R5-COP

Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems

Assessment of the On-line Verification and Incremental Testing

BME

Project	R5-COP	**Grant agreement no.**	621447
Deliverable	D34.50	**Date**	31/01/2017
Contact Person	Istvan Majzik	**Organisation**	BME
E-Mail	majzik@mit.bme.hu	**Diss. Level**	PU

Document History			
Ver.	**Date**	**Changes**	**Author**
0.1	18/11/2016	Initial structure of the content	I. Majzik (BME)
0.2	15/01/2017	Integration of the overview of on-line verification	I Majzik, A. Vörös and others (BME)
0.3	17/01/2017	Integration of the chapter on on-line V&V using SIL methodology	J. Bicevskis, A. Gaujens (IMCS)
0.4	23/01/2017	Integration of the SWOT analysis for on-line verification	I. Majzik (BME)
0.5	24/01/2017	Integration of the chapter about incremental testing	Z. Micskei, D. Honfi (BME)
0.6	25/01/2017	Additional parts about performance evaluation	A. Vörös, D. Honfi, I. Majzik, Z. Micskei and others (BME)
0.7	31/01/2017	Corrections and extensions	I. Majzik, Z. Micskei, A. Vörös (BME)
1.0	03/02/2017	Version for internal review	I. Majzik (BME)
1.1	15/02/2017	Corrections after internal review	M. Spisländer (FAU), I. Majzik (BME)

Note: Filename should be

"R5-COP_D##_#.doc", e.g. „R5-COP_D91.1_v0.1_TUBS.doc"

Fields are defined as follow

1. Deliverable number ***.***

2. Revision number:

draft version **v**

approved **a**

version sequence (two digits) ***.***

3. Company identification (Partner acronym) *****

Content

List of Figures

List of Acronyms

CaTL	Context-aware Timed Propositional Linear Temporal Logic
CTL	Computational Tree Logic
EB	Evaluation Block
EMF	Eclipse Modelling Framework
LSC	Live Sequence Chart
LTL	Linear Temporal Logic
MSC	Message Sequence Chart
OCL	Object Constraint Language
PLTL	Propositional Linear Temporal Logic
PSL	Property Specification Language
R3-COP	Resilient Reasoning Robotic Cooperative Systems
ROS	Robot Operating System
UML	Unified Modelling Language

1 Introduction

1.1 Summary (abstract)

WP34 of R5-COP aims at supporting the off-line and on-line verification of the behaviour of R5-COP systems by elaborating methods and tools for incremental testing and runtime monitoring. Incremental testing focuses on checking the permanent effects of reconfiguration on basic safety and robustness properties, while runtime monitoring focuses also on checking the effects of runtime errors.

Incremental testing of the behaviour is relevant in the design phase and in maintenance phases (to check the behaviour of a changed or reconfigured version), utilizing existing test suites. Runtime monitoring addresses the detection of errors and malfunctions that manifest themselves in runtime, e.g., due to random hardware faults, configuration faults, operator faults, faults in adaptation and self-healing.

To support these activities, in the previous tasks and deliverables the following activities were performed and documented:

- Description languages were developed to capture those properties of the system that characterise its correct behaviour.
- Algorithms and tools were developed for monitor synthesis on the basis of the described properties.
- Algorithms and tools were developed for test classification and test selection in incremental testing on the basis of changed requirements, context or configuration.

The topic of this deliverable is the assessment of on-line verification and incremental testing. In case of on-line verification, the usability and efficiency of the monitoring infrastructure (supporting tools) is assessed and the application is evaluated. In case of incremental testing, the selection and generation of new tests is evaluated. The assessment is completed by the analysis of standardization aspects.

1.2 Purpose of document

This deliverable aims at the assessment of the methods and tools elaborated in the previous tasks of WP34. Since the R5-COP demonstrator applications are presented in their corresponding deliverables (in SP4), this report will not focus on demonstrator environments and uses cases but mainly on the generic properties and capabilities of the monitoring and incremental testing methods and tools, presenting demonstrator applications as examples.

