(0)(x) = \{x\}$

We prove just one of these laws: trivial value approximation.

$$
\begin{array}{ll}
 & approx(0)(x) \\
= & \quad \{ \text{ definition } approx(\varepsilon)(x) = [x-\varepsilon, x+\varepsilon] \} \\
 & [x-0, x+0] \\
= & \quad \{ \text{ arithmetic } \} \\
 & [x, x] \\
= & \quad \{ \text{ sets } \} \\
 & \{x\}
\end{array}
$$

A fifth property is so important that we prove it separately as a lemma: value approximation is monotonic in its tolerance.

Lemma 47 (Value Approximation Monotonicity)**.**

$\delta \leq \varepsilon \Rightarrow approx(\delta)(x) \subseteq approx(\varepsilon)(x)$

Proof.

$$
\begin{array}{ll}
 & approx(\delta)(x) \\
= & \quad \{ \text{ definition } approx(\varepsilon)(x) = [x-\varepsilon, x+\varepsilon] \} \\
 & [x-\delta, x+\delta] \\
\subseteq & \quad \{ \text{ assumption: } \delta \leq \varepsilon, \text{ sets } \} \\
 & [x-\varepsilon, x+\varepsilon] \\
= & \quad \{ \text{ definition } approx(\varepsilon)(x) = [x-\varepsilon, x+\varepsilon] \} \\
 & approx(\varepsilon)(x)
\end{array}
$$

□

Funded by
the European Union

Next, we extend value approximation to sequence approximation. Our motivation for this is to be able to describe how one process trace approximates another.

Definition C.29 (Sequence Pointwise Approximation)**.** *We lift value approximation pointwise to sequences:*

$$seq\_approx(\varepsilon)(xs) \;=\; \{ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i)))\}$$

Sequence approximation has some useful properties.

Law C.30 (Least element value approx)**.**

$$\varepsilon \geq 0 \Rightarrow seq\_approx(0)(xs) \subseteq seq\_approx(\varepsilon)(xs)$$

Law C.31 (Sequence approx monotonicity)**.**

$$\delta \leq \varepsilon \Rightarrow seq\_approx(\delta)(xs) \subseteq seq\_approx(\varepsilon)(xs)$$

Law C.32 (Symmetry)**.**

$$ys \in seq\_approx(\varepsilon)(xs) xs \in seq\_approx(\varepsilon)(ys)$$

We prove some of these facts. First, a generalisation of value self-approximation.

Lemma 48 (Sequences Self-Approximation)**.**

$$\varepsilon \geq 0 \Rightarrow xs \in seq\_approx(\varepsilon)(xs)$$

Proof.

$$\begin{array}{ll} & xs \in seq\_approx(\varepsilon)(xs) \\ = & \left\{ \begin{array}{l} \text{definition } seq\_approx(\varepsilon)(xs) \\ \quad = \{ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i)))\} \end{array} \right\} \\ & xs \in \{ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i)))\} \\ = & \left\{ \text{sets} \right\} \\ & (\#xs = \#xs) \land (\forall i : \mathrm{dom}\, xs \bullet xs(i) \in approx(\varepsilon)(xs(i))) \\ = & \left\{ \text{logic} \right\} \\ & \forall i : \mathrm{dom}\, xs \bullet xs(i) \in approx(\varepsilon)(xs(i)) \\ = & \left\{ \text{lemma: values self-approximate } \varepsilon \geq 0 \Rightarrow x \in approx(\varepsilon)(x) \right\} \\ & \forall i : \mathrm{dom}\, xs \bullet true \\ = & \left\{ \text{logic} \right\} \end{array}$$

Funded by the European Union

true

□

Next, we generalise the approximation of trivial value.

Lemma 49 (Trivial Sequence Approximation)**.**

$seq\_approx(0)(xs) = \{xs\}$

Proof.

$$
\begin{array}{ll}
& seq\_approx(0)(xs) \\
= & \left\{ \begin{array}{l} \text{definition } seq\_approx(\varepsilon)(xs) \\ \quad = \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \} \end{array} \right\} \\
& \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(0)(xs(i))) \\
= & \left\{ \text{lemma: trivial value approximation } approx(0)(x) = \{x\} \right\} \\
& \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in \{xs(i)\}) \\
= & \left\{ \text{sets: } y \in \{x\} = (y = x) \right\} \\
& \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet (ys(i) = xs(i))) \\
= & \left\{ \text{sequences} \right\} \\
& \{ ys \mid (ys = xs) \\
= & \left\{ \text{sets} \right\} \\
& \{xs\}
\end{array}
$$

□

Next, a generalisation of the symmetry of value approximation.

Lemma 50 (Symmetry of Sequence Approximation)**.**

$ys \in seq\_approx(\varepsilon)(xs) \;=\; xs \in seq\_approx(\varepsilon)(ys)$

Proof.

$$
\begin{array}{ll}
& ys \in seq\_approx(\varepsilon)(xs) \\
= & \left\{ \begin{array}{l} \text{lemma: } ys \in seq\_approx(\varepsilon)(xs) \\ \quad = (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \end{array} \right\}
\end{array}
$$

Funded by
the European Union

$$
\begin{array}{ll}
 & (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \\
= & \quad \{\ \text{logic}\ \} \\
 & (\#xs = \#ys) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \\
= & \quad \{\ \text{sequences}\ \} \\
 & (\#xs = \#ys) \land (\forall i : \mathrm{dom}\, xs \bullet ys(i) \in approx(\varepsilon)(xs(i))) \\
= & \quad \{\ \text{lemma: } x \in approx(\varepsilon)(ys) = y \in approx(\varepsilon)(x)\ \} \\
 & (\#xs = \#ys) \land (\forall i : \mathrm{dom}\, xs \bullet xs(i) \in approx(\varepsilon)(ys(i))) \\
= & \quad \left\{ \begin{array}{l} \text{lemma: } ys \in seq\_approx(\varepsilon)(xs) \\ \quad = (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \end{array} \right\} \\
 & xs \in seq\_approx(\varepsilon)(ys)
\end{array}
$$

□

Sequence approximation is monotonic in the tolerance.

Lemma 51 (Sequence Approximation Monotonicity)**.** $\delta \leq \varepsilon \Rightarrow seq\_approx(\delta)(xs) \subseteq seq\_approx(\varepsilon)(xs)$

$$
\begin{array}{ll}
 & seq\_approx(\delta)(xs) \\
= & \quad \left\{ \begin{array}{l} \text{definition } seq\_approx(\varepsilon)(xs) \\ \quad = \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \} \end{array} \right\} \\
 & \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\delta)(xs(i))) \} \\
\subseteq & \quad \left\{ \begin{array}{l} \text{lemma: value approx mono: } \delta \leq \varepsilon \Rightarrow approx(\delta)(x) \subseteq approx(\varepsilon)(x) \\ \text{sets} \end{array} \right\} \\
 & \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \} \\
= & \quad \left\{ \begin{array}{l} \text{definition } seq\_approx(\varepsilon)(xs) \\ \quad = \{ ys \mid (\#ys = \#xs) \land (\forall i : \mathrm{dom}\, ys \bullet ys(i) \in approx(\varepsilon)(xs(i))) \} \end{array} \right\} \\
 & seq\_approx(\varepsilon)(xs)
\end{array}
$$

A consequence of monotonicity in tolerance is that there is a minimum element in the set inclusion order.

Funded by
the European Union

Lemma 52 (Least Element)**.** $\varepsilon \geq 0 \Rightarrow seq\_approx(0)(xs) \subseteq seq\_approx(\varepsilon)(xs)$

$$
\begin{array}{ll}
 & seq\_approx(0)(xs) \subseteq seq\_approx(\varepsilon)(xs) \\
= & \quad \{\ \text{lemma: trivial sequence approximation: } seq\_approx(0)(xs) = \{xs\}\ \} \\
 & \{xs\} \subseteq seq\_approx(\varepsilon)(xs) \\
= & \quad \{\ \text{sets}\ \} \\
 & xs \in seq\_approx(\varepsilon)(xs) \\
= & \quad \{\ \text{lemma: sequences self-approx: } \varepsilon \geq 0 \Rightarrow xs \in seq\_approx(\varepsilon)(xs)\ \} \\
 & true
\end{array}
$$

We lifted value approximation to sequence approximation to account for approximate traces. Now, we lift sequence approximation to process approximation (sets of traces), leading to a definition of *conformance*. To show that Q conforms within ε to P, we generate all acceptable ε-approximate behaviours of P. We then require Q to have one or more of these approximate behaviours. The function $\mathrm{Approx}(\varepsilon)$ generates all the required behaviours.

Definition C.33 (Process Approximation)**.** $\mathrm{Approx}(\varepsilon)(P)$

$$\mathrm{Approx}(\varepsilon)(P) \ \hat{=}\ \textbf{var}\ t := tr \bullet P_{+t}\ ;\ upd\_tr(\varepsilon)$$

$$upd\_tr(\varepsilon) \ \hat{=}\ tr :\in \{\, s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \,\}$$

We call this definition a *simulation*. We take the terminology from UTP's *separating simulations* [HJ98b, Chap. 7], where Hoare and He use them in the definition of their parallel-by-merge operator. The definition starts with variable block **var** $t := tr \bullet \cdots$ introducing a new variable t (which must not be free in P). This variable records the value of tr before P is executed ($t := tr$). Next, we execute P with its list of programming variables augmented by t, unaffected by the execution of P: (P_{+t}). Finally, the relation $upd\_tr(\varepsilon)$ updates the trace. The auxiliary definition of $upd\_tr(\varepsilon)$ generates the set $\{\, s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \,\}$. Here, tr is the trace produced by P, and t is the value of tr recorded before P's execution. So $tr - t$ is the contribution to the trace made by the process P. The set comprehension generates all possible occurrences where this portion is replaced by an ε-approximation. Finally, tr is assigned the value of one of these approximations ($tr :\in \cdots$). In this way, $\mathrm{Approx}(\varepsilon)(P)$ describes all the acceptable approximations of P. (This use of the symbol $:\in$ for assignment from a set is Abrial-esque—blending syntactic economy with semantic precision.)

There are laws for this simulation. First, there are two laws for the trace update relation. The relation $upd\_tr(0)$ is simply the identity relation II. Next, upd_tr is antimonotonic in its tolerance. This law leads to the antimonotonicity of the Approx relation.

Funded by
the European Union

Law C.34 (Identity)**.**

$upd\_tr(0) = \mathit{II}$

Law C.35 (Antimonotonicity of *upd_tr*)**.**

$\delta \leq \varepsilon \Rightarrow (upd\_tr(\varepsilon) \sqsubseteq upd\_tr(\delta))$

Law C.36 (Antimonotonicity of Approx)**.**

$\delta \leq \varepsilon \Rightarrow (\mathrm{Approx}(\varepsilon)(P) \sqsubseteq \mathrm{Approx}(\delta)(P))$

Law C.37 (Local block)**.**

x *not free in* $e \Rightarrow ((\mathbf{var}\ x := e \bullet P) = \mathbf{R3}(P[e/x]))$

We take the law for a local block from the theory of reactive processes [HJ98b, Chap. 8]. The assignment in the process $\mathbf{var}\ x := e \bullet P$ uses a law similar to "leading assignment" in the refinement calculus [Mor94]. We substitute e for the initial value of x, and the result is $\mathbf{var}\ x \bullet P[e/x]$. Suppose we also know that x not free in e, then we can remove the declaration of x. This transformation is valid only if any preceding process has terminated. If it has not, the process $\mathbf{var}\ x := e \bullet P$ must behave as II. We cover this case by applying the healthiness condition **R3**. This derivation explains the law. We can use it to expand the algebraic definition of our simulation.

$$
\begin{array}{ll}
 & \mathbf{var}\ t := tr \bullet P_{+t}\ ;\ upd\_tr(\varepsilon) \\
= & \quad \{\ x \text{ not free in } e \Rightarrow ((\mathbf{var}\ x := e \bullet P) = \mathbf{R3}(P[e/x]))\ \} \\
 & \mathbf{R3}((P_{+t}\ ;\ upd\_tr(\varepsilon))[tr/t]) \\
= & \quad \{\ \text{alphabet extension, substitution}\ \} \\
 & \mathbf{R3}(P \wedge (t' = tr)\ ;\ upd\_tr(\varepsilon)) \\
= & \quad \{\ \text{definition}\ \} \\
 & \mathbf{R3}(P \wedge (t' = tr)\ ;\ tr :\in \{\, s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \,\}) \\
= & \quad \{\ \text{relations}\ \} \\
 & \mathbf{R3}(\exists\, tr_0, t_0 \bullet P[tr_0/tr'] \wedge (t_0 = tr) \wedge tr' \in \{\, s : seq\_approx(\varepsilon)(tr_0 - t_0) \bullet t_0 \frown s \,\}) \\
= & \quad \{\ \text{sets}\ \} \\
 & \mathbf{R3}(\exists\, tr_0, t_0, s : seq\_approx(\varepsilon)(tr_0 - t_0) \bullet P[tr_0/tr'] \wedge (t_0 = tr) \wedge (s = tr' - t_0)) \\
= & \quad \{\ \text{logic}\ \} \\
 & \mathbf{R3}(\exists\, tr_0 \bullet P[tr_0/tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(tr_0 - tr))
\end{array}
$$

Funded by
the European Union

The result is a non-algebraic definition for generating the approximations of P. We have already explained the need to apply the healthiness condition. The predicate within this records the final trace satisfying P, which is called tr_0. This trace has two parts: the initial trace tr and the part added by executing P, which is $tr_0 - tr$. It is these events that we approximate, not those events in tr.

We use the sequence difference operator in the simulation's algebraic and non-algebraic statements. The next lemma establishes that this is well-defined in the update relation.

Lemma 53 (Well-definedness of $upd\_tr(\varepsilon)$)**.**

$$
\begin{array}{ll}
 & t := tr \,;\, P_{+t} \\
= & \quad \{ \text{ relations } \} \\
 & (P_{+t})[tr/t] \\
= & \quad \{ \text{ relations } \} \\
 & (P \wedge (t' = t))[tr/t] \\
= & \quad \{ \text{ assumption } P \text{ is } \mathbf{R1} \} \\
 & (P \wedge tr \leq tr' \wedge (t' = t))[tr/t] \\
= & \quad \{ \text{ substitution } \} \\
 & P[tr/t] \wedge tr \leq tr' \wedge (t' = tr) \\
\Rightarrow & \quad \{ \text{ logic } \} \\
 & t' \leq tr'
\end{array}
$$

Notice that the update relation is not a process. It is not **R1**.

Lemma 54 (Update Isn't **R1**)**.**

$$
\begin{array}{ll}
 & upd\_tr(eps) \\
= & \quad \{ \text{ definition: } upd_{tr}(\varepsilon) = tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \} \\
 & \mathbf{R3}(tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \}) \\
= & \quad \{ \text{ definition } \mathbf{R3}(P) \mathrel{\hat{=}} \mathit{II} \lhd wait \rhd P \} \\
 & \mathit{II} \lhd wait \rhd tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \\
\Rightarrow & \quad \{ \text{ relations } \}
\end{array}
$$

Funded by the European Union

$$
\begin{array}{ll}
 & tr \leq tr' \lhd wait \rhd tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \\
= & \quad \{ \text{ sets } \} \\
 & tr \leq tr' \lhd wait \rhd \exists s : seq\_approx(\varepsilon)(tr - t) \bullet tr' = t \frown s \\
\Rightarrow & \quad \{ \text{ sequences } \} \\
 & tr \leq tr' \lhd wait \rhd t \leq tr'
\end{array}
$$

Neither is it **R2**.

Lemma 55 (Update Isn't **R2**)**.**

$$
\begin{array}{ll}
 & \mathbf{R2}(upd\_tr(\varepsilon)) \\
= & \quad \{ \text{ definition } upd\_tr(\varepsilon) \mathrel{\hat{=}} tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \} \\
 & \mathbf{R2}(\mathbf{R3}(tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \})) \\
= & \quad \{ \text{ reactive } \mathbf{R2}(\mathbf{R3}(P)) = \mathbf{R3}(\mathbf{R2}(P)) \} \\
 & \mathbf{R3}(\mathbf{R2}(tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \})) \\
= & \quad \{ \text{ definition } \mathbf{R2}(P) = P[\langle\rangle, tr' - tr / tr, tr'] \} \\
 & \mathbf{R3}((tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \})[\langle\rangle, tr' - tr / tr, tr']) \\
= & \quad \{ \text{ substitution } \} \\
 & \mathbf{R3}(tr' - tr \in \{ s : seq\_approx(\varepsilon)(\langle\rangle - t) \bullet t \frown s \}) \\
\neq & \quad \{ \text{ definition } upd\_tr(\varepsilon) \mathrel{\hat{=}} tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \} \\
 & upd\_tr(\varepsilon)
\end{array}
$$

We prove that $upd\_tr(0)$ is the relational identity.

Law C.38 (Identity update)**.** $upd\_tr(0) = \mathit{II}$

$$
\begin{array}{ll}
 & upd\_tr(0) \\
= & \quad \{ \text{ definition } upd_{tr}(\varepsilon) = tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \} \\
 & tr :\in \{ s : seq\_approx(0)(tr - t) \bullet t \frown s \} \\
= & \quad \{ \text{ lemma: } seq\_approx(0)(s) = \{s\} \}
\end{array}
$$

Funded by the European Union

$$
\begin{array}{ll}
 & \mathbf{R3}(tr :\in \{ s : \{tr - t\} \bullet t \frown s \}) \\
= & \quad \{ \text{ sets } \} \\
 & \mathbf{R3}(tr :\in \{t \frown (tr - t)\}) \\
= & \quad \{ \text{ sequences } \} \\
 & \mathbf{R3}(tr :\in \{tr\}) \\
= & \quad \{ \text{ relations } \} \\
 & \mathbf{R3}(tr := tr) \\
= & \quad \{ \text{ relations } \} \\
 & \mathbf{R3}(\Pi) \\
= & \quad \{ \text{ reactive } \} \\
 & \Pi
\end{array}
$$

Update is antimonotonic in its tolerance.

Lemma 56 (Antimonotonicity of *upd_tr*)**.**

$\delta \leq \varepsilon \Rightarrow (upd\_tr(\varepsilon) \sqsubseteq upd\_tr(\delta))$

Proof.

$$
\begin{array}{ll}
 & upd\_tr(\delta) \\
= & \quad \{ \text{ definition } upd_{tr} = tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \} \\
 & \mathbf{R3}(tr :\in \{ s : seq\_approx(\delta)(tr - t) \bullet t \frown s \}) \\
\Rightarrow & \quad \left\{ \begin{array}{l} \text{lemma: seq approx mono: } \delta \leq \varepsilon \Rightarrow seq\_approx(\delta)(xs) \subseteq seq\_approx(\varepsilon)(xs) \\ \text{sets} \end{array} \right\} \\
 & \mathbf{R3}(tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \}) \\
= & \quad \{ \text{ definition } upd_{tr} = tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \} \} \\
 & upd\_tr(\varepsilon)
\end{array}
$$

□

A consequence is that Approx is also antimonotonic in its tolerance.

Lemma 57 (Left Antimonotonicity of Approx)**.**

$\delta \leq \varepsilon \Rightarrow (\text{Approx}(\varepsilon)(P) \sqsubseteq \text{Approx}(\delta)(P))$

Proof. Directly from the antimonotonicity of *upd_tr*.

□

Process approximation is an **R**-closure. That is, if P is **R**-healthy, then so is the simulation $\text{Approx}(\varepsilon)(P)$.

Theorem 10 (Approx closure)**.** *An approximated process is a reactive process:*

$\text{Approx}(\varepsilon)(P)$ *is* **R***-healthy*

Proof. See the following reactive healthiness lemmas.

□

First, the approximated process is **R1**-healthy.

Lemma 58 (Approx **R1** closed)**.** Approx *is* **R1***:*

$\text{Approx}(\varepsilon)(\mathbf{R1}(P)) = \mathbf{R1}(\text{Approx}(\varepsilon)(\mathbf{R1}(P)))$

Proof.

$$
\begin{array}{ll}
 & \text{Approx}(\varepsilon)(\mathbf{R1}(P)) \\
= & \quad \{ \text{ definition } \text{Approx}(\varepsilon)(P) = \mathbf{var}\ t := tr \bullet P_{+t}\ ;\ upd\_tr(\varepsilon) \ \} \\
 & \mathbf{var}\ t := tr \bullet \mathbf{R1}(P)_{+t}\ ;\ upd\_tr(\varepsilon) \\
= & \quad \{ \text{ realtions } \} \\
 & \mathbf{var}\ t := tr \bullet \mathbf{R1}(P_{+t})\ ;\ \mathbf{R1}(upd\_tr(\varepsilon)) \\
= & \quad \{ \text{ relations lemma: update!!!!!: } upd\_tr(\varepsilon) = \mathbf{R1}(upd\_tr(\varepsilon)) \ \} \\
 & \mathbf{var}\ t := tr\ ;\ \mathbf{R1}(P_{+t})\ ;\ \mathbf{R1}(upd\_tr(\varepsilon))\ ;\ \mathbf{end}\ t \\
= & \quad \{ \text{ relations } \} \\
 & \mathbf{R1}(\mathbf{var}\ t := tr)\ ;\ \mathbf{R1}(P_{+t})\ ;\ \mathbf{R1}(upd\_tr(\varepsilon))\ ;\ \mathbf{R1}(\mathbf{end}\ t) \\
\Rightarrow & \quad \{ \text{ reactive } \} \\
 & \mathbf{R1}(\mathbf{R1}(\mathbf{var}\ t := tr)\ ;\ \mathbf{R1}(P_{+t})\ ;\ \mathbf{R1}(upd\_tr(\varepsilon))\ ;\ \mathbf{R1}(\mathbf{end}\ t)) \\
\Rightarrow & \quad \{ \text{ reactive } \} \\
 & \mathbf{R1}(\mathbf{true})
\end{array}
$$

Funded by
the European Union

□

Next, the approximated process is **R2**-healthy.