1.3 Partners involved

Partners and Contribution	
Short Name	**Contribution**
BME	Assessment activities
FAU	Review of the document
IMCS	Assessment activities
PIAP	Integration and evaluation in case of the Scout robot

2 The Assessment Approach

The assessment is performed on the following methods:

- On-line verification (Section 3),
- On-line V&V using SIL methodology (Section 4),
- Incremental testing (Section 5).

The assessment includes the following aspects:

- Summary of the method and its novelties.
- SWOT analysis that describes the strengths, weaknesses, opportunities and threats of the method and its application.
- Evaluation of capabilities and efficiency.

According to this approach, separate sections are devoted to the three methods and related subsections for the assessment aspects.

The deliverable is closed by Section 6 that discusses the role of these methods according to the development standards of safety critical systems.

2.1 The SWOT Analysis Method

As a specific step in the assessment, we adopted a so-called *expert evaluation approach* to judge the capabilities of the developed new V&V methodologies. In particular, a SWOT-based analysis was performed to identify the different helpful or harmful factors affecting the newly developed method and tools. The factors were identified by the members of the team who developed and applied the new techniques and tools.

The SWOT (Strengths, Weaknesses, Opportunities, and Threats) method [24] is a method developed for strategic business planning that analyses the external and internal factors affecting a company, a department or an actual product. The factors are categorized as helpful or harmful ones, and depending on whether they are external or internal ones, they are listed as

- Strengths: helpful, internal,
- Weaknesses: harmful, internal,
- Opportunities: helpful, external,
- Threats: harmful, external.

The collected factors are typically aligned in a 2x2 matrix to visualize the results of the SWOT analysis.

3 Assessment of On-line Verification

The topic of this section is the assessment of the on-line verification method developed in WP34. On-line (runtime) verification aims at checking system execution against formally specified behavioural properties. The development of on-line verification methods typically addresses the definition of description languages for specifying the properties to be monitored (see in deliverable D34.10), the corresponding checking algorithms (see in D34.31), the required instrumentation for accessing observations necessary for checking, and the development of the related tool environment (see in D34.32).

3.1 Summary of the Method and its Novelties

The monitoring infrastructure (method and tool support) that was developed allows automated construction of monitor components by the synthesis of their source code. In this subsection the concept, the tool support, and the main novelties are summarized.

3.1.1 The Concept of the Monitoring Infrastructure

The concept of the monitoring infrastructure is presented in Figure 1.

The monitors perform online verification by observing the behaviour of the robot components (i.e., the trace of their events, actions, and the perceived context) to detect the hazardous situations and trigger a reaction (e.g., to stop the robot to maintain safety). The potential hazardous situations (e.g., the sequence of events and interactions among components) are specified using a high-level language: state machine diagram, sequence diagram, or temporal patterns. Accordingly, three tool-chains were developed that generate the source code on the basis of this specification automatically. These tool-chains offer a flexible framework to specify properties according to the focus and level of completeness of the behaviour to be checked, and effectively support the generation of the monitor components:

- *Monitor synthesis on the basis of behaviour specification*, using UML2 statecharts extended with timeouts and context/configuration related events. This tool is especially useful when the designer wants to specify complete reference behaviour. The monitor is responsible to detect and signal any behaviour that is different from this reference behaviour considering the sequence of input events.
- *Monitor synthesis on the basis of scenario specification,* using UML2 sequence diagrams, extended with timing and context/configuration dependency. This tool is useful when the designer wants to specify conditions (including context/configuration fragments) and the related required or forbidden sequence of input events and output actions. The monitor is responsible for matching the observed behaviour with the condition part of the scenario and detect if subsequently the required behaviour is missing or the forbidden behaviour occurs. The behaviours that do not match the condition part are not checked by the monitor, this way the focus of monitoring is only on the specified scenarios.
- *Monitor synthesis on the basis of temporal specification,* using a library of extensible safety and liveness behaviour patterns. This tool is useful when a declarative specification of properties is needed (especially in case of invariant properties that shall be always satisfied to guarantee safe operation). The monitor is responsible for detecting an error when the sequence of observed events does not satisfy the temporal property. All behaviours are checked (there is no explicit condition part in the properties) but focusing only on the events that are included in the specified property.

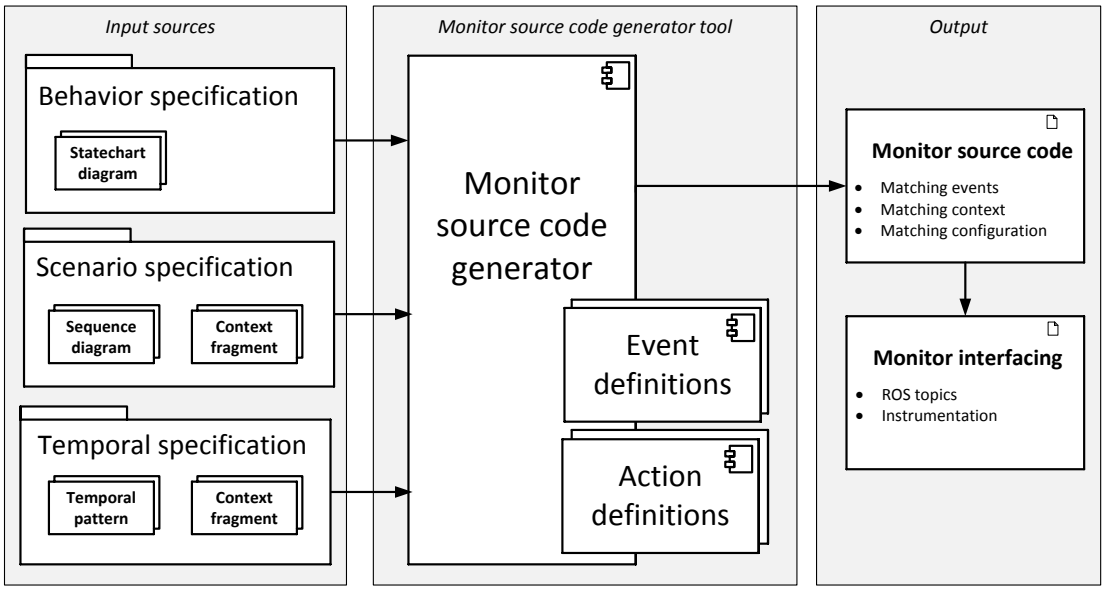

Figure 1. The monitoring infrastructure

3.1.2 The Tool Support

Figure 2 summarizes the tools and steps in the monitor development which will be detailed in the following subsections.

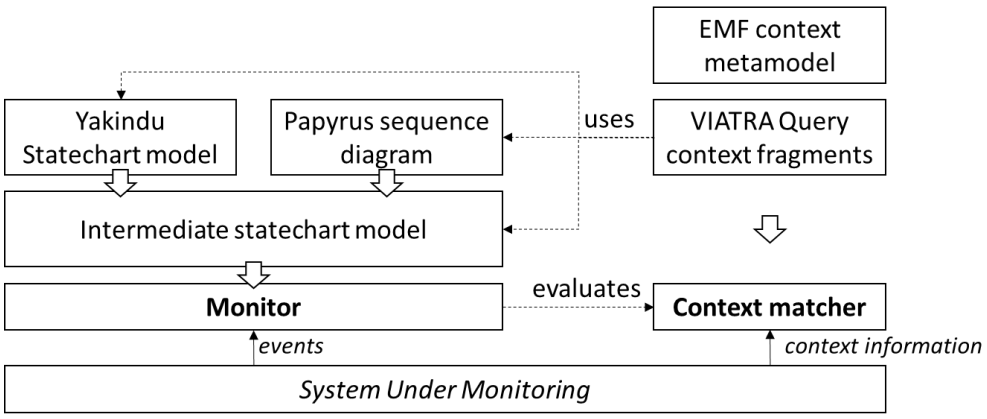

Figure 2. Overview of monitor development process

Statechart-based monitor generation

Our approach supports monitor source code generation from statechart based property specification. The developer can design the model using the popular Yakindu Statecharts Tool (Figure 3).