Lemma 59 (Approx **R2** closed)**.** Approx *is* **R2**

Proof.

$$
\begin{aligned}
& \mathbf{R2}(\mathrm{Approx}(\varepsilon)(P)) \\
= & \quad \left\{ \begin{array}{l} \text{lemma: alternative definition} \\ \mathrm{Approx}(\varepsilon) = \mathbf{R3}(\exists\, t \bullet P[\langle\rangle, t - tr/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr)) \end{array} \right\} \\
& \mathbf{R2}(\exists\, t \bullet P[\langle\rangle, t - tr/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr)) \\
= & \quad \left\{ \text{definition } \mathbf{R2}(P) = P[\langle\rangle, tr' - tr/tr, tr'] \right\} \\
& (\exists\, t \bullet P[\langle\rangle, t - tr/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr))[\langle\rangle, tr' - tr/tr, tr'] \\
= & \quad \left\{ \text{substitution} \right\} \\
& \exists\, t \bullet P[\langle\rangle, t - tr/tr, tr'][\langle\rangle, tr' - tr/tr, tr'] \wedge tr' - tr - \langle\rangle \in seq\_approx(\varepsilon)(t - \langle\rangle) \\
= & \quad \left\{ \text{substitution} \right\} \\
& \exists\, t \bullet P[\langle\rangle, t - \langle\rangle/tr, tr'] \wedge tr' - tr - \langle\rangle \in seq\_approx(\varepsilon)(t) \\
= & \quad \left\{ \text{sequences} \right\} \\
& \exists\, t \bullet P[\langle\rangle, t/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t) \\
= & \quad \left\{ \text{logic} \right\} \\
& \exists\, s, t \bullet (s = tr \frown t) \wedge P[\langle\rangle, t/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t) \\
= & \quad \left\{ \text{sequences} \right\} \\
& \exists\, s, t \bullet (t = s - tr) \wedge P[\langle\rangle, t/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t) \\
= & \quad \left\{ \text{logic} \right\} \\
& \exists\, s \bullet P[\langle\rangle, s - tr/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(s - tr) \\
= & \quad \left\{ \text{logic} \right\} \\
& \exists\, t \bullet P[\langle\rangle, t - tr/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr) \\
= & \quad \left\{ \begin{array}{l} \text{lemma: alternative definition} \\ \mathrm{Approx}(\varepsilon) = \mathbf{R3}(\exists\, t \bullet P[\langle\rangle, t - tr/tr, tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr)) \end{array} \right\}
\end{aligned}
$$

$$\text{Approx}(\varepsilon)(P)$$

□

Finally, the non-algebraic expression of the simulation clearly shows that Approx is **R3**-healthy.

We now prove a few lemmas about process approximation. First, the approximation $\text{Approx}(\varepsilon)(P)$ contains P itself. That is, processes self-approximate.

Lemma 60 (Processes Self-Approximate)**.**

$$\text{Approx}(\varepsilon)(P) \sqsubseteq P$$

Proof.

$$\begin{aligned}
& \text{Approx}(\varepsilon)(P) \\
= & \quad \{ \text{ definition } \text{Approx}(\varepsilon)(P) = \mathbf{var}\ t := tr \bullet P_{+t}\ ;\ upd_tr(\varepsilon) \ \} \\
& \mathbf{var}\ t := tr \bullet P_{+t}\ ;\ upd_tr(\varepsilon) \\
\sqsubseteq & \quad \{ \text{ lemma: } upd_tr(\varepsilon) \sqsubseteq upd_tr(0) \ \} \\
& \mathbf{var}\ t := tr \bullet P_{+t}\ ;\ upd_tr(0) \\
= & \quad \{ \text{ lemma: } upd_tr(0) = \mathit{II} \ \} \\
& \mathbf{var}\ t := tr \bullet P_{+t}\ ;\ \mathit{II} \\
= & \quad \{ \text{ relations } \} \\
& \mathbf{var}\ t := tr \bullet P_{+t} \\
= & \quad \{ \text{ relations } \} \\
& P
\end{aligned}$$

□

Next, process approximation is antimonotonic in its tolerance.

Lemma 61 (Approx Tolerance Antimonotonic)**.**

$$\delta \geq \varepsilon \wedge (P \sqsubseteq Q) \Rightarrow (\text{Approx}(\delta)(P) \sqsubseteq \text{Approx}(\varepsilon)(Q))$$

Proof.

$$\text{Approx}(\delta)(P)$$

$$
\begin{array}{ll}
= & \{\ \text{lemma: } \mathrm{Approx}(\varepsilon)(P) = \mathbf{R3}(\exists\, t \bullet P[t/tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr))\ \} \\
& \mathbf{R3}(\exists\, t \bullet P[t/tr'] \wedge tr' - tr \in seq\_approx(\delta)(t - tr)) \\
\sqsubseteq & \left\{ \begin{array}{l} \text{lemma: } \delta \geq \varepsilon \Rightarrow seq\_approx(\varepsilon)(s) \subseteq seq\_approx(\varepsilon)(s) \\ \text{monotonicity of } \mathbf{R3} \end{array} \right\} \\
& \mathbf{R3}(\exists\, t \bullet P[t/tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr)) \\
\sqsubseteq & \{\ \text{monotonicity of } \mathbf{R3}\ \} \\
& \mathbf{R3}(\exists\, t \bullet Q[t/tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr)) \\
= & \{\ \text{lemma: } \mathrm{Approx}(\varepsilon)(P) = \mathbf{R3}(\exists\, t \bullet P[t/tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr))\ \} \\
& \mathrm{Approx}(\varepsilon)(Q)
\end{array}
$$

□

Definition C.39 (Approx theory:- conformance)**.**

$Q\ conf(\varepsilon)\ P = Approx(\varepsilon)(P) \sqsubseteq Q$

We now have all the mathematical machinery to define conformance.

Definition C.40 (Conformance)**.**

$Q\ conf(\epsilon)\ P \mathrel{\hat{=}} \mathrm{Approx}(\epsilon)(P) \sqsubseteq Q$

This says that Q conforms to P, within the tolerance ε, iff Q is an ε-approximation of P. Note that conformance is a kind of preorder: it is reflexive and transitive in a certain sense. If x is a δ-approximation of y and y is a ε-approximation of z, then x is a (δ, ε)-approximation of z. This pseudo-transitivity expresses the accumulation of uncertainty over design steps. We prove the first result.

Lemma 62 (Conformance is reflexive)**.**

$P\ conf(\varepsilon)\ P$

Proof.

$$
\begin{array}{ll}
& P\ conf(\varepsilon)\ P \\
= & \{\ \text{definition } Q\ conf(\epsilon)\ P \mathrel{\hat{=}} \mathrm{Approx}(\epsilon)(P) \sqsubseteq Q\ \} \\
& \mathrm{Approx}(\varepsilon)(P) \sqsubseteq P \\
= & \{\ \text{law: } \mathrm{Approx}(\varepsilon)(P) \sqsubseteq P\ \} \\
& true
\end{array}
$$

Funded by the European Union

□

Transitivity is not quite so straightforward. If Q *conf*(ε) P and R *conf*(ε) Q, then R *conf*$(2 * \varepsilon)$ P: the tolerances accumulate as we move down the abstraction hierarchy. Conformance is transitive only in this sense. To prove this, we start with a lemma that proves that value approximation is transitive in the same sense.

Lemma 63 (Value approximation is pseudo-transitive)**.**

$$x \in \mathit{approx}(\delta)(y) \land y \in \mathit{approx}(\varepsilon)(z) \;\Rightarrow\; x \in \mathit{approx}(\delta + \varepsilon)(z)$$

Proof.

$$
\begin{array}{ll}
 & x \in \mathit{approx}(\delta)(y) \land y \in \mathit{approx}(\varepsilon)(z) \\
= & \quad \{\ \text{lemma: } x \in \mathit{approx}(\varepsilon)(y) = y - \varepsilon \le x \le y + \varepsilon\ \} \\
 & y - \delta \le x \le y + \delta \land z - \varepsilon \le y \le z + \varepsilon \\
\Rightarrow & \quad \{\ \text{arithmetic}\ \} \\
 & z - \delta - \varepsilon \le y - \delta \le x \land z + \delta + \varepsilon \ge y + \delta \ge x \\
\Rightarrow & \quad \{\ \text{arithmetic}\ \} \\
 & z - \delta - \varepsilon \le x \le z + \delta + \varepsilon \\
= & \quad \{\ \text{lemma: } x \in \mathit{approx}(\varepsilon)(y) = y - \varepsilon \le x \le y + \varepsilon\ \} \\
 & x \in \mathit{approx}(\delta + \varepsilon)(z)
\end{array}
$$

□

Now we lift this to sequences.

Lemma 64 (Sequence approximation is pseudo-transitive)**.**

$$\mathit{xs} \in \mathit{seq\_approx}(\delta)(\mathit{ys}) \land \mathit{ys} \in \mathit{seq\_approx}(\varepsilon)(\mathit{zs}) \;\Rightarrow\; \mathit{xs} \in \mathit{seq\_approx}(\delta + \varepsilon)(\mathit{zs})$$

Proof.

$$
\begin{array}{ll}
 & \mathit{xs} \in \mathit{seq\_approx}(\delta)(\mathit{ys}) \land \mathit{ys} \in \mathit{seq\_approx}(\varepsilon)(\mathit{zs}) \\
= & \quad \{\ \text{definition } \mathit{seq\_approx}(\varepsilon)(\mathit{xs}) \\
 & \qquad = \{\, \mathit{ys} \mid (\#\mathit{ys} = \#\mathit{xs}) \land (\forall i : \mathrm{dom}\,\mathit{ys} \bullet \mathit{ys}(i) \in \mathit{approx}(\varepsilon)(\mathit{xs}(i)))\,\}\ \} \\
 & (\#\mathit{xs} = \#\mathit{ys}) \land (\forall i : \mathrm{dom}\,\mathit{xs} \bullet \mathit{xs}(i) \in \mathit{approx}(\delta)(\mathit{ys}(i))) \land \\
 & (\#\mathit{ys} = \#\mathit{zs}) \land (\forall i : \mathrm{dom}\,\mathit{ys} \bullet \mathit{ys}(i) \in \mathit{approx}(\varepsilon)(\mathit{zs}(i)))
\end{array}
$$

Funded by
the European Union

$$
\begin{array}{ll}
= & \{\ \text{logic}\ \} \\
& (\#xs = \#ys) \land (\#ys = \#zs) \land \\
& (\forall i : \operatorname{dom} xs \bullet xs(i) \in approx(\delta)(ys(i)) \land ys(i) \in approx(\varepsilon)(zs(i))) \\
\Rightarrow & \{\ \text{lemma: } x \in approx(\delta)(y) \land y \in approx(\varepsilon)(z) \Rightarrow x \in approx(\delta+\varepsilon)(z)\ \} \\
& (\#xs = \#ys) \land (\#ys = \#zs) \land (\forall i : \operatorname{dom} xs \bullet xs(i) \in approx(\delta+\varepsilon)(zs(i))) \\
\Rightarrow & \{\ \text{logic}\ \} \\
& (\#xs = \#zs) \land (\forall i : \operatorname{dom} xs \bullet xs(i) \in approx(\delta+\varepsilon)(zs(i))) \\
= & \left\{ \begin{array}{l} \text{definition } seq\_approx(\varepsilon)(xs) \\ \quad = \{ys \mid (\#ys = \#xs) \land (\forall i : \operatorname{dom} ys \bullet ys(i) \in approx(\varepsilon)(xs(i)))\} \end{array} \right\} \\
& xs \in seq\_approx(\delta+\varepsilon)(zs)
\end{array}
$$

□

Next, we prove that process approximation is accumulative:

$$\text{Approx}(\delta)(\text{Approx}(\varepsilon)(P)) \;=\; \text{Approx}(\delta+\varepsilon)(P)$$

This law is a kind of pseudo-idempotence.

Lemma 65 (Process approximation is accumulative)**.**

$$
\begin{array}{ll}
& \text{Approx}(\delta)(\text{Approx}(\varepsilon)(P)) \\
= & \{\ \text{lemma: } \text{Approx}(\varepsilon)(P) = \mathbf{R3}(\exists\, t \bullet P[t/tr'] \land tr' - tr \in seq\_approx(\varepsilon)(t - tr))\ \} \\
& \mathbf{R3}(\exists\, t \bullet \text{Approx}(\varepsilon)(P)[t/tr'] \land tr' - tr \in seq\_approx(\delta)(t - tr)) \\
= & \{\ \text{lemma: } \text{Approx}(\varepsilon)(P) = \mathbf{R3}(\exists\, t \bullet P[t/tr'] \land tr' - tr \in seq\_approx(\varepsilon)(t - tr))\ \} \\
& \mathbf{R3}(\exists\, t \bullet (\mathbf{R3}(\exists\, t \bullet P[t/tr'] \\
& \qquad\qquad \land tr' - tr \in seq\_approx(\varepsilon)(t - tr)))[t/tr'] \\
& \qquad\qquad \land tr' - tr \in seq\_approx(\delta)(t - tr)) \\
= & \{\ \text{reactive processes: } \mathbf{R3}(\mathbf{R3}(P)) = P\ \} \\
& \mathbf{R3}(\exists\, t \bullet (\exists\, t \bullet P[t/tr'] \\
& \qquad\qquad \land tr' - tr \in seq\_approx(\varepsilon)(t - tr))[t/tr'] \\
& \qquad\qquad \land tr' - tr \in seq\_approx(\delta)(t - tr)) \\
= & \{\ \text{logic}\ \}
\end{array}
$$

Funded by
the European Union

$$
\begin{array}{l}
\quad \mathbf{R3}(\exists\, t, u \bullet P[u/tr'] \\
\qquad\qquad \wedge\ t - tr \in seq\_approx(\varepsilon)(u - tr) \\
\qquad\qquad \wedge\ tr' - tr \in seq\_approx(\delta)(t - tr)) \\
\Rightarrow \quad \left\{ \begin{array}{l} \text{lemma: } xs \in seq\_approx(\delta)(ys) \wedge ys \in seq\_approx(\varepsilon)(zs) \\ \quad \Rightarrow xs \in seq\_approx(\delta + \varepsilon)(zs) \end{array} \right\} \\
\quad \mathbf{R3}(\exists\, t, u \bullet P[u/tr'] \wedge tr' - tr \in seq\_approx(\delta + \varepsilon)(u - tr)) \\
= \quad \left\{ \text{lemma: } \mathrm{Approx}(\varepsilon)(P) = \mathbf{R3}(\exists\, t \bullet P[t/tr'] \wedge tr' - tr \in seq\_approx(\varepsilon)(t - tr)) \right\} \\
\quad \mathrm{Approx}(\delta + \varepsilon)(P)
\end{array}
$$

Now, we can prove that conformance is pseudo-transitive.

Lemma 66 (Conformance is pseudo-transitive)**.**

$$R\ conf(\varepsilon)\ Q \wedge Q\ conf(\delta)\ P \Rightarrow R\ conf(\delta + \varepsilon)\ P$$

Proof.

$$
\begin{array}{ll}
 & (Q\ conf(\delta)\ P) \wedge (R\ conf(\varepsilon)\ Q) \\
= & \quad \left\{ \text{definition } Q\ conf(\varepsilon)\ P = \mathrm{Approx}(\varepsilon)(P) \sqsubseteq Q \right\} \\
 & (\mathrm{Approx}(\delta)(P) \sqsubseteq Q) \wedge (\mathrm{Approx}(\varepsilon)(Q) \sqsubseteq R) \\
\Rightarrow & \quad \left\{ \text{lemma: right-monotonicity of Approx} \right\} \\
 & (\mathrm{Approx}(\varepsilon)(\mathrm{Approx}(\delta)(P)) \sqsubseteq \mathrm{Approx}(\varepsilon)(Q)) \wedge (\mathrm{Approx}(\varepsilon)(Q) \sqsubseteq R) \\
\Rightarrow & \quad \left\{ \text{transitivity of refinement} \right\} \\
 & \mathrm{Approx}(\varepsilon)(\mathrm{Approx}(\delta)(P)) \sqsubseteq R \\
\Rightarrow & \quad \left\{ \text{lemma: } \mathrm{Approx}(\varepsilon)(\mathrm{Approx}(\delta)(P)) = \mathrm{Approx}(\delta + \varepsilon)(P) \right\} \\
 & \mathrm{Approx}(\delta + \varepsilon)(P) \sqsubseteq R \\
= & \quad \left\{ \text{definition } Q\ conf(\varepsilon)\ P = \mathrm{Approx}(\varepsilon)(P) \sqsubseteq Q \right\} \\
 & R\ conf(\delta + \varepsilon)\ P
\end{array}
$$

□

The approximation of a reactive process is itself a reactive process, so we can calculate its precondition and postcondition. The next two lemmas does just that.

Funded by
the European Union

Lemma 67 (Approx theory:- precondition 1)**.**

$$\neg\,(\mathit{Approx}(\varepsilon)(\mathbf{R3}(P1 \models P2)))^f_f = \mathit{ok} \wedge \forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']$$

Lemma 68 (Approx theory:- postcondition)**.**

$$\begin{aligned}&(\mathit{Approx}(\varepsilon)(\mathbf{R3}(P1 \models P2)))^t_f\\ &\quad = \mathit{ok} \wedge (\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'])\\ &\qquad \Rightarrow (\exists\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'])\end{aligned}$$

We can put these two results together to expand an approximated reactive design.

Theorem 11 (Approx theory:- approx R3-R1-design expand)**.**

$$\mathit{Approx}(\varepsilon)(\mathbf{R3}(P1 \models P2)) = \mathbf{R3}(\exists\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet (P1 \models P2)[tr \frown tr_0/tr'])$$

If x approxmiates y within ε, then y also approxmiates x within ε. xxxxx

Lemma 69 (Approx theory:- *approx*-symmetry)**.**

$$x \in \mathit{approx}(\varepsilon)(y) = y \in \mathit{approx}(\varepsilon)(x)$$

We give the formal definition of *upd_tr*, which requires that the free variable t is a prefix of the trace tr for $tr - t$ to be well defined.

Definition C.41 (Approx theory:- upd_tr)**.**

$$(t \leq tr) \Rightarrow \mathit{upd\_tr}(\varepsilon) = tr :\in \{\, s : \mathit{seq\_approx}(\varepsilon)(tr - t) \bullet t \frown s \,\}$$

Despite the change that *upd_tr* makes, *Approx* is still **R1**-healthy.

Lemma 70 (Approx theory:- Approx is R1)**.**

$$\mathit{Approx}(\varepsilon)(P) = \mathbf{R1}(\mathit{Approx}(\varepsilon)(P))$$

Now we have the formal definition of *Approx*.

Definition C.42 (Approx theory:- Approx)**.**

$$\mathit{Approx}(\varepsilon)(P) = (\mathbf{var}\ t := tr \bullet P_{+t}\ ;\ \mathit{updtr}(\varepsilon))$$

Here are the properties.

Law C.43. *Approx theory:- approx interval*

$$x \in \mathit{approx}(\varepsilon)(y) = y - \varepsilon \leq x \leq y + \varepsilon$$

Funded by
the European Union

Definition C.44 (Approx theory:- value approximation)**.**

$$y \in approx(\varepsilon)(y) = x - \varepsilon \leq y \leq x + \varepsilon$$

Lemma 71 (Approx theory value approx symmetry)**.**

$$x \in approx(\varepsilon)(y) = y \in approx(\varepsilon)(x)$$

Proof.

$$\begin{aligned}
& x \in approx(\varepsilon)(y) \\
= & \quad \left\{ \begin{array}{l} \text{Law C.43: Approx theory:- approx interval:} \\ x \in approx(\varepsilon)(y) = y - \varepsilon \leq x \leq y + \varepsilon \end{array} \right\} \\
& y - \varepsilon \leq x \leq y + \varepsilon \\
= & \quad \left\{ \begin{array}{l} \text{Arithmetic theory:- chained inequality:} \\ (a \leq b \leq c) = (a \leq b) \wedge (b \leq c) \end{array} \right\} \\
& y - \varepsilon \leq x \wedge x \leq y + \varepsilon \\
= & \quad \left\{ \begin{array}{l} \text{Arithmetic theory:- additive invariance:} \\ (a - c \leq b) = (a \leq b + c) \end{array} \right\} \\
& y \leq x + \varepsilon \wedge x - \varepsilon \leq y \\
= & \quad \left\{ \begin{array}{l} \text{Arithmetic theory:- chained inequality:} \\ (a \leq b \leq c) = (a \leq b) \wedge (b \leq c) \end{array} \right\} \\
& x - \varepsilon \leq y \leq x + \varepsilon \\
= & \quad \left\{ \begin{array}{l} \text{Definition C.44: Approx theory:- value approximation:} \\ y \in approx(\varepsilon)(y) = x - \varepsilon \leq y \leq x + \varepsilon \end{array} \right\} \\
& y \in approx(\varepsilon)(x)
\end{aligned}$$

□

The following law is need to prove that *Approx* is well defined.

Law C.45 (Relation theory:- initialised alphabet-extension)**.** *For* **R1***-healthy P:*

$$(t := tr \,;\, P_{+t}) \Rightarrow (t' \leq tr')$$

Lemma 72 (Approx theory:- well-definedness of *upd_tr*)**.**

$$(t := tr \,;\, P_{+t}) \;\Rightarrow\; (t' \leq tr')$$

Funded by
the European Union

Proof.

$$
\begin{array}{ll}
& t := tr \,;\, P_{+t} \\
= & \left\{ \begin{array}{l} \text{assumption:} \\ P \text{ is } R1 \end{array} \right\} \\
& t := tr \,;\, (\mathbf{R1}(P))_{+t} \\
= & \left\{ \begin{array}{l} \text{Definition C.12: Design theory:- } \mathbf{R1}\text{:} \\ \mathbf{R1}(P) = P \wedge (tr \leq tr') \end{array} \right\} \\
& t := tr \,;\, (P \wedge (tr \leq tr'))_{+t} \\
= & \left\{ \begin{array}{l} \text{Definition C.5: Relation theory:- alphabet extension:} \\ P_{+x} = P \wedge (x' = x) \end{array} \right\} \\
& t := tr \,;\, P \wedge (tr \leq tr') \wedge (t' = t) \\
= & \left\{ \begin{array}{l} \text{Law C.4: Relation theory:- leading assignment:} \\ x := e \,;\, P = P[e/x] \end{array} \right\} \\
& (P \wedge (tr \leq tr') \wedge (t' = t))[tr/t] \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
& (P \wedge (tr \leq tr'))[tr/t] \wedge (t' = t)[tr/t] \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
= & \left\{ \begin{array}{l} \text{assumption:} \\ t \text{ not free in } P \end{array} \right\} \\
& P \wedge (tr \leq tr') \wedge (t' = t)[tr/t] \\
= & \left\{ \text{Logic theory:- substitution} \right\} \\
& P[tr/t] \wedge (tr \leq tr') \wedge (t' = tr) \\
= & \left\{ \text{Logic theory:- Leibniz:} \right\} \\
& P \wedge (tr \leq tr') \wedge (t' = tr) \\
\Rightarrow & \left\{ \text{Logic theory:- propositions:} \right\} \\
& (t' \leq tr')
\end{array}
$$

□

Funded by the European Union

Lemma 73 (Expand Approx)**.**

$$Approx(\varepsilon)(P) = (\exists\, tr_0 \bullet P[tr \frown tr_0, tr/tr', t'] \land (tr \leq tr') \land tr_0 \in seq\_approx(\varepsilon)(tr' - tr))$$

Proof.

$$
\begin{array}{ll}
& Approx(\varepsilon)(P) \\
= & \left\{ \begin{array}{l} \text{Lemma 70: Approx theory:- Approx is R1:} \\ Approx(\varepsilon)(P) = \mathbf{R1}(Approx(\varepsilon)(P)) \end{array} \right\} \\
& \mathbf{R1}(Approx(\varepsilon)(P)) \\
= & \left\{ \begin{array}{l} \text{Definition C.12: Design theory:- } \mathbf{R1}\text{:} \\ \mathbf{R1}(P) = P \land (tr \leq tr') \end{array} \right\} \\
& Approx(\varepsilon)(P) \land (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Definition C.42: Approx theory:- Approx:} \\ Approx(\varepsilon)(P) = (\mathbf{var}\ t := tr \bullet P_{+t}\ ;\ updtr(\varepsilon)) \end{array} \right\} \\
& (vart := tr \bullet P_{+t}\ ;\ updtr(\varepsilon)) \land (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Law C.45: Relation theory:- initialised alphabet-extension:} \\ (t := tr\ ;\ P_{+t}) \Rightarrow (t' \leq tr') \end{array} \right\} \\
& (vart := tr \bullet P_{+t} \land (t' \leq tr')\ ;\ updtr(\varepsilon)) \land (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Law C.45: Relation theory:- initialised alphabet-extension:} \\ (t := tr\ ;\ P_{+t}) \Rightarrow (t' \leq tr') \end{array} \right\} \\
& (vart := tr \bullet P_{+t}\ ;\ (t \leq tr) \land updtr(\varepsilon)) \land (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Definition: C.3: Relation theory:- initialised block:} \\ x \text{ not free in } e \Rightarrow ((\mathbf{var}\ x := e \bullet P) = (\exists x, x' \bullet P[e/x])) \end{array} \right\} \\
& (\exists\, t, t' \bullet ((t = tr) \land P_{+t}\ ;\ (t \leq tr) \land upd\_tr(\varepsilon))) \land (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Definition C.2: Relation theory:- sequence:} \\ P\ ;\ Q = \exists x_0 \bullet P[x_0/x'] \land Q[x_0/x] \end{array} \right\} \\
& (\exists\, t, t', tr_0, t_0 \bullet ((t = tr) \land P_{+t})[tr_0, t_0/tr', t'] \land ((t \leq tr) \land upd\_tr(\varepsilon))[tr_0, t_0/tr, t]) \land (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Definition C.5: Relation theory:- alphabet extension:} \\ P_{+x} = P \land (x' = x) \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
(\exists\, t, t', tr_0, t_0 \bullet ((t = tr) \\
\qquad \wedge\ P \\
\qquad \wedge\ (t' = t))[tr_0, t_0/tr', t'] \\
\qquad \wedge\ ((t \leq tr) \\
\qquad \wedge\ upd\_tr(\varepsilon))[tr_0, t_0/tr, t]) \\
\qquad \wedge\ (tr \leq tr')
\end{array}
$$

$=$ { Logic theory:- substitution : }

$$
\begin{array}{l}
(\exists\, t, t', tr_0, t_0 \bullet (t = tr) \\
\qquad \wedge\ P[tr_0, t_0/tr', t'] \\
\qquad \wedge\ (t_0 = t) \\
\qquad \wedge\ ((t \leq tr) \\
\qquad \wedge\ upd\_tr(\varepsilon))[tr_0, t_0/tr, t]) \\
\qquad \wedge\ (tr \leq tr')
\end{array}
$$

$=$ { Logic theory:- predicates }

$$(\exists\, t', tr_0, t_0 \bullet P[tr_0, t_0/tr', t'] \wedge (t_0 = tr) \wedge ((t \leq tr) \wedge upd\_tr(\varepsilon))[tr_0, t_0/tr, t]) \wedge (tr \leq tr')$$

$=$ { Logic theory:- predicates }

$$(\exists\, tr_0, t_0 \bullet P[tr_0, t_0/tr', t'] \wedge (t_0 = tr) \wedge ((t \leq tr) \wedge upd\_tr(\varepsilon))[tr_0, t_0/tr, t]) \wedge (tr \leq tr')$$

$=$ { Logic theory:- predicates }

$$(\exists\, tr_0 \bullet P[tr_0, tr/tr', t'] \wedge ((t \leq tr) \wedge upd\_tr(\varepsilon))[tr_0, tr/tr, t]) \wedge (tr \leq tr')$$

$=$ { Definition C.41: Approx theory:- upd_tr:
$(t \leq tr) \Rightarrow upd\_tr(\varepsilon) = tr :\in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \}$ }

$$(\exists\, tr_0 \bullet P[tr_0, tr/tr', t'] \wedge (tr' \in \{ s : seq\_approx(\varepsilon)(tr - t) \bullet t \frown s \})[tr_0, tr/tr, t]) \wedge (tr \leq tr')$$

$=$ { Logic theory:- substitution : }

$$(\exists\, tr_0 \bullet P[tr_0, tr/tr', t'] \wedge tr' \in \{ s : seq\_approx(\varepsilon)(tr_0 - tr) \bullet tr \frown s \}) \wedge (tr \leq tr')$$

$=$ { Set theory:- comprehension:
$x \in \{ y : T \bullet f(y) \} = \exists y : T \bullet (x = f(y))$ }

$$(\exists\, tr_0 \bullet P[tr_0, tr/tr', t'] \wedge \exists s : seq\_approx(\varepsilon)(tr_0 - tr) \bullet (tr' = tr \frown s)) \wedge (tr \leq tr')$$

Funded by
the European Union

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{Sequence theory:- difference:} \\ (u \leq s) \Rightarrow (s = t \frown u) = (t = s - u) \end{array} \right\} \\
& (\exists\, tr_0 \bullet P[tr_0, tr/tr', t'] \wedge \exists s : seq\_approx(\varepsilon)(tr_0 - tr) \bullet (s = tr' - tr)) \wedge (tr \leq tr') \\
= & \left\{ \text{Logic theory:- predicates} \right\} \\
& (\exists\, tr_0 \bullet P[tr_0, tr/tr', t'] \wedge tr' - tr \in seq\_approx(\varepsilon)(tr_0 - tr)) \wedge (tr \leq tr') \\
= & \left\{ \text{Logic theory:- predicates} \right\} \\
& (\exists\, tr_0, tr_1 \bullet (tr_1 = tr_0 - tr) \wedge P[tr_0, tr/tr', t'] \wedge tr' - tr \in seq\_approx(\varepsilon)(tr_1)) \wedge (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Sequence theory:- difference:} \\ (u \leq s) \Rightarrow (s = t \frown u) = (t = s - u) \end{array} \right\} \\
& (\exists\, tr_0, tr_1 \bullet (tr_0 = tr \frown tr_1) \wedge P[tr_0, tr/tr', t'] \wedge tr' - tr \in seq\_approx(\varepsilon)(tr_1)) \wedge (tr \leq tr') \\
= & \left\{ \text{Logic theory:- predicates} \right\} \\
& (\exists\, tr_1 \bullet P[tr \frown tr_1, tr/tr', t'] \wedge tr' - tr \in seq\_approx(\varepsilon)(tr_1)) \wedge (tr \leq tr') \\
= & \left\{ \text{Logic theory:- predicates} \right\} \\
& (\exists\, tr_0 \bullet P[tr \frown tr_0, tr/tr', t'] \wedge tr' - tr \in seq\_approx(\varepsilon)(tr_0)) \wedge (tr \leq tr') \\
= & \left\{ \begin{array}{l} \text{Lemma 69: Approx theory:- } approx\text{-symmetry:} \\ x \in approx(\varepsilon)(y) = y \in approx(\varepsilon)(x) \end{array} \right\} \\
& (\exists\, tr_0 \bullet P[tr \frown tr_0, tr/tr', t'] \wedge tr_0 \in seq\_approx(\varepsilon)(tr' - tr)) \wedge (tr \leq tr') \\
= & \left\{ \text{Logic theory:- propositions:} \right\} \\
& (\exists\, tr_0 \bullet P[tr \frown tr_0, tr/tr', t'] \wedge (tr \leq tr') \wedge tr_0 \in seq\_approx(\varepsilon)(tr' - tr))
\end{array}
$$

□

Corollary C.46 (Approx theory:- Approx is a reactive design)**.**

$Approx(\varepsilon)(P) = \mathbf{R3}(\neg\, (Approx(\varepsilon)(P))^f_f \models (Approx(\varepsilon)(P))^t_f)$

Proof. $Approx(\varepsilon)(P)$ is a CSP process. □

Funded by the European Union

Lemma 74 (Approx theory:- reactive design not waiting)**.**

$$
\begin{aligned}
&Approx(\varepsilon)(\mathbf{R3}(P1 \models P2))_f \\
&\quad = \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\
&\qquad \vdash \\
&\qquad \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&(Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))_f \\
&= \left\{ \begin{array}{l} \text{Theorem 11: Approx theory:- approx R3-R1-design expand:} \\ Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)) = \mathbf{R3}(\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \models P2)[tr \frown tr_0/tr']) \end{array} \right\} \\
&(\mathbf{R3}(\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (\mathbf{R3}(P1 \models P2))[tr \frown tr_0/tr']))_f \\
&= \left\{ \begin{array}{l} \text{Lemma 31: CSP theory:- } \mathbf{R3} \text{ not waiting:} \\ (\mathbf{R3}(P))_f = P_f \end{array} \right\} \\
&(\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (\mathbf{R3}(P1 \models P2))[tr \frown tr_0/tr'])_f \\
&= \left\{ \text{Logic theory:- substitution} \right\} \\
&\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet ((\mathbf{R3}(P1 \models P2))[tr \frown tr_0/tr'])_f \\
&= \left\{ \text{Logic theory:- substitution} \right\} \\
&\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (\mathbf{R3}(P1 \models P2))^t_f[tr \frown tr_0/tr'] \\
&= \left\{ \begin{array}{l} \text{Lemma 31: CSP theory:- } \mathbf{R3} \text{ not waiting:} \\ (\mathbf{R3}(P))_f = P_f \end{array} \right\} \\
&\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \models P2)^t_f[tr \frown tr_0/tr'] \\
&= \left\{ \begin{array}{l} \text{Lemma 30: Design theory:- } \mathbf{R1} \text{ design not waiting:} \\ (P \models Q)_f = (P_f \models Q_f) \end{array} \right\} \\
&\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1_f \models P2_f)[tr \frown tr_0/tr'] \\
&= \left\{ \begin{array}{l} \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design:} \\ (P \models Q) = \mathbf{R1}(P \vdash Q) \end{array} \right\} \\
&\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1_f \vdash \mathbf{R1}(P2_f))[tr \frown tr_0/tr']
\end{aligned}
$$

Funded by the European Union

$$
\begin{array}{ll}
 & \left\{ \begin{array}{l} \text{Definition C.12: Design theory:- } \mathbf{R1}\text{:} \\ \mathbf{R1}(P) = P \wedge (tr \leq tr') \end{array} \right\} \\
= & \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1_f \vdash P2_f \wedge (tr \leq tr'))[tr \frown tr_0/tr'] \\
 & \left\{ \begin{array}{l} \text{Lemma 28: Design theory:- design-substitution:} \\ x \text{ isn't } \{ok, ok'\} \Rightarrow (P \vdash Q)[e/x] = (P[e/x] \vdash Q[e/x]) \end{array} \right\} \\
= & \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1^t_f[tr \frown tr_0/tr'] \vdash P2^t_f[tr \frown tr_0/tr'] \wedge (tr \leq tr \frown tr_0)) \\
 & \left\{ \begin{array}{l} \text{Sequence theory:- monotonicity:} \\ s \leq s \frown t \end{array} \right\} \\
= & \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1^t_f[tr \frown tr_0/tr'] \vdash P2^t_f[tr \frown tr_0/tr']) \\
 & \left\{ \begin{array}{l} \text{Law C.11: Design theory:- existential design:} \\ (\exists x \bullet (P \vdash Q)) = (\forall x \bullet P \vdash \exists x \bullet Q) \end{array} \right\} \\
= & \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1^t_f[tr \frown tr_0/tr'] \\
 & \vdash \\
 & \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2^t_f[tr \frown tr_0/tr']
\end{array}
$$

□

Lemma 75 (Approx theory:- Approx precondition)**.**

$$\neg\, (Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^f_f = ok \wedge \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']$$

Proof.

$$
\begin{array}{ll}
 & \neg\, Approx(\varepsilon)(\mathbf{R3}(P1 \models P2))^f_f \\
 & \left\{ \begin{array}{l} \text{Approx theory:- reactive design not waiting:} \\ Approx(\varepsilon)(\mathbf{R3}(P1 \models P2))_f \\ \quad = \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\ \quad\quad \vdash \\ \quad\quad \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'] \end{array} \right\} \\
= & \neg\, (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\
\vdash & \\
 & \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'])^f
\end{array}
$$

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{Law: C.9: Design theory:- abort:} \\ (P \vdash Q)^f = ok \Rightarrow P^f \end{array} \right\} \\
& \neg\,(ok \Rightarrow \neg\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'])_f) \\
= & \left\{ \text{Logic theory:- predicates} \right\} \\
& ok \wedge \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']_f
\end{array}
$$

□

Lemma 76 (Approx theory:- Approx postcondition)**.**

$$
\begin{array}{l}
(Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^t_f \\
\quad = \\
\quad ok \wedge (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \\
\qquad \Rightarrow \\
\qquad\quad (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'])
\end{array}
$$

Proof.

$$
\begin{array}{ll}
& (Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^t_f \\
= & \left\{ \begin{array}{l} \text{Approx theory:- reactive design not waiting:} \\ Approx(\varepsilon)(\mathbf{R3}(P1 \models P2))_f \\ \quad = \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\ \quad\ \vdash \\ \quad\ \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'] \end{array} \right\} \\
& (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\
& \vdash \\
& \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'] \\
&)^t \\
= & \left\{ \text{Law C.10: Design theory:- not abort: } (P \vdash Q)^t = (ok \wedge P^t \vdash Q^t) \right\}
\end{array}
$$

$$
\begin{array}{l}
\quad ok \\
\quad \wedge\ (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \\
\qquad \Rightarrow \\
\qquad\quad (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'])
\end{array}
$$

□

Corollary C.47 (Approx theory:- Approx reactive design)**.**

$$
\begin{array}{l}
Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)) \\
\quad = \mathbf{R3}(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\
\qquad \models \\
\qquad \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'])
\end{array}
$$

Proof.

$$
\begin{array}{l}
\quad Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)) \\
= \quad \left\{ \begin{array}{l} \text{Theorem C.22: CSP theory:- } \mathbf{R3}\text{-}\mathbf{R1}\text{-design process:} \\ P \text{ is } \mathbf{R2}\text{-healthy } \Rightarrow P = \mathbf{R3}(\neg\, P^f_f \models P^t_f) \end{array} \right\} \\
\quad \mathbf{R3}(\neg\, (Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^f_f \models (Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^t_f) \\
= \quad \left\{ \begin{array}{l} \text{Lemma 67: Approx theory:- precondition 1:} \\ \neg\, (Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^f_f = ok \wedge \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\ | \\ \text{Lemma 68: Approx theory:- postcondition:} \\ (Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)))^t_f \\ \qquad = ok \wedge (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \\ \qquad\quad \Rightarrow (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']) \end{array} \right\} \\
\quad \mathbf{R3}(ok \wedge \forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\
\qquad \models \\
\qquad\quad ok \wedge (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'])\ ) \\
\qquad\quad\ \Rightarrow \\
\qquad\quad\ (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'])
\end{array}
$$

$$
= \left\{ \begin{array}{l} \text{Lemma C.14: Design theory:- } \mathbf{R1}\text{-design } ok \text{ pre:} \\ (P \models Q) = (ok \land P \models Q) \\ | \\ \text{Lemma C.16: Design theory:- } \mathbf{R1}\text{-design } ok \text{ post:} \\ (P \models Q) = (P \models ok \land Q) \end{array} \right\}
$$

$$
\begin{array}{l} \mathbf{R3}(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\ \quad \models \\ \quad (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \\ \qquad \Rightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad) \\ \qquad\quad (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']) \end{array}
$$

$$
= \left\{ \begin{array}{l} \text{Lemma C.15: Design theory:- } \mathbf{R1}\text{-design precondition simp:} \\ (P \models P \land Q) = (P \models Q) \end{array} \right\}
$$

$$
\begin{array}{l} \mathbf{R3}(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\ \quad \models \\ \quad \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr'] \\ \quad) \end{array}
$$

□

Our next theorem is about the conformance between two reactive designs, expanding the algebraic statement of conformance. It replaces the conformance statement with two assertions, which are sound conditions for approximate conformance between two reactive designs under the healthiness condition **R3**, which distinguishes between waiting and final states. If the reactive design $\mathbf{R3}(Q1 \models Q2)$ conforms within ε to the reactive design $\mathbf{R3}(P1 \models P2)$, then we have:

1. An assertion about the two preconditions:

   $$[\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow Q1\,]$$

   If every trace tr' observed by Q could have been approximated, within ε, by extending some trace tr accepted by P, and if under that approximation the specification's precondition $P1$ would have held, then we may conclude that the implementation's precondition $Q1$ must also hold.

   In other words, if the specification would have accepted a close enough trace, then the implementation must not reject it. This reflects a robust precon-

Funded by
the European Union

dition guarantee: the implementation's admissible behaviours include all ε-perturbed variations of the specification's precondition.

2. An assertion about the two postconditions:

$$(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(Q2)$$
$$\Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']\,]$$

Premise 1 The specification would have accepted any ε-close trace:

$$\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']$$

Every trace tr' that the implementation may have produced is ε-approximated by some trace $tr \frown tr_0$, and the specification's precondition $P1$ holds under that approximated trace. So: we are assuming that the implementation's observed behaviour lies within an ε-margin of something the specification would have allowed.

Premise 2 The implementation behaves reactively well:

$$\mathbf{R1}(Q2)$$

This is the **R1** healthiness condition, which ensures trace monotonicity: the implementation's postcondition is well-behaved in terms of reactive extensions (e.g., appending to traces preserves feasibility). It's a technical condition needed to ensure that implementation behaviour does not cheat by shrinking or invalidating its own trace history.

Conclusion The specification can match some ε-close behaviour:

$$\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']$$

There exists a trace close to what the implementation produced such that the specification's postcondition $P2$ holds under that approximation.

Why does this matter? This is a trace-based lifting of value-level approximation: we are asserting that the implementation tolerates all ε-close observations of the specification. It is a forward-closed and over-approximating condition: the implementation may admit more behaviours, but all must be close to permitted specification behaviours. It is essential for sound reasoning in the presence of numerical error, rounding, or learned variability in reactive neural systems.