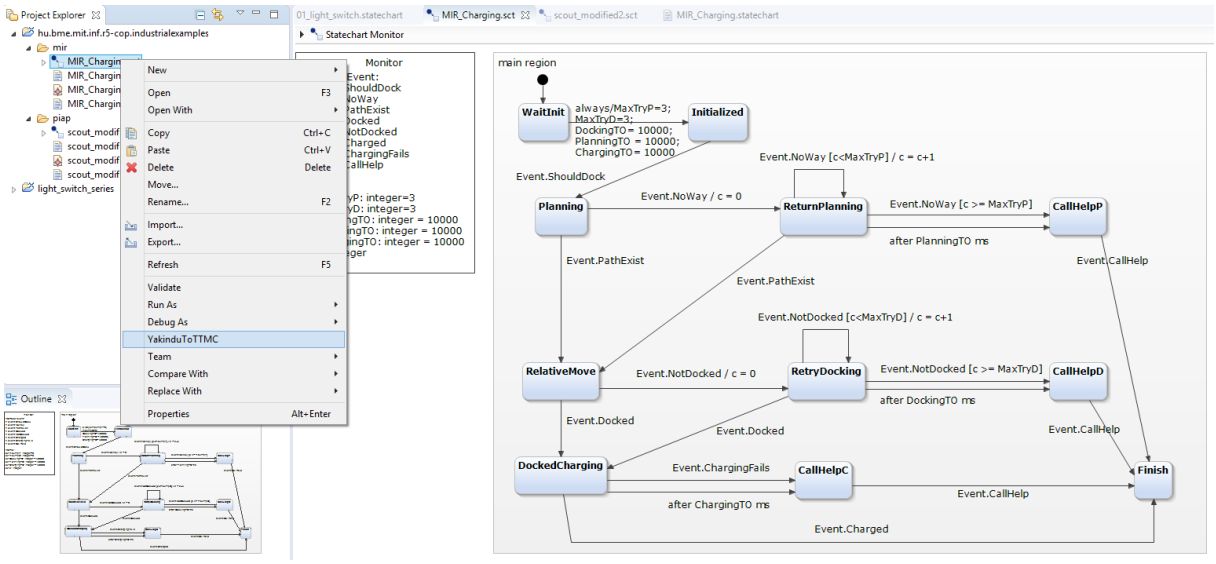

Figure 3. Specification of the monitored property as a statechart in the Yakindu tool

By right clicking on the diagram, the context menu first offers the generation of the so-called intermediate statechart model. Note that this representation (Figure 4) can also be used to design the monitor directly when the designer prefers textual modelling. We provided an Eclipse-based editor to support the developer to construct/edit this intermediate model. The editor provides syntax highlighting and content assist.

MIR_Charging.statechart | par.di | opt.di | alt.di | robot_turning.cpp | par.statechart

```
specification Monitor {
    signal ShouldDock
    signal NoWay
    signal PathExist
    signal ChargingFails
    signal NotDocked
    signal Docked
    signal CallHelp
    signal Charged

    statechart MonitorDeclaration := {
        var DockingTO : integer := 10000
        var c : integer
        var MaxTryD : integer := 3
        var MaxTryP : integer := 3
        var PlanningTO : integer := 10000
        var ChargingTO : integer := 10000
        timeout RetryDocking
        timeout DockedCharging
        timeout ReturnPlanning
            transition from CallHelpC to Finish upon CallHelp
            transition from ReturnPlanning to ReturnPlanning upon NoWay [ c < MaxTryP ]
            /
                assign c := c + 1
            transition from ReturnPlanning to RelativeMove upon PathExist
            transition from DockedCharging to CallHelpC upon ChargingFails
            transition from WaitInit to Initialized /
                assign MaxTryD := 3;
                assign MaxTryP := 3;
                assign ChargingTO := 10000;
                assign PlanningTO := 10000;
                assign DockingTO := 10000
            transition from ReturnPlanning to CallHelpP when ReturnPlanning
            transition from RetryDocking to RetryDocking upon NotDocked [ c < MaxTryD ]
            /
                assign c := c + 1
            transition from ReturnPlanning to CallHelpP upon NoWay [ c >= MaxTryP ]
            transition from Planning to ReturnPlanning upon NoWay /
                assign c := 0
            transition from RelativeMove to DockedCharging upon Docked
            transition from DockedCharging to Finish upon Charged
            transition from CallHelpD to Finish upon CallHelp
            transition from Planning to RelativeMove upon PathExist
            transition from Initialized to Planning upon ShouldDock
            transition from RelativeMove to RetryDocking upon NotDocked /
                assign c := 0
            transition from CallHelpP to Finish upon CallHelp
            transition from Entry0 to WaitInit
```