Here, at last, is the statement and proof of the theorem:

Theorem 12 (CSP theory:- reactive-design conformance)**.**

$$\mathbf{R3}(Q1 \models Q2)\ conf(\varepsilon)\ \mathbf{R3}(P1 \models P2)$$
$$= [\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow Q1\,]$$
$$\wedge$$
$$[\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(Q2)$$
$$\Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']\,]$$

Funded by
the European Union

Proof.

$$
\begin{array}{ll}
& \mathbf{R3}(Q1 \models Q2)\ conf(\varepsilon)\ \mathbf{R3}(P1 \models P2) \\
= & \left\{ \begin{array}{l} \text{Definition C.39: Approx theory:- conformance:} \\ Q\ conf(\varepsilon)\ P = Approx(\varepsilon)(P) \sqsubseteq Q \end{array} \right\} \\
& Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)) \sqsubseteq \mathbf{R3}(Q1 \models Q2) \\
= & \left\{ \begin{array}{l} \text{Theorem 11: Approx theory:- approx R3-R1-design expand:} \\ Approx(\varepsilon)(\mathbf{R3}(P1 \models P2)) = \mathbf{R3}(\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \models P2)[tr \frown tr_0/tr']) \end{array} \right\} \\
& \mathbf{R3}(\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \models P2)[tr \frown tr_0/tr']) \sqsubseteq \mathbf{R3}(Q1 \models Q2) \\
\Leftarrow & \left\{ \begin{array}{l} \text{Theorem C.23: CSP theory:- } \mathbf{R3}\text{-monotonicity:} \\ (P \sqsubseteq Q) \Rightarrow (\mathbf{R3}(P) \sqsubseteq \mathbf{R3}(Q)) \end{array} \right\} \\
& (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \models P2)[tr \frown tr_0/tr']) \sqsubseteq (Q1 \models Q2) \\
= & \left\{ \begin{array}{l} \text{Definition C.13: Design theory:- } \mathbf{R1}\text{-design:} \\ (P \models Q) = \mathbf{R1}(P \vdash Q) \end{array} \right\} \\
& (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \vdash \mathbf{R1}(P2))[tr \frown tr_0/tr']) \sqsubseteq (Q1 \vdash \mathbf{R1}(Q2)) \\
= & \left\{ \begin{array}{l} \text{Lemma 28: Design theory:- design-substitution:} \\ x \text{ isn't } \{ok, ok'\} \Rightarrow (P \vdash Q)[e/x] = (P[e/x] \vdash Q[e/x]) \end{array} \right\} \\
& (\exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1[tr \frown tr_0/tr'] \vdash (\mathbf{R1}(P2))[tr \frown tr_0/tr'])) \\
& \quad \sqsubseteq (Q1 \vdash \mathbf{R1}(Q2)) \\
= & \left\{ \begin{array}{l} \text{Law C.11: Design theory:- existential design:} \\ (\exists x \bullet (P \vdash Q)) = (\forall x \bullet P \vdash \exists x \bullet Q) \end{array} \right\} \\
& (\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr'] \\
& \vdash \\
& \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (\mathbf{R1}(P2))[tr \frown tr_0/tr']) \\
& \quad \sqsubseteq (Q1 \vdash \mathbf{R1}(Q2)) \\
= & \left\{ \begin{array}{l} \text{Theorem 6: Design theory:- design refinement:} \\ ((P1 \vdash P2) \sqsubseteq (Q1 \vdash Q2)) = [P1 \Rightarrow Q1] \wedge [P1 \wedge Q2 \Rightarrow P2] \end{array} \right\}
\end{array}
$$

Funded by
the European Union

$$[(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow Q1]$$

$$\wedge$$

$$[(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge Q2$$

$$\Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (\mathbf{R1}(P2))[tr \frown tr_0/tr']]$$

□

We prove the following result in the predicate calculus before using it in the next theorem.

Lemma 77 (Internal choice:- predicate)**.**

$$((\forall x \bullet P_1(x)) \wedge R_2 \Rightarrow \exists x \bullet P_2(x)) \wedge ((\forall x \bullet Q_1(x)) \wedge S_2 \Rightarrow \exists x \bullet Q_2(x))$$

$$\wedge (\forall x \bullet P_1(x) \wedge Q_1(x)) \wedge (R_2 \vee S_2)$$

$$\Rightarrow \exists x \bullet P_2(x) \vee Q_2(x)$$

Proof. We are given:

1. $(\forall x \bullet P1(x)) \wedge R2 \Rightarrow \exists x \bullet P2(x)$
2. $(\forall x \bullet Q1(x)) \wedge S2 \Rightarrow \exists x \bullet Q2(x)$
3. $\forall x \bullet P1(x) \wedge Q1(x)$
4. $R2 \vee S2$

We want to prove:

$\exists x \bullet P2(x) \vee Q2(x)$

From assumption (3), we have:

$\forall x \bullet P1(x)$ and $\forall x \bullet Q1(x)$

via conjunction elimination. Now consider (4), $R2 \vee S2$, and proceed by case analysis:

Case 1: $R2$ From (1) and knowing *forallx* $\bullet P1(x)$ and $R2$, we conclude:

$\exists x \bullet P2(x)$

Then, since $\exists x \bullet P2(x) \Rightarrow \exists x \bullet P2(x) \vee Q2(x)$ (disjunction introduction), we get the goal.

Case 2: $S2$ From (2) and $\forall x \bullet Q1(x)$ and $S2$, we conclude:

$\exists x \bullet Q2(x)$

Then, again using disjunction introduction:

$\exists x \bullet P2(x) \vee Q2(x)$

Funded by
the European Union

In both cases, we obtain the desired disjunction inside the existential. Thus, by disjunction elimination and universal instantiation, the conclusion follows:

$\exists x \bullet P2(x) \lor Q2(x)$

□

The next theorem is currently our key result: monotonicity of internal choice with respewct to conformance. If two implementations R and S each conform approximately (within ε) to specifications P and Q respectively, then a system that nondeterministically chooses between R and S also conforms (within ε) to a specification that nondeterministically chooses between P and Q .

Why does this matter? It ensures that approximate correctness is preserved under nondeterministic composition. In reactive or neural systems, R and S might represent different learned controllers or operational modes. The specification allows the system to behave either as P or Q. This theorem tells us that as long as each controller conforms individually, the overall controller (which switches between them nondeterministically) still conforms—within the same ε bound.

Why is it sound? The logic behind this is that conformance is defined pointwise: If all behaviours of R are ε-approximated by P, and similarly for S and Q, then any behaviour of $R \sqcap S$ is a behaviour of either R or S. So it must be ε-approximated by some behaviour of either P or Q, which are the behaviours of $P \sqcap Q$.

Formally, conformance is defined as: $R\ conf(\varepsilon)\ P = R \sqsubseteq Approx(\varepsilon)(P)$. Since refinement is preserved by internal choice (as we have shown earlier in our reactive design laws), and approximation is disjunctive (i.e., $Approx(\varepsilon)(P \sqcap Q) = Approx(\varepsilon)(P) \sqcap Approx(\varepsilon)(Q)$), the result follows.

What is the practical relevance of this theorem? If each of two implementation processes conforms approximately (within the same tolerance ε) to its respective specification, then their nondeterministic combination also conforms—under the same ε bound—to the nondeterministic combination of the specifications.

This property ensures that approximate conformance is compositional with respect to internal choice. In practice, it means that we can verify components of a system in isolation and then soundly infer approximate correctness of a nondeterministic combination of those components. The result is particularly relevant when modelling systems that include mode switches, failovers, or learned behaviours from multiple networks. It guarantees that combining approximate implementations using internal choice cannot lead to an overall violation of the specification, as long as each branch is itself conformant.

Theorem 13 (Conformance theory:- conformance internal choice monotonicity)**.**

$(R\ conf(\varepsilon)\ P) \land (S\ conf(\varepsilon)\ Q) \Rightarrow (R \sqcap S\ conf(\varepsilon)\ P \sqcap Q)$

Proof. We start by expanding the goal: $(R \sqcap S\ conf(\varepsilon)\ P \sqcap Q)$. It is sufficient to prove this for explicit reactive designs:

$P = \mathbf{R3}(P1 \models P2)$, $Q = \mathbf{R3}(Q1 \models Q2)$, $R = \mathbf{R3}(R1 \models R2)$, and $S = \mathbf{R3}(S1 \models S2)$.

Funded by
the European Union

We use these explicit reactive designs in our goal and expand it accordingly:

$$(\mathbf{R3}(R1 \models R2) \sqcap \mathbf{R3}(S1 \models S2))\ \mathit{conf}(\varepsilon)\ (\mathbf{R3}(P1 \models P2) \sqcap \mathbf{R3}(Q1 \models Q2))$$

$$= \left\{ \begin{array}{l} \text{Lemma CSP theory internal choice reactive design :} \\ \mathbf{R3}(P1 \models P2) \sqcap \mathbf{R3}(Q1 \models Q2) \ = \ \mathbf{R3}(P1 \wedge Q1 \models P2 \vee Q2) \end{array} \right\}$$

$$\mathbf{R3}(R1 \wedge S1 \models R2 \vee S2)\ \mathit{conf}(\varepsilon)\ \mathbf{R3}(P1 \wedge Q1 \models P2 \vee Q2)$$

$$= \left\{ \begin{array}{l} \text{Theorem 12: CSP theory:- reactive-design conformance:} \\ \mathbf{R3}(Q1 \models Q2)\ \mathit{conf}(\varepsilon)\ \mathbf{R3}(P1 \models P2) \\ = [\,(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow Q1\,] \\ \quad \wedge \\ \quad [\,(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(Q2) \\ \qquad \Rightarrow \exists\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']\,] \end{array} \right\}$$

$$[\,(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet (P1 \wedge Q1)[tr \frown tr_0/tr']) \Rightarrow R1 \wedge S1\,] \qquad (SG1)$$

$$\wedge$$

$$[\,(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet (P1 \wedge Q1)[tr \frown tr_0/tr']) \wedge \mathbf{R1}(R2 \vee S2) \qquad (SG2)$$

$$\Rightarrow \exists\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet (P2 \vee Q2)[tr \frown tr_0/tr']\,]$$

We refer to these as the first and second subgoals, $(SG1)$ and $(SG2)$, respectively. Rewriting the assumptions:

$$\mathbf{R3}(R1 \models R2)\ \mathit{conf}(\varepsilon)\ \mathbf{R3}(P1 \models P2) \wedge \mathbf{R3}(S1 \models S2)\ \mathit{conf}(\varepsilon)\ \mathbf{R3}(Q1 \models Q2)$$

$$\Rightarrow \left\{ \begin{array}{l} \text{Theorem 12: CSP theory:- reactive-design conformance:} \\ \mathbf{R3}(Q1 \models Q2)\ \mathit{conf}(\varepsilon)\ \mathbf{R3}(P1 \models P2) \\ = [\,(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow Q1\,] \\ \quad \wedge \\ \quad [\,(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(Q2) \\ \qquad \Rightarrow \exists\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']\,] \end{array} \right\}$$

$$((\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet Q1[tr \frown tr_0/tr']) \Rightarrow S1)$$

$$\wedge$$

$$((\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow R1)$$

$$\Rightarrow \left\{ \text{Logic theory:- predicates} \right\}$$

$$(\forall\, tr_0 : \mathit{seq\_approx}(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge Q1[tr \frown tr_0/tr']) \Rightarrow R1 \wedge S1$$

Funded by
the European Union

$$= \quad \left\{ \text{Logic theory:- substitution} \right\}$$

$$(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \wedge Q1)[tr \frown tr_0/tr']) \Rightarrow R1 \wedge S1$$

which discharges the first subgoal ($SG1$). Now we turn to ($SG2$).

$$\mathbf{R3}(R1 \models R2)\ conf(\varepsilon)\ \mathbf{R3}(P1 \models P2) \wedge \mathbf{R3}(S1 \models S2)\ conf(\varepsilon)\ \mathbf{R3}(Q1 \models Q2)$$

$$\Rightarrow \quad \left\{ \begin{array}{l} \text{Theorem 12: CSP theory:- reactive-design conformance:} \\ \mathbf{R3}(Q1 \models Q2)\ conf(\varepsilon)\ \mathbf{R3}(P1 \models P2) \\ = [\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \Rightarrow Q1\,] \\ \quad \wedge \\ \quad [\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(Q2) \\ \qquad \Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']\,] \end{array} \right\}$$

$$(\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(R2)$$

$$\Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet P2[tr \frown tr_0/tr']\,)$$

$$\wedge$$

$$(\,(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet Q1[tr \frown tr_0/tr']) \wedge \mathbf{R1}(S2)$$

$$\Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet Q2[tr \frown tr_0/tr']\,)$$

$$\Rightarrow \quad \left\{ \begin{array}{l} \text{Lemma 77: Internal choice:- predicate:} \\ ((\forall\, x \bullet P_1(x)) \wedge R_2 \Rightarrow \exists\, x \bullet P_2(x)) \wedge ((\forall\, x \bullet Q_1(x)) \wedge S_2 \Rightarrow \exists\, x \bullet Q_2(x)) \\ \wedge\ (\forall\, x \bullet P_1(x) \wedge Q_1(x)) \wedge (R_2 \vee S_2) \\ \qquad \Rightarrow \exists\, x \bullet P_2(x) \vee Q_2(x) \end{array} \right\}$$

$$(\forall\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P1 \wedge Q1)[tr \frown tr_0/tr']) \wedge \mathbf{R1}(R2 \vee S2)$$

$$\Rightarrow \exists\, tr_0 : seq\_approx(\varepsilon)(tr' - tr) \bullet (P2 \vee Q2)[tr \frown tr_0/tr']$$

□

A weaker result would be to let the two bounds be different. This would perhaps give a result that the nondeterministic choice conforms to the maximum of the two bounds.

We list some laws of the relational calculus in Table 4; we list laws of set theory in Table **??**; we list laws of the theory of sequences in Table **??**.

We take the following law for a local block from the theory of reactive processes (see Hoare and He [HJ98a, Chap. 8]). The assignment in the process **var** $x := e \bullet P$

Funded by
the European Union

1.	left unit	$P = \mathit{II} \,;\, P$
2.	right unit	$P = P \,;\, \mathit{II}$
3.	identity trace	$\mathit{II} \Rightarrow (tr' = tr)$
4.	identity lifting	$\mathit{II}_{+x}(\alpha) = \mathit{II}(\alpha \cup \{x\})$
5.	identity extension	$\mathit{II}(\alpha)_{+x} = \mathit{II}(\alpha \cup \{x\})$
6.	introduce local block	fresh $x \Rightarrow (P = \mathbf{var}\ x := e \bullet P_{+x})$
7.	assignment one point	$x :\in \{e\} = x := e$
8.	identity assignment	$\mathit{II}(w) = w := w$
9.	leading assignment	x not free in $e \Rightarrow (x := e \,;\, P = P[e/x])$
10.	alphabet extension	$P_{+x} = P \wedge (x' = x)$
11.	refinement monotonicity	$(P \sqsubseteq Q) \Rightarrow (P \,;\, R \sqsubseteq Q \,;\, R)$
12.	refinement transitivity	$(P \sqsubseteq Q) \wedge (Q \sqsubseteq R) \Rightarrow P \sqsubseteq R$
13.	variable block separation	$\mathbf{var}\ x := e \bullet P = \mathbf{var}\ x := e \,;\, P \,;\, \mathbf{end}\ x$
14.	variable block is **R1**???	
15.	Relations theory:- sequential composition	$P \,;_v\, Q$

Table 4: Some laws of the relational calculus

uses a law similar to the "leading assignment" in the refinement calculus [Mor94]. We substitute e for the initial value of x, and the result is $\mathbf{var}\, x \bullet P[e/x]$. Suppose we also know that x not free in e, then we can remove the declaration of x. This transformation is valid only if any preceding process has terminated. If it has not, the process $\mathbf{var}\ x := e \bullet P$ must behave as II. We cover this case by applying the healthiness condition **R3**. This derivation explains the law:

x not free in $e \Rightarrow ((\mathbf{var}\ x := e \bullet P) = \mathbf{R3}(P[e/x]))$

We use this law to simplify the algebraic definition of our simulation.

We use the sequence difference operator in the simulation's algebraic and non-algebraic statements. The next lemma establishes that this is well-defined.

Lemma 78 (Well-definedness of $upd\_tr(\epsilon)$)**.**

$$
\begin{array}{ll}
 & t := tr \,;\, P_{+t} \\
= & \quad \{\, \textit{relations: leading assignment: } x \textit{ not free in } e \Rightarrow (x := e \,;\, P = P[e/x]) \,\} \\
 & (P_{+t})[tr/t] \\
= & \quad \{\, \textit{relations: alphabet extension: } P_{+x} = P \wedge (x' = x) \,\} \\
 & (P \wedge (t' = t))[tr/t] \\
= & \quad \{\, \textit{assumption: } P \textit{ is } \mathbf{R1} \,\} \\
 & (P \wedge tr \leq tr' \wedge (t' = t))[tr/t] \\
= & \quad \{\, \textit{logic: substitution} \,\} \\
 & P[tr/t] \wedge tr \leq tr' \wedge (t' = tr) \\
\Rightarrow & \quad \{\, \textit{logic} \,\} \\
 & t' \leq tr'
\end{array}
$$

????????

Funded by the European Union

???????

We note that the update relation is not a process. It is not **R1**. Neither is it **R2**.

Funded by
the European Union

D Identifying Uncertainty in Self-Adaptive Robotics with Large Language Models

The appended paper follows.

Funded by
the European Union

IDENTIFYING UNCERTAINTY IN SELF-ADAPTIVE ROBOTICS WITH LARGE LANGUAGE MODELS

Hassan Sartaj
Simula Research Laboratory
Oslo, Norway
hassan@simula.no

Jalil Boudjadar
Aarhus University
Aarhus, Denmark
jalil@ece.au.dk

Mirgita Frasheri
Aarhus University
Aarhus, Denmark
mirgita.frasheri@ece.au.dk

Shaukat Ali
Simula Research Laboratory
Oslo, Norway
shaukat@simula.no

Peter Gorm Larsen
Aarhus University
Aarhus, Denmark
pgl@ece.au.dk

ABSTRACT

Future self-adaptive robots are expected to operate in highly dynamic environments while effectively managing uncertainties. However, identifying the sources and impacts of uncertainties in such robotic systems and defining appropriate mitigation strategies is challenging due to the inherent complexity of self-adaptive robots and the lack of comprehensive knowledge about the various factors influencing uncertainty. Hence, practitioners often rely on intuition and past experiences from similar systems to address uncertainties. In this article, we evaluate the potential of large language models (LLMs) in enabling a systematic and automated approach to identify uncertainties in self-adaptive robotics throughout the software engineering lifecycle. For this evaluation, we analyzed 10 advanced LLMs with varying capabilities across four industrial-sized robotics case studies, gathering the practitioners' perspectives on the LLM-generated responses related to uncertainties. Results showed that practitioners agreed with 63–88% of the LLM responses and expressed strong interest in the practicality of LLMs for this purpose.

***K*eywords** Self-Adaptive Systems · Robotics · Uncertainty · Large Language Models

1 Introduction

Self-adaptive robotics refers to robotic systems that autonomously adjust their behavior, configuration, or decision-making processes in response to environmental changes and unforeseen circumstances [1]. A fundamental framework for enabling self-adaptation in such systems is the MAPE-K (Monitoring, Analysis, Planning, Execution, and Knowledge) loop, which employs advanced techniques such as artificial intelligence and data analysis to allow robots to continuously collect and analyze data, plan and validate actions, execute adaptive behaviors, and refine decisions using a dynamic knowledge base [2]. At this level of autonomy, these robots must ensure dependability while effectively managing uncertainty and addressing ethical considerations to operate reliably in real-world environments.

A key challenge in self-adaptive robotics is managing uncertainty throughout the entire engineering lifecycle, from initial design to active operations [3]. Uncertainty may arise from various sources, such as unpredictable environmental conditions, sensor and actuator noise, and human-robot and robot-robot interactions. Such factors directly influence a robot's dependability, ultimately impacting its overall performance and decision-making capabilities. A crucial step in managing uncertainty within these systems is identifying the sources and potential impacts of uncertainties at early stages of the robotics software engineering lifecycle [3]. However, it is challenging due to the inherent complexity of self-adaptive robots, including their complex interactions, evolving configurations, and adaptive behaviors, as well as the lack of comprehensive knowledge about unpredictable operational contexts and environmental dynamics. Consequently,

Funded by
the European Union

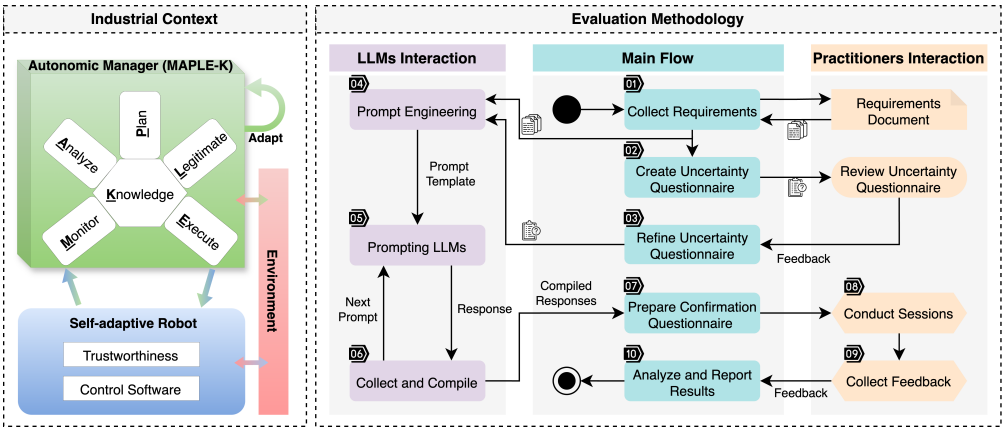


Figure 1: Industrial context (left) and evaluation methodology overview (right). In the workflow, rounded rectangles indicate our activities, ovals represent individual activities by the practitioners, and hexagons denote collaborative activities.

3 Evaluation Methodology

Goal: Our goal is to evaluate the effectiveness of LLMs in identifying uncertainties in self-adaptive robotics and to determine their practical value for the practitioners.

Figure 1 presents an overview of our evaluation methodology, encompassing the primary workflow and interactions involving the practitioners and LLMs. Below, we elaborate on each step of the process.

Step ❶: We collected documented robotics software requirements from the practitioners. These documents included detailed descriptions of each use case scenario, clearly specifying the operational context, expected behavior, objectives, and constraints. Moreover, they explicitly outlined hardware characteristics, including details of sensors and actuators, as well as thorough software requirements defining the system architecture, functionalities, and performance expectations.

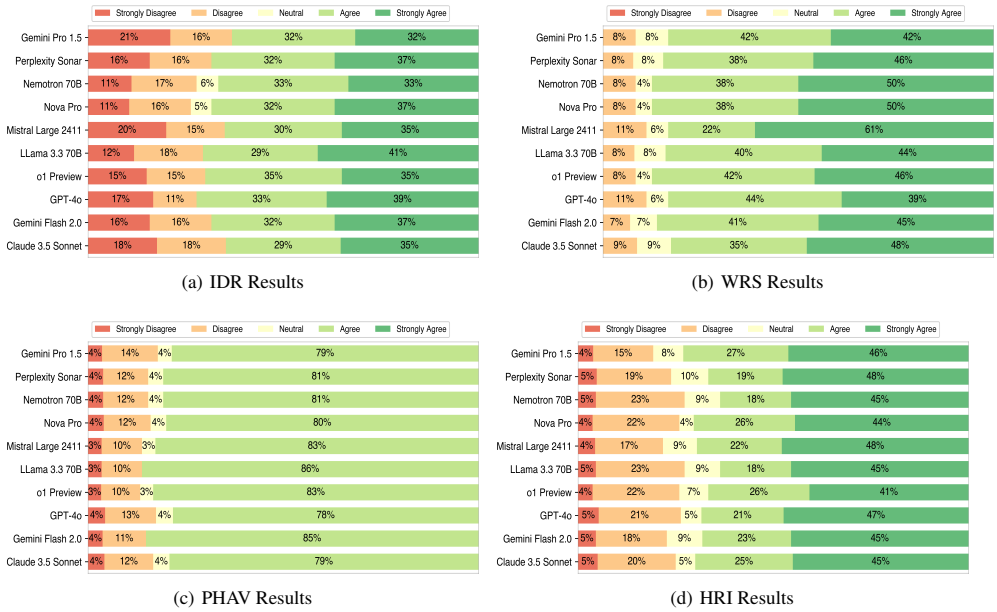
Steps ❷–❸: In the second step, we designed an uncertainty questionnaire inspired by existing work [7]. This questionnaire consisted of seven questions covering various aspects, including uncertainty sources, methods to identify uncertainties, impacted engineering lifecycle phases, the effects of uncertainties on performance and safety, potential mitigation strategies, real-world scenarios where uncertainties resulted in failures, and the overall impact of uncertainties on project outcomes. We sent out this questionnaire to the practitioners from all four use cases. In the third step, based on the practitioners' feedback, we revised the questionnaire by rewording certain questions to enhance clarity and comprehensibility.

Steps ❹–❻: The subsequent steps, from four to six, involve interaction with LLMs. In step four, we perform prompt engineering to prepare prompts for LLMs. Specifically, we used the role-prompting technique, which asks the model to assume a particular role or expertise. We provided LLMs with the requirement documents and asked them to assume the role of expert analyst. Next, we provided step-by-step instructions to follow, which included carefully reading and comprehending the requirements document, utilizing their robotics knowledge, understanding the purpose of the questionnaire, and responding to questions concisely with brief justifications. After finalizing the prompt (available on GitHub[1]), the subsequent steps involve iteratively prompting all selected LLMs for each case study and collecting and compiling LLMs' responses. Once we have obtained responses from all LLMs and case studies, the next step is to prepare confirmation questionnaires for each partner.

Step ❼: In the seventh step, to develop confirmation questionnaires, we analyzed all responses to identify unique responses, as we observed that some LLMs provided similar answers to specific questions during prompting. We then incorporated all unique responses into the confirmation questionnaire, removing any indication of which LLM generated a particular response. The resulting confirmation questionnaire contained all questions from the uncertainty

[1] https://github.com/Simula-COMPLEX/Prompt4RoboUI

Funded by the European Union



(a) IDR Results (b) WRS Results

(c) PHAV Results (d) HRI Results

Figure 2: Results for all LLMs across four case studies, presenting the practitioners' assessments of LLMs' effectiveness in identifying uncertainties.

questionnaire and responses from different LLMs. For each LLM-generated response, we used a five-point Likert scale to capture practitioners' level of agreement, ranging from "strongly disagree" and "disagree" to "neutral", "agree" and "strongly agree". The questionnaire was developed using Google Forms to facilitate efficient data collection and analysis. It is important to note that the confirmation questionnaire strictly focused on evaluating the responses provided by LLMs and did not include any questions related to personal information or participants' identities.

Steps ❽–❾: In the eighth step, we conducted four individual in-person sessions during project meetings with the practitioners involved in each case study. Each session included one senior and one junior participant from the respective case study. Each session lasted 40 minutes and followed a consistent format. At the beginning of each session, we provided participants with a concise overview of the planned activities and time allocation, presented the revised uncertainty questionnaire, and explained the objectives and motivations of the activity. After this introductory session of 10 minutes, we distributed the confirmation questionnaire to the participants. We allocated 20 minutes for the questionnaire and 10 minutes of verbal discussion. During the questionnaire activity, participants were asked to assess each response and indicate their level of agreement based on their experience and knowledge of the case study. In the ninth step, we collected participants' feedback on the confirmation questionnaire through Google Forms. During the verbal discussion, we gathered participants' opinions on the LLM-generated responses, the LLMs' understanding of the robotic case study, and their potential for identifying uncertainties. For this discussion, we opted to take informal notes instead of formally recording the feedback.

Step ❿: We analyzed the data collected through Google Forms, focusing mainly on feedback from the confirmation questionnaire. Our results are primarily derived from the responses to this questionnaire, supplemented by key observations gathered during verbal discussions. We analyze the results for each case study, specifically examining the agreement levels expressed by practitioners for each LLM-generated response. From this analysis, we identify and report which LLM-generated responses were confirmed by the practitioners, highlighting the levels of consensus and reliability for each response across all use cases. It is important to note that we obtained informed consent from the participants involved in all case studies to report the evaluation results.

Funded by
the European Union

4 Results and Insights

4.1 IDR Results

Figure 2(a) shows that the responses generated by GPT-4o received the highest number of agreements and the lowest number of disagreements. The second highest number of agreements was observed for the responses generated by o1 Preview and LLama 3.3 70B. Moreover, practitioners indicated neutral agreement only for the responses generated by Nemotron 70B and Nova Pro, at 6% and 5%, respectively. Regarding disagreements, the responses generated by Gemini Pro 1.5, Mistral Large 2411, and Claude 3.5 Sonnet received a notably higher number of disagreements compared to other models. Among these, practitioners strongly disagreed with most of the responses obtained from Gemini Pro 1.5 and Mistral Large 2411. For simple disagreement opinions, practitioners showed a high level of disagreement with responses generated by LLama 3.3 70B and Claude 3.5 Sonnet at 18%, whereas responses from GPT-4o received the lowest level of disagreement at 11%. Overall, the results indicate that practitioners agreed with more than 60% of the LLM-generated responses, including a majority of strong agreements.

4.2 WRS Results

From Figure 2(b), it can be observed that practitioners showed a high number of strong agreements with responses generated by Claude 3.5 Sonnet. For general agreements (including agreeing and strongly agreeing), practitioners agreed with the responses generated by Nemotron 70B, Nova Pro, and o1 Preview. Interestingly, none of the practitioners strongly disagreed with any of the responses. Moreover, overall disagreements are noticeable but remain up to 11%. For the responses from Gemini Flash 2.0, the number of disagreements is low compared to other models. The percentage of neutral agreements is 8% for the responses obtained from Gemini Pro 1.5, Perplexity Sonar, and LLama 3.3 70B. The overall results show that the practitioners agreed with more than 80% of the LLM-generated responses, indicating the high practicality of LLMs in this case study.

4.3 PHAV Results

Results in Figure 2(c) indicate that the responses generated by LLama 3.3 70B received the highest level of agreement and the fewest disagreements. Following closely, the responses generated by Gemini Flash 2.0 received the second-highest score in terms of agreement. For LLama 3.3 70B and Gemini Flash 2.0, practitioners did not indicate any neutral agreements. In contrast, other models exhibited neutral agreements, with practitioners showing up to 4% and 3% for responses generated by Mistral Large 2411 and o1 Preview, respectively. Furthermore, the number of strong disagreements remains relatively low, with 3% for the responses from Mistral Large 2411, LLama 3.3 70B, and o1 Preview, and 4% for other models. Similarly, the highest number of disagreements were observed for responses generated by Gemini Pro 1.5, while Mistral Large 2411, LLama 3.3 70B, and o1 Preview had the lowest disagreement rates. In general, practitioners expressed positive opinions for more than 80% of the LLM-generated responses, highlighting the high effectiveness of LLMs.

4.4 HRI Results

Figure 2(d) shows that the practitioners strongly agreed with many of the responses (48%) generated by Perplexity Sonar and Mistral Large 2411, followed by GPT-4o with 47% strong agreements. Notably, practitioners demonstrated strong agreement with over 40% of all LLMs' responses. Regarding the overall agreement, the highest number of positive opinions were particularly observed for responses generated by Nova Pro and o1 Preview. For neutral agreements, practitioners indicated the highest neutrality for responses generated by Perplexity Sonar, while Nova Pro received the least. In disagreements case, practitioners strongly disagreed with the smallest percentage (4–5%) of LLM-generated responses. In contrast, the percentage of simple disagreements ranged from 15% to 23%. Among all models, the responses generated by Perplexity Sonar and LLama 3.3 70B had the highest number of disagreements, while fewest disagreements were observed for Gemini Pro 1.5. Overall, practitioners provided positive opinions for more than 70% of the responses generated by LLMs, demonstrating their usefulness for this case study.

4.5 Insights

Below, we present key insights from results analysis and discussions with practitioners.

Funded by
the European Union

4.5.1 LLMs' Familiarity with Domain

When we asked practitioners from all case studies whether they believed LLMs could effectively understand the case study specifications and domain, all participants agreed that LLMs' responses indicated a strong grasp of the case study. This highlights LLMs' ability to comprehend text by using the provided information about various robots and their prior knowledge from related domains.

4.5.2 Logical Responses

We asked practitioners about their perspectives on whether LLMs' responses were logical and contextually relevant. All participants agreed that the responses were logical and aligned with the given context. This demonstrates LLMs' analytical capability and their usefulness in identifying uncertainties in self-adaptive robotics.

4.5.3 Novel Uncertainties

We asked practitioners whether they found the uncertainties identified by LLMs to be realistic and whether any of them were previously unknown yet logically valid. They unanimously agreed that the LLM-identified uncertainties were realistic and were particularly surprised by some previously unconsidered uncertainties, which they found valuable.

4.5.4 Knowledge Gaps in Engineering Phases

During the discussion, practitioners revealed a lack of knowledge about uncertainties across different robotics software engineering phases. Some participants were involved exclusively in development, others in testing, and some solely in the operations phase. Those involved in the development phase had limited awareness of uncertainties that could arise during operations, while those in the operations phase were unfamiliar with uncertainties in the design phase. Consequently, they could not confirm uncertainties outside their expertise and either disagreed or remained neutral when evaluating LLM-identified uncertainties related to phases they were not directly involved in.

4.5.5 Uncertainty in Engineering Phases

Although all practitioners agreed that uncertainties arise in every engineering phase, the majority emphasized that the most uncertainties are typically encountered during the testing phase. Those involved in the operational phase highlighted that uncertainties are equally prevalent during live operations. This was especially noticed in the autonomous vessel case study, where real-world operations in complex environments often introduce significant environmental uncertainties that may go undetected during simulation-based testing. Therefore, depending on the complexity and dynamics of the environment, both the testing and operational phases are likely to encounter uncertainties. Furthermore, all practitioners unanimously confirmed that environmental dynamics are a major source of uncertainty.

4.5.6 Managing Uncertainty

Our analysis revealed that the techniques mostly used to manage uncertainty are modeling, simulation, digital twins, and uncertainty quantification. However, we also noticed that applying these techniques, monitoring uncertain scenarios, and taking countermeasures require significant manual effort. This necessitates developing frameworks and tools to support automation. In this context, we foresee significant potential for LLMs, given the usefulness demonstrated in our evaluation.

4.5.7 LLM Recommendations

While all LLMs exhibited varying levels of usefulness in identifying uncertainties, their responses differed in depth and detail. Models such as Claude 3.5 Sonnet generated concise responses that lacked comprehensive explanations, whereas others, like o1 Preview, provided more detailed and well-reasoned answers for each point. This was particularly notable when LLMs were asked to provide example uncertainty scenarios for specific robotic cases, which practitioners endorsed and found interesting. Thus, based on the results analysis, we recommend LLama 3.3 70B, o1 Preview, GPT-4o, Mistral Large 2411, and Nova Pro as the most effective LLMs for uncertainty analysis.

4.6 Threats to Validity

A potential threat to external validity is the limited sample size of eight practitioners in four use cases. Furthermore, the robotic cases studied specifically focused on small subsystems dedicated to self-adaptive robotic behaviors. Therefore, results might not generalize to all robotic systems, which is a common threat in empirical studies. An internal validity

Funded by
the European Union

threat may occur due to participants' personal biases and familiarity with robotic systems. Although our sessions with practitioners involved two participants per case (one junior and one senior), their backgrounds and confidence levels may have influenced their responses. To handle this, we included open discussions during sessions to clarify interpretations. A potential conclusion validity threat may occur due to variability in practitioners' experience levels and the subjective nature of agreement measurements (i.e., Likert levels). To manage this, we used standardized procedures in all sessions, providing clear instructions and consistent timing, although subjective assessment remains an inherent challenge. Moreover, to address the validity threat from informal notes during verbal discussions, we sent the draft article to all participating practitioners for review and revised it based on their feedback. A possible threat involves the clarity of the uncertainty questionnaire and confirmation questionnaire. To mitigate this, we developed the uncertainty questionnaire based on existing frameworks and revised it based on practitioners' feedback. We also used the standard Likert scale to capture agreement levels for the confirmation questionnaire precisely.

5 Related Works

Despite tremendous research efforts on identifying uncertainty in self-adaptive robots [8, 9, 10, 11, 12], a systematic approach to identifying and mitigating uncertainties is still lacking [13]. These studies focus on specific types, sources, and dimensions of uncertainty separately, overlooking complex interdependencies and environmental correlations [14]. Hezavehi *et al.* [9] conducted a field survey and identified that uncertainties in self-adaptive systems are mostly application-dependent and rely on non-functional requirements. Chirayil *et al.* [12] proposed a unified classification of anomalies for autonomous robotic missions. Similarly, Busch *et al.* [13] proposed a setup for identifying and quantifying the uncertainty of the eigenfrequency in machining robots. Recently, Zheng *et al.* [8] developed an uncertainty-based LLM failure detector for autonomous robots to enable efficient task planning. Moreover, Betzer *et al.* [15] developed a cloud-based digital twin to enable real-time identification and mitigation of uncertainties for an autonomous mobile robot. Compared to these studies, our work explores LLMs' potential in systematically identifying uncertainties in self-adaptive robotics.

6 Conclusion

We evaluated the LLMs' potential to support uncertainty identification in four industrial-sized self-adaptive robotic cases using 10 LLMs. We provided these LLMs with case study requirements and a prompt that contained an uncertainty questionnaire. The responses generated by the LLMs were then evaluated with input from the practitioners. Results demonstrated that practitioners agreed with 63–88% of the responses generated by all LLMs. Furthermore, discussions with practitioners highlighted the usefulness of LLMs in understanding robotic cases, generating logical responses, and identifying novel uncertainties. Our evaluation suggests two promising research directions: creating an uncertainty taxonomy for self-adaptive robotics and developing an LLM-based framework for automatically identifying and managing uncertainty.

Acknowledgments

This work is funded by the RoboSAPIENS project under the EU Horizon Europe program (Grant No. 101133807).

References

[1] Peter G Larsen, Shaukat Ali, Roland Behrens, Ana Cavalcanti, Claudio Gomes, Guoyuan Li, Paul De Meulenaere, Mikkel L Olsen, Nikolaos Passalis, Thomas Peyrucain, et al. Robotic safe adaptation in unprecedented situations: the RoboSAPIENS project. *Research Directions: Cyber-Physical Systems*, 2:e4, 2024.

[2] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[3] Danny Weyns, Radu Calinescu, Raffaela Mirandola, Kenji Tei, Maribel Acosta, Nelly Bencomo, Amel Bennaceur, Nicolas Boltz, Tomas Bures, Javier Camara, et al. Towards a research agenda for understanding and managing uncertainty in self-adaptive systems. *ACM SIGSOFT Software Engineering Notes*, 48(4):20–36, 2023.

[4] Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. The Dawn of LMMs: Preliminary Explorations with GPT-4V(ision), 2023.

[5] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

Funded by
the European Union

[6] Ahmed E. Hassan, Gustavo A. Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, and Jiang. Rethinking Software Engineering in the Foundation Model Era: From Task-Driven AI Copilots to Goal-Driven AI Pair Programmers, 2024.

[7] Andres J Ramirez, Adam C Jensen, and Betty HC Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 99–108. IEEE, 2012.

[8] Zhi Zheng, Qian Feng, Hang Li, Alois Knoll, and Jianxiang Feng. Evaluating uncertainty-based failure detection for closed-loop llm planners. In *ICRA 2024 Workshop on Back to the Future: Robot Learning Going Probabilistic*, 2024.

[9] Sara M. Hezavehi, Danny Weyns, Paris Avgeriou, Radu Calinescu, Raffaela Mirandola, and Diego Perez-Palacin. Uncertainty in self-adaptive systems: A research community perspective. *ACM Trans. Auton. Adapt. Syst.*, 15(4), 2021.

[10] Harriet R. Cameron, Simon Castle-Green, Muhammad Chughtai, Liz Dowthwaite, Ayse Kucukyilmaz, Horia A. Maior, Victor Ngo, Eike Schneiders, and Bernd C. Stahl. A taxonomy of domestic robot failure outcomes: Understanding the impact of failure on trustworthiness of domestic robots. In *Proceedings of the Second International Symposium on Trustworthy Autonomous Systems*, 2024.

[11] Ho Suk, Yerin Lee, Taewoo Kim, and Shiho Kim. Chapter ten - addressing uncertainty challenges for autonomous driving in real-world environments*equally contributed. In *Artificial Intelligence and Machine Learning for Open-world Novelty*, volume 134 of *Advances in Computers*, pages 317–361. Elsevier, 2024.

[12] Shivoh Chirayil Nandakumar, Daniel Mitchell, Mustafa Suphi Erden, David Flynn, and Theodore Lim. Anomaly detection methods in autonomous robotic missions. *Sensors*, 24(4), 2024.

[13] Maximilian Busch, Florian Schnoes, Amr Elsharkawy, and Michael F Zaeh. Methodology for model-based uncertainty quantification of the vibrational properties of machining robots. *Robotics and Computer-Integrated Manufacturing*, 73, 2022.

[14] Ke Wang, Chongqiang Shen, Xingcan Li, and Jianbo Lu. Uncertainty quantification for safe and reliable autonomous vehicles: A review of methods and applications. *IEEE Transactions on Intelligent Transportation Systems*, 2025.

[15] Joakim Schack Betzer, Jalil Boudjadar, Mirgita Frasheri, and Prasad Talasila. Digital twin enabled runtime verification for autonomous mobile robots under uncertainty. In *International Symposium on Distributed Simulation and Real Time Applications*, 2024.

Funded by
the European Union

E Assessing the Uncertainty and Robustness of the Laptop Refurbishing Software

The appended paper follows.

Funded by
the European Union

Assessing the Uncertainty and Robustness of the Laptop Refurbishing Software

Chengjie Lu
Simula Research Laboratory and University of Oslo
Oslo, Norway
chengjielu@simula.no

Jiahui Wu
Simula Research Laboratory and University of Oslo
Oslo, Norway
jiahui@simula.no

Shaukat Ali
Simula Research Laboratory
Oslo, Norway
shaukat@simula.no

Mikkel Labori Olsen
Danish Technological Institute
Odense, Denmark
miol@teknologisk.dk

***Abstract*—Refurbishing laptops extends their lives while contributing to reducing electronic waste, which promotes building a sustainable future. To this end, the Danish Technological Institute (DTI) focuses on the research and development of several robotic applications empowered with software, including laptop refurbishing. Cleaning represents a major step in refurbishing and involves identifying and removing stickers from laptop surfaces. Software plays a crucial role in the cleaning process. For instance, the software integrates various object detection models to identify and remove stickers from laptops automatically. However, given the diversity in types of stickers (e.g., shapes, colors, locations), identification of the stickers is highly uncertain, thereby requiring explicit quantification of uncertainty associated with the identified stickers. Such uncertainty quantification can help reduce risks in removing stickers, which, for example, could otherwise result in software faults damaging laptop surfaces. For uncertainty quantification, we adopted the Monte Carlo Dropout method to evaluate six sticker detection models (*SDM*s) from DTI using three datasets: the original image dataset from DTI and two datasets generated with vision language models, i.e., *DALL-E-3* and *Stable Diffusion-3*. In addition, we presented novel robustness metrics concerning detection accuracy and uncertainty to assess the robustness of the *SDM*s based on adversarial datasets generated from the three datasets using a dense adversary method. Our evaluation results show that different *SDM*s perform differently regarding different metrics. Based on the results, we provide *SDM* selection guidelines and lessons learned from various perspectives.**

***Index Terms*—Uncertainty Quantification, Robustness Evaluation, Object Detection, Deep Neural Network**

I. Introduction

The European Union's Circular Economy Action Plan (CEAP) highlights the need for sustainable operations to promote circular economy processes and encourage sustainable consumption [1]. One essential activity is refurbishing electronic devices, e.g., laptops, to extend their lives, reduce electronic waste, and provide affordable options for consumers. A critical and time-consuming step in refurbishment is removing stickers from the laptop by first identifying stickers and their locations. Manual cleaning is time-consuming and faces challenges in finding enough skilled workers, and current automation solutions are not built for this amount of variation, limiting sustainability and scalability. Thus, novel solutions to automate the processes are needed. Robotics offers a promising solution to simplify and scale up this process, increasing efficiency and reducing labor costs.

The Danish Technological Institute (DTI) develops, applies, and transfers technology to industry and society. One leading area that DTI focuses on is laptop refurbishing automatically with robots, where software is a crucial part of all laptop refurbishing steps. The software responsible for the cleaning process in laptop refurbishment integrates deep neural networks (DNN)–based sticker detection models (*SDM*s) built by DTI for automatic sticker detection, which is the basis for successful automatic sticker removal. The *SDM*s are built on open-source object detection DNNs and trained using a sticker detection dataset specially designed by DTI. Due to inappropriate model architecture or insufficient training data, the design and training process may introduce uncertainties into the *SDM*s, making them vulnerable under certain conditions, such as adversarial attacks, noisy data, or unforeseen input distributions [2]. This vulnerability highlights the need to quantify uncertainty and evaluate the robustness of the *SDM*s and software they are integrated in, which is crucial for trustworthy sticker removal, as incorrect detection may damage the laptop surface or incomplete sticker removal.

As a first step towards holistic uncertainty quantification (UQ) and handling in laptop refurbishing robotic software, we conduct a comprehensive empirical evaluation to assess the *SDM*s DTI uses regarding detection accuracy, prediction uncertainty, and adversarial robustness. Specifically, we adopt Monte Carlo Dropout (MC-Dropout) [3] as the UQ method to capture the uncertainty in model predictions. We run the model to perform multiple predictions, and based on these predictions, we calculate two types of UQ metrics: label classification UQ metrics and bounding box regression UQ metrics. We employ Dense Adversary Generation (DAG) [4] as the adversarial attack and define *robustness score (RS)* to measure the robustness of the *SDM*s. *RS* measures robustness from two perspectives: robustness concerning predictive precision and robustness concerning prediction uncertainty. Regarding benchmark datasets, we construct benchmark datasets from three data sources: datasets provided by our partner DTI, datasets synthesized by prompting two vision language models (VLMs), i.e., *DALL-E-3* [5] and *Stable Diffusion-3* [6], and datasets created using the adversarial attack, i.e., DAG. Our evaluation results show that different *SDM*s achieve different performance regarding different evaluation metrics. Specifically, regarding

Funded by the European Union

datasets synthesized by prompting two VLMs, and datasets created using an adversarial attack technique. For each image in the benchmark datasets, the robot's camera captures the image, which is then processed by specialized robotic software for sticker detection. Specifically, the *SDM* in the robotic software makes T predictions based on the MC-Dropout method to capture the prediction uncertainty. The T predictions are then fed to the UQ component in the robotic software to calculate UQ metrics. The UQ component first employs the density-based algorithm, HDBSCAN, to cluster the detected stickers in T predictions and then calculates the UQ metrics based on the detected stickers. We consider two types of UQ metrics: label classification UQ metrics and bounding box regression UQ metrics. We present the MC-Dropout method in Section III-A and the UQ metrics in Section III-B. Sections III-C and III-D introduce VLM-based and adversarial attack-based dataset creation.

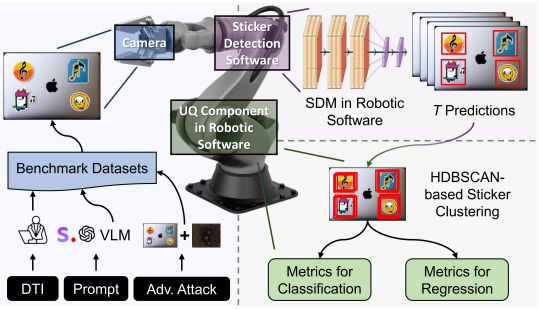


Fig. 1: Overview of the Uncertainty Quantification Process for Sticker Detection Models in the Robotic Software.

A. Monte Carlo Dropout-Based Uncertainty Quantification

Bayesian probabilistic theory [10] is a universal theoretical framework for uncertainty reasoning, and as discussed in Section II-B, MC-dropout is a Bayesian approximation for quantifying uncertainty. Hence, we adopt MC-Dropout as the UQ method and inject inference-time activated dropout layers into the pre-trained *SDM*s. By doing so, we build uncertainty-aware variants of the original models. Specifically, for a *SDM* f, we first build its variant as f^p by applying dropout layers with dropout rate p. Next, we let f^p make T predictions for a given image input X to obtain a sample of outputs $O = \{f_t^p\}_{t=1}^T$. The outputs O represent a sample from the model's predictive distribution, and MC-Dropout quantifies the uncertainty of f^p's outputs by extracting information regarding the variability of the T predictions. We set T to 20 according to existing guidelines [3].

B. Uncertainty Quantification Metrics

1) Basic Concepts: As discussed in Section III-A, we utilize MC-Dropout to quantify the uncertainty over T prediction outputs given an input image X: $O = \{f_t^p\}_{t=1}^T$. A prediction f_t^p is a vector containing N different detected objects, and a detected object $f_t^p(i), (1 \leq i \leq N)$ is characterized by its label classification and bounding box regression. The label classification assigns the softmax score $sm_{f_t^p(i)}$ for the candidate objects of three classes: stickers, logos, and backgrounds. The softmax score is a vector of probabilities, and we take the position with the maximum probability value as the predicted label, i.e., $argmax()$. The bounding box regression returns four different real values $box_{f_t^p(i)} = \{x_1, x_2, y_1, y_2\}$ to locate the detected object. Specifically, a bounding box is a rectangle whose start and end points are denoted as (x_1, y_1) and (x_2, y_2). In a deterministic model without uncertainty, all predictions should be the same, for example, a detected object $f_t^p(i)$ in prediction f_t^p should also be detected in other predictions and have the same softmax score and bounding box regression values as those in other predictions. However, recall that MC-Dropout quantifies the uncertainty by utilizing T model predictions with randomly dropped units. Therefore, each prediction is not necessarily the same. The model based on MC-Dropout can produce predictions with different locations for each object in a single image. To this end, a clustering method is needed to cluster objects in the T predictions. Details will be introduced in the following section.

2) HDBSCAN-based Object Clustering: We adopt the hierarchical density-based clustering algorithm, HDBSCAN [11], to cluster objects based on their predicted bounding boxes. The algorithm identifies clusters by calculating core distances for each point and adjusting these distances based on local density. It then builds a minimum spanning tree to form a hierarchy of clusters, which is condensed to highlight meaningful clusters. This hierarchical structure enables the recognition of clusters of various shapes and sizes. Two key parameters need to be adjusted: *minimum samples* $(minSamples)$, which sets the minimum number of samples in the neighborhood for a point to be considered a core point, and *minimum cluster size* $(minPts)$, which determines the minimum number of points to be considered as a cluster. Based on our preliminary study, we set both $minSamples$ and $minPts$ to 3. As a result, each cluster represents a detected object in the T predictions whose label classification softmax score and the bounding box regression are denoted as $\{sm_k\}_{k=1}^W$ and $\{box_k\}_{k=1}^W$. Note that W is the number of times the object is detected, and it is in the range $[1, T]$, as some objects are not always detected in all T predictions. We then calculate UQ metrics for each detected object in the T predictions.

3) Uncertainty Metrics for Label Classification: We employ three common metrics to measure uncertainty in classification tasks [12], including Variation Ratio (*VR*) [13], Shannon Entropy (*SE*) [14], and Mutual Information (*MI*) [14]. *VR* is a measure of dispersion. It is calculated by determining the proportion of cases that are not in the mode (the most frequent category) relative to the total number of cases. For the detected object in each cluster, we first extract a set of labels as the classes with the highest softmax scores:

$$\mathbf{y} = \{\text{argmax}(sm_k)\}_{k=1}^W, \tag{2}$$

we then find the mode of the distribution as:

$$c^* = \underset{c=1,\ldots,C}{\text{argmax}} \sum_k \mathbb{1}[\mathbf{y}^k = c], \tag{3}$$

Funded by the European Union

where $\mathbb{1}$ is the indicator function. The number of times c^* was sampled is $\sum_k \mathbb{1}[\mathbf{y}^k = c^*]$. Finally, *VR* is calculated as:

$$VR = 1 - \frac{\sum_k \mathbb{1}[\mathbf{y}^k = c^*]}{W}. \quad (4)$$

VR ranges from 0 to $2/3$. It attains the maximum value of $2/3$ when all three classes are sampled equally and the minimum value of 0 when only a single class is sampled.

SE captures the average amount of information contained in the predictive distribution. For a detected object, we calculate *SE* by considering softmax scores over T predictions, i.e., $\{sm_k\}_{k=1}^{W}$ as:

$$SE = -\sum_{c=1}^{nc}(\frac{1}{W}\sum_{k=1}^{W} sm_k(c)) \times \log(\frac{1}{W}\sum_{k=1}^{W} sm_k(c)), \quad (5)$$

where nc denotes the number of classes, i.e., 3. *SE* values range from 0 to $\log(3)$, reaching a maximum of $\log(3)$ when all classes are predicted to have the same softmax score, i.e., $1/3$, indicating the most uncertain case. Its minimum value is 0 when the probability of one class is 1 and all other probabilities are 0, indicating no uncertainty in the classification.

MI quantifies the information difference between the predicted and posterior of the model parameters, providing a different uncertainty measure for classification tasks. For the detected object in each cluster, *MI* can be computed as [12]:

$$\begin{aligned} MI = &-\sum_{c=1}^{nc}(\frac{1}{W}\sum_{k=1}^{W} sm_k(c)) \times \log(\frac{1}{W}\sum_{k=1}^{W} sm_k(c)) \\ &+\frac{1}{W}\sum_{k=1}^{W}\sum_{c=1}^{nc} sm_k(c) \times \log sm_k(c), \end{aligned} \quad (6)$$

where nc denotes the number of classes, i.e., 3. *MI* measures the model's confidence in its outputs. It ranges from 0 to 1, and the larger the *MI* value, the higher the uncertainty.

4) Uncertainty Metrics for Bounding Box Regression: For each detected object, we estimate the uncertainty for bounding box regression using Total Variance (*TV*) [15] and Predictive Surface (*PS*) [16]. *TV* captures the uncertainty in the bounding box regression by calculating the trace of the covariance matrix of $\{box_k\}_{k=1}^{W}$, which sums the variances of each variable in the bounding box. Specifically, a bounding box has four variables used to locate an object: $box = \{x_1, x_2, y_1, y_2\}$, for variable $v \in box$, we calculate its variance as:

$$\delta_v^2 = \frac{1}{W-1}\sum_{k=1}^{W}(v_k - \mu_v)^2, \quad (7)$$

where δ^2 is the variance of v, μ is the mean value of v, and W is the number of times the object is detected. We then calculate *TV* by summing the variances for all variables:

$$TV = \sum_{v \in box} \delta_v^2. \quad (8)$$

TV ranges in $[0, +\infty)$, and larger *TV* means higher uncertainty.

PS was originally proposed to quantify prediction uncertainty in object detection models for autonomous driving [16]. It measures uncertainty by considering the convex hull of each corner point of the predicted bounding box. A convex hull is the smallest convex shape enclosing all points and *PS* estimates uncertainty by calculating the area covered by the convex hull. To be concrete, four corner points define the bounding box to locate an object: (x_1, y_1), (x_2, y_2), (x_2, y_1), and (x_1, y_2). For the object in each cluster where the bounding box is detected W times, i.e., $\{box_k\}_{k=1}^{W}$, we first obtain the cluster of each corner point as:

$$CP = \{(x_1, y_1)_k, (x_2, y_2)_k, (x_2, y_1)_k, (x_1, y_2)_k\}_{k=1}^{W}, \quad (9)$$

we then identify the convex hulls for its four corner points, respectively, as:

$$convexh = ConvexHull(cp), cp \in CP. \quad (10)$$

Finally, we calculate *PS* by averaging the area covered by the convex hulls of the four corner points to approximate prediction uncertainty:

$$PS = \frac{1}{|CP|}\sum_{cp \in CP} Area(convexh_{cp}). \quad (11)$$

PS ranges in $[0, +\infty)$, and a higher *PS* value indicates higher variability of the predicted corner points, meaning higher uncertainty in the bounding box prediction.

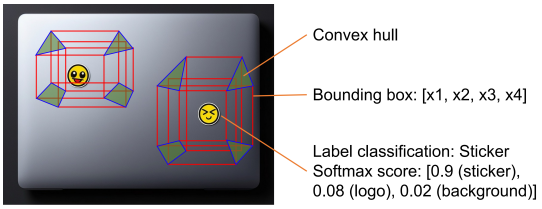


Fig. 2: An example for sticker detection. The left column shows the laptop image with two stickers on it. The right column shows the sticker detection outputs and what the convex hull looks like in the left image.

Recall that for an input image X, multiple objects will be detected, so after performing the HDBSCAN clustering algorithm, multiple clusters will be obtained, each cluster representing a detected object. To get the uncertainty of the input image X, for each UQ metric, we take the average of all metric values of all detected objects in X. Figure 2 shows an example of using the *SDM* to detect stickers on a laptop and how to calculate UQ metrics based on the model outputs. The *SDM* makes T predictions, and each prediction returns a label based on the softmax score and a bounding box containing four corner points. The convex hull of each corner point is then calculated. The UQ metrics for label classification are calculated based on the softmax scores in T predictions. Regarding UQ metrics for bounding box regression, *TV* is calculated based on the variance of each corner point, and *PS* is calculated based on the area covered by convex hulls.

Funded by
the European Union

C. VLM-based Image Generation

VLMs are generic pre-trained multimodal models that learn from large-scale image-text pairs sourced from the Internet, enabling them to directly address downstream visual tasks without task-specific fine-tuning [17]. Therefore, we apply the zero-shot generalization ability of VLMs to generate a synthetic image dataset to study whether *SDM*s can identify stickers on laptops that they have not encountered before. Following the representative categories of VLMs, i.e., autoregressive and diffusion models [18], we select two state-of-the-art VLMs: *DALL-E-3* [5] and *Stable Diffusion-3* [6]. *DALL-E-3* is the latest VLM developed by OpenAI, leveraging advanced transformer architectures to generate images from textual descriptions. *Stable Diffusion-3*, developed by StablilityAI, is a text-to-image generation VLM that uses diffusion processes to create images from textual descriptions.

To ensure consistency and equity in using VLMs, we design the same prompt template to prompt the selected VLMs. Table I lists the main prompt phrases and relevant examples, which involve common laptop models, the number, size, style, and location of generated stickers, and other detailed image content information. The prompt template consists of these phrases, e.g., *"a laptop, three stickers on the lid, one small music sticker, one small animal sticker, one small superhero sticker, stickers close to each other, top-down view, screen turn off, on a flat surface, a solid dark background"*. Note that to enhance the generality of the synthesized dataset, the prompt template omits information about the quality of the generated images, e.g., *sharp focus* or *highly detailed*, thereby generating images with varying levels of clarity that can be used to test and evaluate the generalization capability of *SDM*s.

D. Adversarial Image Generation

DNNs are vulnerable to adversarial attacks, which are imperceptible to humans, but easily misclassified by DNNs [19]. To study the robustness of *SDM*s against adversarial attacks and how these attacks affect the prediction uncertainty of *SDM*s, we construct an adversarial image dataset by perturbing images in an original dataset. Specifically, given an image X containing N targets $\mathcal{T} = \{t_1, t_2, ..., t_N\}$, each target is labeled by a ground-truth $l_n \in \{sticker, logo, background\}$, an adversarial attacker aims to inject perturbations on X to obtain a perturbed image X' that satisfies:

$$\forall 1 \leq n \leq N \wedge t_n \in \mathcal{T} : f(X', t_n) = l'_n \neq f(X, t_n) = l_n, \tag{12}$$

where f denotes the *SDM*. As a result, the generated adversarial image X' makes all targets incorrectly predicted. We adopt an adversarial attack approach called Dense Adversary Generation (DAG) [4] to construct the adversarial image dataset. DAG aims to generate adversarial examples for semantic segmentation and object detection tasks where multiple objects must be recognized. It requires an original dataset to perturb the images in it. We use the dataset to evaluate the pre-trained *SDM* as the original dataset and then add adversarial perturbations to the images in the original dataset to obtain an adversarial image dataset.

TABLE I: Prompt Phrase with Example

Phrase	Example
laptop model	*"a laptop", "a MacBook laptop", "a Dell laptop", "an HP laptop", ...*
sticker number	*"one", "two", "three", ...*
sticker size	*"small", "medium", "big"*
sticker style	*"Apple logo", "Dell logo", ..., "cartoon character", "animal", ...*
sticker location	*"close to the center of the laptop", "stickers close to each other", ...*
other	*"top-down view", "screen turn off", "a solid dark background", ...*

IV. Experiment Design

A. Sticker Detection Models

We obtain six pre-trained *SDM*s employed in the robotic software, representing state-of-the-art DNN-based object detection models. These models are designed by considering multiple model architectures which are *Faster R-CNN* [20], *Faster R-CNN_v2* [21], *RetinaNet* [22], *RetinaNet_v2* [23], *SSD300* [24], and *SSDLite* [25]. DTI trained these models using a specially designed sticker detection dataset containing thousands of labeled laptop images with stickers in various poses, sizes, and lighting conditions. All six *SDM*s are trained based on the open-source implementation in PyTorch [26].

We modify the six pre-trained models to be compatible with the MC-Dropout method by injecting prediction-time activated dropout layers. Specifically, for *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2*, we add dropout layers to the convolutional layers of the Feature Pyramid Network, which is a common component of these four models and built on the backbones of these four models. Regarding *SSD300* and *SSDLite*, we add dropout layers to their detection heads. In our experiments, we choose 9 dropout ratios from 0.1 to 0.5 with an interval of 0.05 to study the effect of dropout rates on the performance of *SDM*s.

B. Benchmark Datasets

We obtain a dataset containing 150 images from DTI, i.e., *origImg*, and synthesize two datasets by prompting two VLMs, *DALL-E-3* and *Stable Diffusion-3*, i.e., *dalleImg* and *stableImg*, each containing 150 images. By applying the adversarial attack method (DAG) to *origImg*, *dalleImg*, and *stableImg*, we generate three adversarial datasets: *advOrigImg*, *advDalleImg*, and *advStableImg*. Specifically, for each image X in each dataset, we execute DAG 10 times and obtain 10 adversarial images for X, and therefore, each adversarial dataset contains 1500 images. These three adversarial datasets are used to assess the robustness of the *SDM*s.

C. Evaluation Metrics

Mean Average Precision (mAP) is a metric for evaluating the accuracy of object detectors [27] and is calculated as:

$$mAP = \frac{1}{nc} \sum_{c=1}^{nc} AP_c, \tag{13}$$

where AP_c is the average precision for class c and nc is the number of classes. AP is the area under the precision-recall curve $p(r)$ that is calculated as $\int_0^1 p(r)dr$. For object detection tasks, precision measures how accurate the predictions are and is calculated as the number of true positives divided by the number of all detected objects, while recall measures how good

Funded by the European Union

TABLE II: Datasets, metrics, and statistical tests for each RQ

RQ	Dataset	Metric	Statistical Test
1	*origImg*, *dalleImg*, *stableImg*	*mAP*	Friedman test, Wilcoxon Signed-Rank test, rank-biserial correlation, Holm–Bonferroni method
2	*origImg*, *dalleImg*, *stableImg*	*VR*, *SE*, *MI*, *TV*, *PS*	
3	*origImg*, *advOrigImg*, *dalleImg*, *advDalleImg*, *stableImg*, *advStableImg*	RS_{mAP}, RS_{uq}	
4	*origImg*, *dalleImg*, *stableImg*	*mAP*, *VR*, *SE*, *MI*, *TV*, *PS*	Spearman's rank correlation, Holm–Bonferroni method

the model is at recalling classes and is defined as the number of true positives divided by the number of all ground-truth objects. True positives are determined based on Intersection over Union (IoU), which quantifies how close the predicted and ground-truth bounding boxes are by taking the ratio between the area of intersection and the area of the union of the predicted and ground-truth boxes: $IoU = (A_P \cap A_G)/(A_P \cup A_G)$, where A_P and A_G are the area of the predicted and ground-truth boxes, respectively. Then, if the IoU of the predicted and true boxes is greater than a threshold and the object is correctly classified, they are considered a match and therefore a true positive. We set the threshold to 0.5, commonly used for object detection [16].
UQ Metrics (UQMs) are the UQ metrics defined in Section III-B, which includes *VR*, *SE*, *MI*, *TV*, and *PS*.
Robustness Score (RS) measures the robustness of each *SDM* in performing the sticker detection task. We calculate *RS* regarding *mAP* and *UQM*, respectively, i.e., RS_{mAP} and RS_{uq} by considering the model's performance on dataset $\mathcal{D}$ and $\mathcal{D}$'s adversarial version $\mathcal{D}_{adv}$. Specifically, for image $X \in \mathcal{D}$, there are 10 adversarial images $\{X^1_{adv}, X^2_{adv}, ..., X^{10}_{adv}\}$ from $\mathcal{D}_{adv}$, and we consider *RS* regarding metric $\mathcal{M}$ to be high when the model performs well in terms of $\mathcal{M}$ (i.e., high *mAP* or low *UQM*) and the difference in $\mathcal{M}$ between X and X's 10 adversarial examples is small. We then calculate RS_{mAP} as:

$$RS_{mAP} = Avg_{mAP} - Diff_{mAP}, \tag{14}$$

where Avg_{mAP} is the mean value of *mAP* calculated by averaging the value achieved on X and X's 10 adversarial examples: $(mAP_X + \sum_{i=1}^{10} mAP_{X^i_{adv}})/11$; $Diff_{mAP}$ measures how differently the model performs on X and X's 10 adversarial examples in terms of *mAP*: $\sum_{i=1}^{10} |mAP_X - mAP_{X^i_{adv}}|/10$. For RS_{uq}, we first calculate *RS* for each uncertainty metric $\mathcal{M}$:

$$RS_{\mathcal{M}} = 1 - (Avg_{\mathcal{M}} + Diff_{\mathcal{M}}), \tag{15}$$

where $\mathcal{M} \in \{VR, SE, MI, TV, PS\}$. $Avg_{\mathcal{M}}$ and $Diff_{\mathcal{M}}$ are calculated using the same equations as Avg_{mAP} and $Diff_{mAP}$. We then calculate RS_{uq} by taking the mean value of *RS* for each uncertainty metric:

$$RS_{uq} = \frac{\sum_{\mathcal{M} \in UQMs} RS_{\mathcal{M}}}{count(UQMs)}. \tag{16}$$

RS_{uq} combines the *RS* of the five *UQM*s to show the overall robustness of the *SDM*s in terms of uncertainty.

D. Research Questions

We answer the following four research questions (RQs): ***RQ1:*** How does each *SDM* perform when detecting stickers regarding detection accuracy? ***RQ2:*** How does each *SDM* perform when detecting stickers regarding prediction uncertainty? ***RQ3:*** How robust is each *SDM* in detecting stickers? ***RQ4:*** How does the detection accuracy correlate with the prediction uncertainty? Table II describes the employed datasets, metrics, and statistical tests for answering RQs.

E. Statistical Test

Since we compare more than two paired groups, i.e., *SDM*s using the same image dataset, we conduct the statistical analysis using the Friedman test [28], the Wilcoxon Signed-Rank test [29], and rank-biserial correlation [30]. Following the guideline [31], [32], we employ the Friedman test to verify if there are overall significant differences among all the paired groups (i.e., the models). If at least one group differs significantly from the others, $p < 0.05$ will be computed by this statistical test. If this is the case, we apply the Wilcoxon Signed-Rank test for pairwise comparisons (i.e., comparing two models) with a significance level of 5%. To interpret the Wilcoxon Signed-Rank test more precisely, we calculate the median and the standard deviation by the median absolute deviation (MAD) of our data, i.e., [median $\pm$ 2 MAD] [33], [34]. It checks if the underlying distribution is stochastically equal to, less than, or greater than a distribution symmetric about zero, and accordingly defines the relevant alternative hypothesis for more effective execution of the Wilcoxon Signed-Rank test. Moreover, for pairwise groups with significant differences, we use the rank-biserial correlation as the effect size of the Wilcoxon Signed-Rank test to determine which group shows better results. The rank-biserial correlation ranges from -1 to 1, where a positive result indicates that the first group tends to be larger than the second, and a negative result means the opposite, whereas 0 explains there is no significant difference between the two paired groups.

To study the correlation between *mAP* and *UQM*s for RQ4, we use Spearman's rank correlation coefficient [35], which provides the correlation coefficient (ρ) and the significance level (p). A p less than 0.05 indicates a significant correlation between *mAP* and *UQM*s, whereas ρ ranges from -1 to 1, revealing the direction and strength of the correlation. A positive ρ indicates a positive correlation, meaning that one variable increases as the other increases, and vice versa. A ρ of 0 shows no correlation. Considering that the Wilcoxon Signed-Rank test and Spearman's rank correlation coefficient are utilized multiple times to compare multiple paired groups, we address the issue of aggregated error probabilities resulting from multiple comparisons by applying the Holm–Bonferroni method [36]. This multiple comparison correction technique adjusts the significance level from the overall and controls the family-wise error rate at an α level, e.g., 5%.

V. Results and Analyses

We present descriptive statistics and summarize the key results of the statistical tests in Sections V-A to V-D, and then provide guidelines for selecting *SDM*s in Section V-E. The complete results are available in our replication package [37].

Funded by the European Union

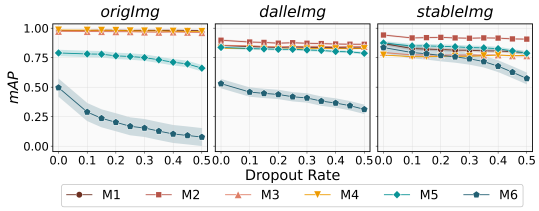


Fig. 3: Average *mAP* achieved by *SDM*s for *origImg*, *dalleImg*, and *stableImg* at different dropout rates. M1: *Faster R-CNN*, M2: *Faster R-CNN_v2*, M3: *RetinaNet*, M4: *RetinaNet_v2*, M5: *SSD300*, and M6: *SSDLite* – *RQ1*.

A. Results for RQ1 – Mean Average Precision

Figure 3 shows the *mAP* results achieved by each *SDM* for each dataset at different dropout rates with 95% confidence intervals. For the *origImg* dataset, *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2* all achieved *mAP* values close to 1 across all dropout rates, which indicates these four *SDM*s achieve high precision and accuracy in detecting stickers correctly, and their performance is stable as the dropout rate increases. Lower *mAP* values are observed for *SSD300* and *SSDLite*, with *SSDLite* consistently performing the worst. Moreover, the *mAP* values for both *SDM*s (*SSD300* and *SSDLite*) decrease as the dropout rate increases, indicating that they are more susceptible to the dropout rate. For *dalleImg* dataset, *Faster R-CNN_v2* achieved the consistent best performance at all dropout rates. *Faster R-CNN*, *RetinaNet*, and *RetinaNet_v2* performed comparably, which slightly outperformed *SSD300*. *SSDLite* performed consistently the worst. For *stableImg* dataset, we can observe that at all dropout rates, *Faster R-CNN_v2* performed the best followed by *SSD300* and *Faster R-CNN*. *SSDLite* outperformed *RetinaNet* and *RetinaNet_v2* at dropout rates below 0.2, after which the performance of *SSDLite* started to decrease significantly and underperformed *RetinaNet* and *RetinaNet_v2*, while *RetinaNet* and *RetinaNet_v2* performed consistently across all dropout rates. Besides, when comparing the *mAP* achieved by *SDM*s on LLM-synthetic datasets (*dalleImg* and *stableImg*) and *origImg*, we observe that *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2* perform poorer on the LLM-synthetic datasets than on *origImg* dataset, while *SSD300* and *SSDLite* exhibit better performance on the LLM-synthetic datasets than on *origImg* dataset.

We rank the best *SDM* that significantly outperformed the other *SDM*s on each metric according to the pairwise comparison results. For *SDM*s having no significant differences, we consider them to be a tie. Table III shows the results of best *SDM*(s) for each metric, where the last row shows the *SDM* recommendations based on different metrics. The best *SDM* for *mAP* is shown in the second column of Table III. We can observe that for *origImg* dataset, *Faster R-CNN*, *Faster R-CNN_v2*, and *RetinaNet_v2* all performed the best in terms of *mAP*, significantly outperforming the other *SDM*s (*RetinaNet*, *SSD300*, and *SSDLite*) across all the dropout rates. As for *dalleImg* and *stableImg* datasets, *Faster R-CNN_v2* significantly outperforms the other five *SDM*s. When looking at the best *SDM* across all datasets, *Faster R-CNN_v2* is recommended for all three datasets.

TABLE III: Best *SDM*(s) across different metrics and datasets for all dropout rates based on pairwise comparison results. M1: *Faster R-CNN*, M2: *Faster R-CNN_v2*, M3: *RetinaNet*, M4: *RetinaNet_v2*, M5: *SSD300*, and M6: *SSDLite*. Mi-j means that Mi, M$i+1$, ..., Mj all performed the best and had no significant differences – *RQ1*, *RQ2*, and *RQ3*.

Dataset	RQ1	RQ2					RQ3	
	mAP	*VR*	*SE*	*MI*	*TV*	*PS*	RS_{mAP}	RS_{uq}
origImg	M1,2,4	M1,2,5,6	M1,4	M4	M4	M4	M1,4	M4
dalleImg	M2	M1-6	M4	M4	M4	M4	M2	M4
stableImg	M2	M1-6	M4	M4	M4	M4	M2	M4
Rec_{metric}	M2	M1,2,5,6	M4	M4	M4	M4	M2	M4

Conclusion for RQ1: Different *SDM*s perform differently in terms of *mAP* on different datasets. Specifically, *Faster R-CNN*, *Faster R-CNN_v2*, and *RetinaNet_v2* all perform the best for *origImg* dataset, while *Faster R-CNN_v2* achieves the best performance for *dalleImg* and *stableImg* datasets. Regarding the best *SDM* across all datasets, *Faster R-CNN_v2* achieves the overall best performance regarding *mAP* and is recommended.

B. Results for RQ2 – Uncertainty

Figure 4 presents the mean results of *UQM*s for each dataset at different dropout rates with 95% confidence intervals.

Uncertainty for Label Classification. We have the following observations regarding three label classification *UQM*s (i.e., *VR*, *SE*, and *MI*). For *VR*, *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, *RetinaNet_v2*, *SSD300*, and *SSDLite* all achieve a very low *VR*, i.e., less than 0.015 for all three datasets at all dropout rates. The results suggest that all *SDM*s exhibit negligible uncertainty in label classification dispersion in the three datasets, that is, when an object is detected multiple times, the *SDM*s are confident that they will give the same classification result. However, for *origImg* dataset, we observe slightly higher *VR* values, which increase as the dropout rate increases for both *RetinaNet* and *RetinaNet_v2*, showing higher uncertainty and less stability of these two *SDM*s in terms of *VR* as dropout rate increases. As for *SE* and *MI*, *SSDLite* consistently achieves the highest uncertainty values of both *UQM*s for all datasets at all dropout rates, followed by *SSD300*, which exhibits high uncertainty values regarding these two *UQM*s for *origImg* and *dalleImg* datasets. However, exceptions can be observed for *stableImg* dataset, where *SSD300* shows lower *SE* and *MI* values than *Faster R-CNN* and *Faster R-CNN_v2* at low dropout rates and starts to show higher uncertainty as the dropout rate increases. Besides, *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2* achieve comparable performance in terms of *SE* and *MI* for *origImg* dataset at all dropout rates, while when it comes to *dalleImg* and *stableImg* datasets, *RetinaNet* and *RetinaNet_v2* consistently achieve the lowest *SE* and *MI* values. Recall that *SE* quantifies the amount of information in the predictions,

Funded by
the European Union

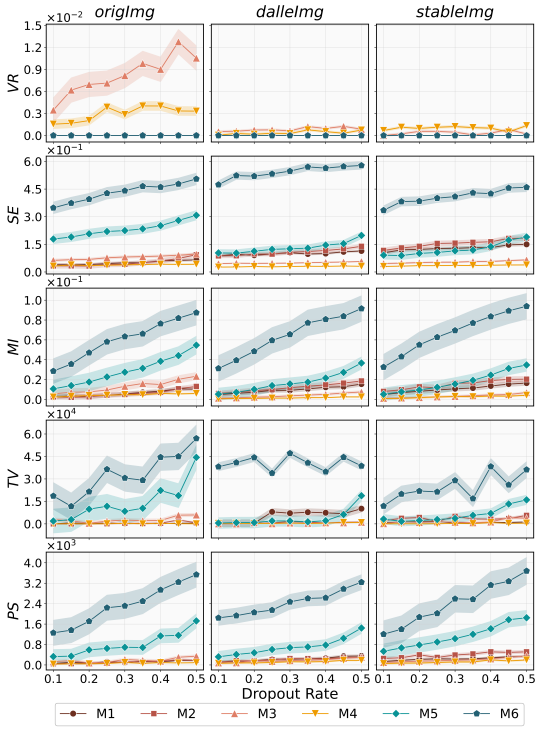


Fig. 4: Average *UQM*s achieved by *SDM*s for *origImg*, *dalleImg*, and *stableImg* at different dropout rates. M1: *Faster R-CNN*, M2: *Faster R-CNN_v2*, M3: *RetinaNet*, M4: *RetinaNet_v2*, M5: *SSD300*, and M6: *SSDLite* – *RQ2*.

while *MI* measures the difference in information between the prediction and the posterior. The results for *SE* and *MI* show that the prediction of *SSD300* and *SSDLite* contain more information and differ more from the posterior, thus exhibiting higher uncertainty in label classification as compared to the other four *SDM*s, i.e., *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2*.

Uncertainty for Bounding Box Regression. Similar observations can be made for the bounding box regression *UQM*s, that is, *SSD300* and *SSDLite* show higher *TV* and *PS* values for all datasets at all dropout rates except for *TV* on *dalleImg* dataset, where *SSD300* achieves lower *TV* than *Faster R-CNN* until the dropout rate reaches 0.45. Besides, *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2* achieve comparable and consistent low *TV* and *PS*, except for *TV* of *Faster R-CNN* on *dalleImg* dataset. The results of *TV* and *PS* show that *SSD300* and *SSDLite* exhibit higher uncertainty in predicting bounding boxes than the other four *SDM*s. Moreover, when the dropout rate increases, the uncertainties in *SSD300* and *SSDLite* also increase, while for the other *SDM*s, the dropout rate has little effect on their uncertainties.

Table III shows the best *SDM*(s), where we can find that

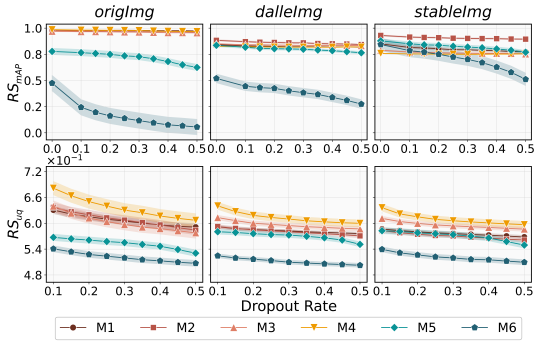


Fig. 5: Average RS_{mAP} and RS_{uq} achieved by *SDM*s for *origImg*, *dalleImg*, and *stableImg* at different dropout rates. M1: *Faster R-CNN*, M2: *Faster R-CNN_v2*, M3: *RetinaNet*, M4: *RetinaNet_v2*, M5: *SSD300*, and M6: *SSDLite* – *RQ3*.

regarding *VR*, *Faster R-CNN*, *Faster R-CNN_v2*, *SSD300*, and *SSDLite* achieve the best performance on *origImg* dataset, while all *SDM*s perform comparable on *dalleImg* and *stableImg* datasets. For *SE*, *MI*, *TV*, and *PS*, *RetinaNet_v2* perform the best, i.e., shows the lowest uncertainty, except for *SE* on *origImg* dataset, where *Faster R-CNN* and *RetinaNet_v2* achieve comparable best performance. When looking at the *SDM* recommendations across all datasets, regarding *VR*, *Faster R-CNN*, *Faster R-CNN_v2*, *SSD300*, and *SSDLite* are recommended, while for *SE*, *MI*, *TV*, and *PS*, *RetinaNet_v2* is recommended as the best *SDM*.

Conclusion for RQ2: Different *SDM*s show different levels of uncertainties for different *UQM*s. Regarding *VR*, *Faster R-CNN*, *Faster R-CNN_v2*, *SSD300*, and *SSDLite* are recommended as the best *SDM*s since they have the lowest uncertainties across all datasets, while for *SE*, *MI*, *TV*, and *PS*, *RetinaNet_v2* achieves the lowest uncertainties and is recommended as the best *SDM*.

C. Results for RQ3 – Robustness

Figure 5 shows the mean RS_{mAP} and RS_{uq} results with 95% confidence intervals. Regarding RS_{mAP}, we observe that *Faster R-CNN_v2* performs the best for *dalleImg* and *stableImg* datasets at all dropout rates, while for *origImg* dataset, *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2* all perform the best with a RS_{mAP} close to 1. *SSDLite* consistently performs the worst for *origImg* and *dalleImg* datasets at all dropout rates, and performs worst on *stableImg* when the dropout rate is above 0.2. Besides, the increase in dropout rate has different effects on different *SDM*s, for example, the dropout rate increase leads to a decrease of RS_{mAP} for *SSDLite*, while does not have much impact on *Faster R-CNN*, *Faster R-CNN_v2*, *RetinaNet*, and *RetinaNet_v2*. Regarding RS_{uq}, we observe that the results are consistent for all three datasets at all dropout rates, that is, *RetinaNet_v2* achieves the best RS_{uq}, while *SSDLite* performs the worst. We also see a decrease in RS_{uq} as the dropout rate increases. We also

Funded by
the European Union

TABLE IV: Results of the Spearman's rank correlation coefficient between *mAP* and *UQM*s for each *SDM* on all dropout rates and datasets. A value or *N/A* denotes that the correlation is not statistically significant or non-existent, otherwise, it is significant, i.e., $p < 0.05$ – *RQ4*.

SDM	*VR*	*SE*	*MI*	*TV*	*PS*
Faster R-CNN	N/A	-0.312	-0.277	-0.476	-0.461
Faster R-CNN_v2	N/A	-0.408	-0.357	-0.365	-0.334
RetinaNet	0.012	-0.161	-0.107	-0.328	-0.329
RetinaNet_v2	0.098	-0.209	-0.065	-0.316	-0.295
SSD300	N/A	-0.530	-0.455	-0.016	0.047
SSDLite	N/A	-0.510	-0.070	-0.269	-0.174

show the best *SDM*(s) in Table III. For RS_{mAP}, *Faster R-CNN* and *RetinaNet_v2* perform the best on *origImg* dataset, while *Faster R-CNN_v2* perform the best on *dalleImg* and *stableImg* datasets. When looking at the best *SDM* across all datasets, *Faster R-CNN_v2* is recommended. Regarding RS_{uq}, *RetinaNet_v2* performs consistently the best across all three datasets and is therefore recommended.

Conclusion for RQ3: The six *SDM*s perform differently for different datasets in terms of RS_{mAP} and RS_{uq}. Specifically, regarding RS_{mAP}, *Faster R-CNN* and *RetinaNet* achieve the best performance on *origImg* dataset, and *Faster R-CNN_v2* perform the best on *dalleImg* and *stableImg* datasets. Regarding RS_{uq}, *RetinaNet_v2* performs consistently the best across all three datasets.

D. Results for RQ4 – Correlation

Table IV shows the results of Spearman's rank correlation test for the correlation between *mAP* and the five *UQM*s for different *SDM*s. As for the correlation between *mAP* and *VR*, for all *SDM*s the difference is either insignificant or non-existent, while in terms of the correlation between *mAP* and the other four *UQM*s (*SE*, *MI*, *TV*, and *PS*), negative correlations can be observed for all *SDM*s, except for *SSD300*, which achieves insignificant correlation between *mAP* and *TV*/*PS*. This observation suggests that as uncertainty increases (except for *VR*), detection precision and accuracy will decrease. In addition, we also observe that as the dropout rate increases, the correlation between *mAP* and *SE*, *MI*, *TV*, and *PS* increases. This is reasonable since higher dropout rates will introduce higher uncertainties in the predictions.

Conclusion for RQ4: We observe significant negative correlations between *mAP* and *SE*, *MI*, *TV*, and *PS* for all *SDM*s, except for *TV* and *PS* achieved by *SSD300*, while insignificant or non-existent correlations are observed for the correlation between *mAP* and *VR*. Besides, the correlation between *mAP* and *SE*, *MI*, *TV*, and *PS* increases as the dropout rate increases.

E. Concluding Remarks and Guidelines

We analyze the performance of each *SDM* regarding detection accuracy (*mAP*), prediction uncertainty (*UQM*), and adversarial robustness (RS_{mAP} and RS_{uq}). Based on the results, we observe that different *SDM*s perform differently regarding different metrics. In practice, considering different purposes and applications, selecting the appropriate *SDM* based on specific performance requirements is crucial. Therefore, we provide guidelines for selecting *SDM*s from various perspectives in Figure 6. Regarding detection accuracy, i.e., *mAP*, *Faster R-CNN_v2* is recommended as it achieves the overall best performance. As for *UQM*s, for both label classification and bounding box regression tasks, we recommend *RetinaNet_v2* as it shows lower uncertainty in terms of both types of uncertainty metrics. For the robustness of *SDM*s, *Faster R-CNN_v2* shows the overall highest robustness regarding detection accuracy, i.e., RS_{mAP}, and is recommended as the best *SDM*. When considering robustness regarding uncertainty, *RetinaNet_v2* is recommended as the best *SDM*. Based on the guidelines, adaptive software integrated within the robot to select *SDM*s would significantly enhance the robot's decision-making capabilities, particularly in dynamic and uncertain environments. Such software can automatically select the most appropriate *SDM*s by considering various metrics.

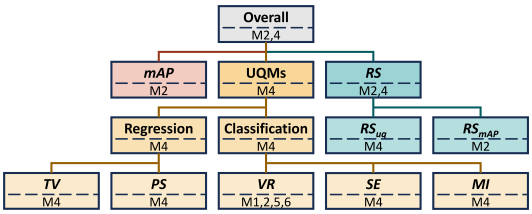


Fig. 6: Guidelines for Selecting Sticker Detection Models.

VI. Threats to Validity

Conclusion Validity concerns the reliability of the conclusions. We employed appropriate statistical tests to draw reliable conclusions and followed a rigorous statistical procedure to analyze the collected data. *Internal Validity* is related to the parameter settings. To determine the hyperparameters for the clustering algorithm, i.e., HDBSCAN, we ran different combinations of hyperparameter values on a small subset of the original dataset and selected the combination that correctly clustered all objects. Besides, the threshold value for IoU is set to 0.5, following Catak et al. [16]. *Constructive Validity* concerns the metrics used for the evaluation and we employed comparable metrics to ensure fair comparisons. Specifically, we used *mAP* to measure the sticker detection performance and five *UQM*s commonly used in classification and regression to measure model uncertainty. We also calculated *RS* to measure the robustness of the models under adversarial attacks. *External Validility* is about the generalizability of the empirical evaluation. We employ six models and six datasets. These models are based on different architectural designs and represent state-of-the-art object detection models. Regarding the benchmark datasets, in addition to the original dataset, we synthesize two datasets by prompting two VLMs, i.e., *DALL-E-3* and *Stable Diffusion-3*. Besides, we generate adversarial datasets by perturbing the original and synthetic datasets.

Funded by
the European Union

VII. Lessons Learned and Industrial Perspectives

Understanding, Utilizing, and Handling Uncertainty in Refurbishing Laptop software. UQ plays a pivotal role in enhancing the refurbishment of laptops by guiding laptop image data collection, ensuring dataset diversity, and determining the need for manual intervention. Collecting laptop images is time-consuming and significantly impacts solution quality, and UQ can guide decisions on whether additional data is needed. Specifically, UQ helps identify areas with high prediction uncertainty, indicating the need for more diverse and comprehensive training data. Besides, UQ is essential in addressing the limitations of existing datasets created by DTI, which primarily features Lenovo laptops and common sticker placements, e.g., stickers on the back cover. Also, images in the DTI dataset were captured in a relatively sterile setup with high quality that may not reflect real-world conditions. Thus, *SDM*s need to be tested under different conditions in handling unexpected situations, such as stickers near a mouse pad, where processor branding stickers are often found. By highlighting gaps in data diversity, UQ directs efforts towards collecting images from various laptop brands and unexpected sticker locations, ensuring *SDM*s are robust and adaptable to real-world conditions. Finally, UQ informs the need for manual intervention in the refurbishment process, enabling efficient and reliable operations by indicating when human oversight is required. In addition to the uncertainty in *SDM*s, the overall process of laptop refurbishing experiences uncertainties from various aspects, such as humans working in collaboration with robots, hardware errors, and other environmental factors. Hence, a holistic UQ method is required for the entire laptop refurbishing process that quantifies uncertainties from various aspects and eventually provides an overall uncertainty. Such quantified uncertainty plays a key role in identifying factors contributing to uncertainties, followed by devising best practices to reduce the overall uncertainty.

Realism of VLM-Generated Images. To assess the *SDM*s, we generated images of laptops with stickers using two commonly used pre-trained VLMs. However, this poses an open question of whether the images generated are realistic. In our context, all the generated images were manually checked for realism. However, such an approach is not scalable, especially when generating a large-scale dataset. To this end, we foresee the need for an automated method to generate realistic images of laptops with stickers. Pre-trained VLMs already possess a strong understanding of general image features, and domain-specific fine-tuning can further refine their output. Thus, one possible solution is to fine-tune VLMs to adapt to specific domains related to laptops and stickers, thereby helping produce more realistic images from the outset and reducing the need for extensive post-production filtering. Besides, large language models can be another way of realism evaluation [38]. Furthermore, training a realism classifier to distinguish realistic from unrealistic images can automate the manual check process. In the future, we plan to investigate scalable and efficient methods to generate large-scale realistic laptop image datasets.

VIII. Related Works

UQ in DNNs. UQ helps enhance model reliability and trustworthiness. Various UQ methods have been studied to measure confidence in the model predictions. BNN provides a probabilistic framework for DNNs through Bayesian inference [7]. It is systematic but computationally expensive and has a complex implementation. As a Bayesian approximation, MC-Dropout is a practical and widely used UQ method [3], which reduces the computational cost. Deep Ensembles (DE) [39] involves independently training multiple DNNs with different initialization, and the variance in the predictions of the multiple DNNs serves as an uncertainty measure. DE is robust and reliable in uncertainty estimation but computationally and memory-wise expensive, requiring training and storing multiple models. UQ methods have been applied in many domains, e.g., cyber-physical systems [40], [41], computer vision tasks [15], [42], [43], and heathcare [44]. In contrast, this paper adopts MC-dropout as the UQ method to quantify the uncertainty of sticker detection tasks in laptop refurbishing robots, thereby studying their real-world application in a new context.

Robustness Assessment of DNNs. Robustness measures DNNs' reliability. Recent robustness evaluation methods focus on adversarial robustness [45]–[47], which refers to the ability of DNNs to maintain performance and provide reliable outputs in the face of various perturbations, noises, and adversarial attacks. For instance, Carlini and Wagner [2] constructed three adversarial attack methods to evaluate the robustness of defensively distilled networks. Madry et al. [19] proposed a Projected Gradient Descent (PGD) attack to assess the adversarial robustness of DNNs from the robust optimization perspective. This paper employs a dense adversarial attack technique to measure adversarial robustness. Regarding the robustness metrics, in addition to measuring the robustness using the performance metric (i.e., *mAP*), we further calculate the robustness score concerning *UQM*s.

IX. Conclusions and Future Work

In this paper, we conduct an empirical study to evaluate the detection accuracy, prediction uncertainty, and adversarial robustness of six sticker detection models in the laptop refurbishing software. We adopt the Monte Carlo Dropout method to quantify the prediction uncertainty and measure uncertainty from two aspects: uncertainty in classification and regression. Besides, we present novel robustness metrics to evaluate the robustness of the models regarding detection accuracy and uncertainty. The results show *Faster R-CNN_v2* and *RetinaNet_v2* achieve the overall best performance regarding all metrics. Future works include quantifying uncertainties of the entire laptop refurbishing process and automatically generating realistic image datasets.

Acknowledgement

The work is supported by the RoboSapiens project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807. Jiahui Wu is also partially supported by the Co-tester project from the Research Council of Norway (No. 314544).

References

[1] E. Commission and D.-G. for Communication, *Circular economy action plan – For a cleaner and more competitive Europe*. Publications Office of the European Union, 2020.

[2] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*, pp. 39–57, Ieee, 2017.

[3] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, pp. 1050–1059, PMLR, 2016.

[4] C. Xie, J. Wang, Z. Zhang, Y. Zhou, L. Xie, and A. Yuille, "Adversarial examples for semantic segmentation and object detection," in *Proceedings of the IEEE international conference on computer vision*, pp. 1369–1378, 2017.

[5] J. Betker, G. Goh, L. Jing, T. Brooks, J. Wang, L. Li, L. Ouyang, J. Zhuang, J. Lee, Y. Guo, *et al.*, "Improving image generation with better captions," *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, vol. 2, no. 3, p. 8, 2023.

[6] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," 2021.

[7] D. Tran, M. Dusenberry, M. Van Der Wilk, and D. Hafner, "Bayesian layers: A module for neural network uncertainty," *Advances in neural information processing systems*, vol. 32, 2019.

[8] D. J. MacKay, "A practical bayesian framework for backpropagation networks," *Neural computation*, vol. 4, no. 3, pp. 448–472, 1992.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[10] J. M. Bernardo and A. F. Smith, *Bayesian theory*, vol. 405. John Wiley & Sons, 2009.

[11] R. J. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Pacific-Asia conference on knowledge discovery and data mining*, pp. 160–172, Springer, 2013.

[12] Y. Gal *et al.*, "Uncertainty in deep learning," 2016.

[13] L. C. Freeman, "Elementary applied statistics: for students in behavioral science," *(No Title)*, 1965.

[14] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[15] D. Feng, L. Rosenbaum, and K. Dietmayer, "Towards safe autonomous driving: Capture uncertainty in the deep neural network for lidar 3d vehicle detection," in *2018 21st international conference on intelligent transportation systems (ITSC)*, pp. 3266–3273, IEEE, 2018.

[16] F. O. Catak, T. Yue, and S. Ali, "Prediction surface uncertainty quantification in object detection models for autonomous driving," in *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pp. 93–100, IEEE, 2021.

[17] J. Zhang, J. Huang, S. Jin, and S. Lu, "Vision-language models for vision tasks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[18] Y. Li, H. Liu, Q. Wu, F. Mu, J. Yang, J. Gao, C. Li, and Y. J. Lee, "Gligen: Open-set grounded text-to-image generation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 22511–22521, 2023.

[19] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[20] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[21] Y. Li, S. Xie, X. Chen, P. Dollar, K. He, and R. Girshick, "Benchmarking detection transfer learning with vision transformers," *arXiv preprint arXiv:2111.11429*, 2021.

[22] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017.

[23] S. Zhang, C. Chi, Y. Yao, Z. Lei, and S. Z. Li, "Bridging the gap between anchor-based and anchor-free detection via adaptive training sample selection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 9759–9768, 2020.

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pp. 21–37, Springer, 2016.

[25] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1314–1324, 2019.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[27] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, pp. 303–338, 2010.

[28] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the american statistical association*, vol. 32, no. 200, pp. 675–701, 1937.

[29] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*, pp. 196–202, Springer, 1992.

[30] D. S. Kerby, "The simple difference formula: An approach to teaching nonparametric correlation," *Comprehensive Psychology*, vol. 3, pp. 11–IT, 2014.

[31] S. S. Mangiafico, "Summary and analysis of extension program evaluation in r," *Rutgers Cooperative Extension: New Brunswick, NJ, USA*, vol. 125, pp. 16–22, 2016.

[32] S. García, A. Fernández, J. Luengo, and F. Herrera, "Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power," *Information sciences*, vol. 180, no. 10, pp. 2044–2064, 2010.

[33] J. W. Tukey *et al.*, *Exploratory data analysis*, vol. 2. Springer, 1977.

[34] C. Reimann, P. Filzmoser, and R. G. Garrett, "Background and threshold: critical comparison of methods of determination," *Science of the total environment*, vol. 346, no. 1-3, pp. 1–16, 2005.

[35] C. Spearman, "The proof and measurement of association between two things.," 1961.

[36] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[37] ComplexSE, "Assessing the uncertainty and robustness of object detection models for detecting stickers on laptops." https://github.com/Simula-COMPLEX/laptop-sticker-uncertainty-robustness.

[38] J. Wu, C. Lu, A. Arrieta, T. Yue, and S. Ali, "Reality bites: Assessing the realism of driving scenarios with large language models," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pp. 40–51, 2024.

[39] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," *Advances in neural information processing systems*, vol. 30, 2017.

[40] Q. Xu, T. Yue, S. Ali, and M. Arratibel, "Pretrain, prompt, and transfer: Evolving digital twins for time-to-event analysis in cyber-physical systems," *IEEE Transactions on Software Engineering*, 2024.

[41] F. O. Catak, T. Yue, and S. Ali, "Uncertainty-aware prediction validator in deep learning models for cyber-physical system data," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–31, 2022.

[42] S. Su, Y. Li, S. He, S. Han, C. Feng, C. Ding, and F. Miao, "Uncertainty quantification of collaborative detection for self-driving," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5588–5594, IEEE, 2023.

[43] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?," *Advances in neural information processing systems*, vol. 30, 2017.

[44] C. Lu, Q. Xu, T. Yue, S. Ali, T. Schwitalla, and J. F. Nygård, "Evoclinical: Evolving cyber-cyber digital twin with active transfer learning for automated cancer registry system," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1973–1984, 2023.

[45] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *arXiv preprint arXiv:1902.06705*, 2019.

[46] W. Brendel, J. Rauber, M. Kümmerer, I. Ustyuzhaninov, and M. Bethge, "Accurate, reliable and fast robustness evaluation," *Advances in neural information processing systems*, vol. 32, 2019.

[47] F. Croce and M. Hein, "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks," in *International conference on machine learning*, pp. 2206–2216, PMLR, 2020.

Funded by the European Union

F IsaCircus model for the MakePlan state of the navigation robot

Listing 6: Trans model of *MakePlan* for the navigation robot

```
datatype NIDS_MakePlan_Adaptation_Plan =
       NID_i0_MakePlan_Adaptation_Plan |
       NID_CalculateRotations_MakePlan_Adaptation_Plan |
       NID_PlanEmptyRotation_MakePlan_Adaptation_Plan |
       NID_PlanPositiveRotation_MakePlan_Adaptation_Plan |
       NID_PlanNegativeRotation_MakePlan_Adaptation_Plan |
       NID_PlanFullRotation_MakePlan_Adaptation_Plan |
       NID_PlanPositiveFirstRotation_MakePlan_Adaptation_Plan |
       NID_PlanNegativeFirstRotation_MakePlan_Adaptation_Plan |
       NID_f0_MakePlan_Adaptation_Plan

record SpinConfig =
       commands :: "(SpinCommand list)"
       period :: "int"
record SpinCommand =
       angleVelocity :: "real"
       duration :: "real"
record LidarRange =
       angleIncrement :: "real"
       ranges :: "(real list)"
record BoolLidarMask =
       values :: "(bool list)"
       baseAngle :: "real"
record ProbLidarMask =
       values :: "(real list)"
       baseAngle :: "real"

chantype mychan =
share :: unit |
terminate :: unit |
internal__MakePlan_Adaptation_Plan :: "NIDS_MakePlan_Adaptation_Plan" |
exited_MakePlan_Adaptation_Plan :: unit |
exit_MakePlan_Adaptation_Plan :: unit |
"get_minMaxRotation" :: "real ×real" |
enter_i0_MakePlan_Adaptation_Plan :: unit |
enter_CalculateRotations_MakePlan_Adaptation_Plan :: unit |
enter_PlanEmptyRotation_MakePlan_Adaptation_Plan :: unit |
enter_PlanPositiveRotation_MakePlan_Adaptation_Plan :: unit |
enter_PlanNegativeRotation_MakePlan_Adaptation_Plan :: unit |
enter_PlanFullRotation_MakePlan_Adaptation_Plan :: unit |
enter_PlanPositiveFirstRotation_MakePlan_Adaptation_Plan :: unit |
enter_PlanNegativeFirstRotation_MakePlan_Adaptation_Plan :: unit |
enter_f0_MakePlan_Adaptation_Plan :: unit |
entered_f0_MakePlan_Adaptation_Plan :: unit |
aviol :: unit |
gviol :: unit
```

Funded by
the European Union

```
47
48 abbreviation "assume b Q P ≡(if b then P else aviol →Q)"
49 abbreviation "guar b P ≡(if b then P else gviol →STOP)"
50
51 actions is "(mychan, unit) action" where
52 "SSTOP = share →SSTOP" |
53 " Trans_MakePlan_Adaptation_Plan(n) =
54 ((SSTOP △(get_minMaxRotation?minMaxRotation →(assume (fst minMaxRotation ≤snd
       minMaxRotation) (Trans_MakePlan_Adaptation_Plan(n)) ((((((((((((
55 ((n = NID_i0_MakePlan_Adaptation_Plan) &  (((internal__MakePlan_Adaptation_Plan.
       NID_i0_MakePlan_Adaptation_Plan →Skip);; (
       enter_CalculateRotations_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_CalculateRotations_MakePlan_Adaptation_Plan))
       )))
56 □
57 ((n = NID_CalculateRotations_MakePlan_Adaptation_Plan) &  (((((fstminMaxRotation) =
       (sndminMaxRotation))) &  (((internal__MakePlan_Adaptation_Plan.
       NID_CalculateRotations_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (
       enter_PlanEmptyRotation_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_PlanEmptyRotation_MakePlan_Adaptation_Plan)))
       ))))))))
58 □
59 ((n = NID_CalculateRotations_MakePlan_Adaptation_Plan) &  ((((((fstminMaxRotation) >
        0) ∧((sndminMaxRotation) > 0))) &  (((internal__MakePlan_Adaptation_Plan.
       NID_CalculateRotations_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (
       enter_PlanPositiveRotation_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_PlanPositiveRotation_MakePlan_Adaptation_Plan
       ))))))))))
60 □
61 ((n = NID_CalculateRotations_MakePlan_Adaptation_Plan) &  ((((((fstminMaxRotation) <
        0) ∧((sndminMaxRotation) < 0))) &  (((internal__MakePlan_Adaptation_Plan.
       NID_CalculateRotations_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (
       enter_PlanNegativeRotation_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_PlanNegativeRotation_MakePlan_Adaptation_Plan
       ))))))))))
62 □
63 ((n = NID_CalculateRotations_MakePlan_Adaptation_Plan) &  (((((((fstminMaxRotation)
       < 0) ∧((sndminMaxRotation) > 0)) ∧(((sndminMaxRotation) - (fstminMaxRotation)) >
        1))) &  (((internal__MakePlan_Adaptation_Plan.
       NID_CalculateRotations_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (
       enter_PlanFullRotation_MakePlan_Adaptation_Plan →Trans_MakePlan_Adaptation_Plan
       (NID_PlanFullRotation_MakePlan_Adaptation_Plan)))))))))))
64 □
```

Funded by
the European Union

```
65 ((n = NID_CalculateRotations_MakePlan_Adaptation_Plan) &  (((((((fstminMaxRotation)
       < 0) ∧((sndminMaxRotation) > 0)) ∧((-(fstminMaxRotation)) < (sndminMaxRotation))
       )) &  (((internal__MakePlan_Adaptation_Plan.
       NID_CalculateRotations_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (
       enter_PlanPositiveFirstRotation_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(
       NID_PlanPositiveFirstRotation_MakePlan_Adaptation_Plan)))))))))))
66 □
67 ((n = NID_CalculateRotations_MakePlan_Adaptation_Plan) &  (((((((fstminMaxRotation)
       < 0) ∧((sndminMaxRotation) > 0)) ∧((-(fstminMaxRotation)) ≥(sndminMaxRotation))
       )) &  (((internal__MakePlan_Adaptation_Plan.
       NID_CalculateRotations_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (
       enter_PlanNegativeFirstRotation_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(
       NID_PlanNegativeFirstRotation_MakePlan_Adaptation_Plan)))))))))))
68 □
69 ((n = NID_PlanEmptyRotation_MakePlan_Adaptation_Plan) &  (((
       internal__MakePlan_Adaptation_Plan.
       NID_PlanEmptyRotation_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (enter_f0_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_f0_MakePlan_Adaptation_Plan)))))))))
70 □
71 ((n = NID_PlanPositiveRotation_MakePlan_Adaptation_Plan) &  (((
       internal__MakePlan_Adaptation_Plan.
       NID_PlanPositiveRotation_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (enter_f0_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_f0_MakePlan_Adaptation_Plan)))))))))
72 □
73 ((n = NID_PlanNegativeRotation_MakePlan_Adaptation_Plan) &  (((
       internal__MakePlan_Adaptation_Plan.
       NID_PlanNegativeRotation_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (enter_f0_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_f0_MakePlan_Adaptation_Plan)))))))))
74 □
75 ((n = NID_PlanFullRotation_MakePlan_Adaptation_Plan) &  (((
       internal__MakePlan_Adaptation_Plan.NID_PlanFullRotation_MakePlan_Adaptation_Plan
        →Skip);; ((SSTOP △(exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
       exited_MakePlan_Adaptation_Plan →Skip);; (enter_f0_MakePlan_Adaptation_Plan →
       Trans_MakePlan_Adaptation_Plan(NID_f0_MakePlan_Adaptation_Plan)))))))))
76 □
77 ((n = NID_PlanPositiveFirstRotation_MakePlan_Adaptation_Plan) &  (((
       internal__MakePlan_Adaptation_Plan.
       NID_PlanPositiveFirstRotation_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
       exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
```

Funded by
the European Union

```
    exited_MakePlan_Adaptation_Plan →Skip);; (enter_f0_MakePlan_Adaptation_Plan →
    Trans_MakePlan_Adaptation_Plan(NID_f0_MakePlan_Adaptation_Plan)))))))))
□
((n = NID_PlanNegativeFirstRotation_MakePlan_Adaptation_Plan) &  (((
    internal__MakePlan_Adaptation_Plan.
    NID_PlanNegativeFirstRotation_MakePlan_Adaptation_Plan →Skip);; ((SSTOP △(
    exit_MakePlan_Adaptation_Plan →Skip));; (SSTOP △((
    exited_MakePlan_Adaptation_Plan →Skip);; (enter_f0_MakePlan_Adaptation_Plan →
    Trans_MakePlan_Adaptation_Plan(NID_f0_MakePlan_Adaptation_Plan)))))))))
□
((n = NID_f0_MakePlan_Adaptation_Plan) &  (terminate →Skip))
))))))) "
```

Funded by
the European Union