Figure 4. Intermediate statechart model

In addition to the generation of the textual intermediate model (with extension *.statechart*), this step also produces the EMF-based model (with extension *.statechartmodel*) and traceability links (with extension *.y2ttraceability*) related to the statechart diagram to support backtracking. The resulting files (in case of the *MIR_Charging* statechart specification) are shown in Figure 5. The model generation steps from the statechart diagram to the intermediate model and the EMF-based model are implemented with the help of precise model transformations.

Figure 5. The set of input and generated models

The intermediate language is based on an *XText* grammar and has precise semantics (in the same way as the statechart diagram). It is the direct input of the monitor source code generation that can also be started from the context menu of the tool.

In case of the Yakindu Statechart based modelling, the designer can use all statechart elements except priorities and parameterized events. Pseudo states, hierarchy, parallel regions, user defined guard functions are supported by the transformations, model validations and the monitor generation algorithm.

Sequence diagram based monitor generation

A rich subset of sequence diagrams can also be used to specify the property to be monitored. The sequence diagrams can be constructed in the Papyrus tool, which is a widely used open-source UML model editor (Figure 6).

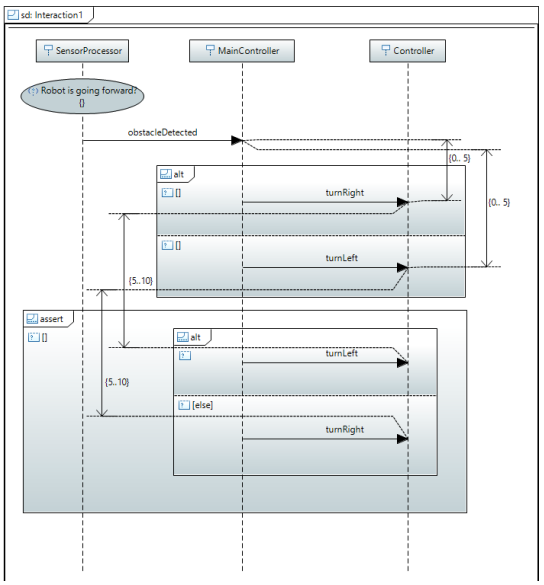

Figure 6. Specification of the monitored property as sequence diagram in the Papyrus tool

From the sequence diagram based monitor specification a built-in transformation generates the intermediate statechart model (the same intermediate model that was used in case of statechart diagram based property specifications), from which an additional step generates the source code of the monitor component.

Besides modelling events and actions, the following elements of sequence diagrams are supported: alternative, loop, and assert fragments, and guard expressions. Timing aspects can be expressed as time intervals with minimum and maximum durations.

Context and configuration description based generation of guard functions

The statechart and sequence diagram models can refer to so-called guard functions that specify in which context an event is acceptable. The source code of these guard functions is generated by a separate tool. Its inputs are a *context metamodel* (that specifies the artefacts in the context of the robot) and *context patterns* that give the context configurations that shall be matched by the observed context to have a true guard. The context metamodel can be specified using structure modelling in form of EMF metamodel, while patterns can be specified using a query language (the VIATRA[1] Query language).

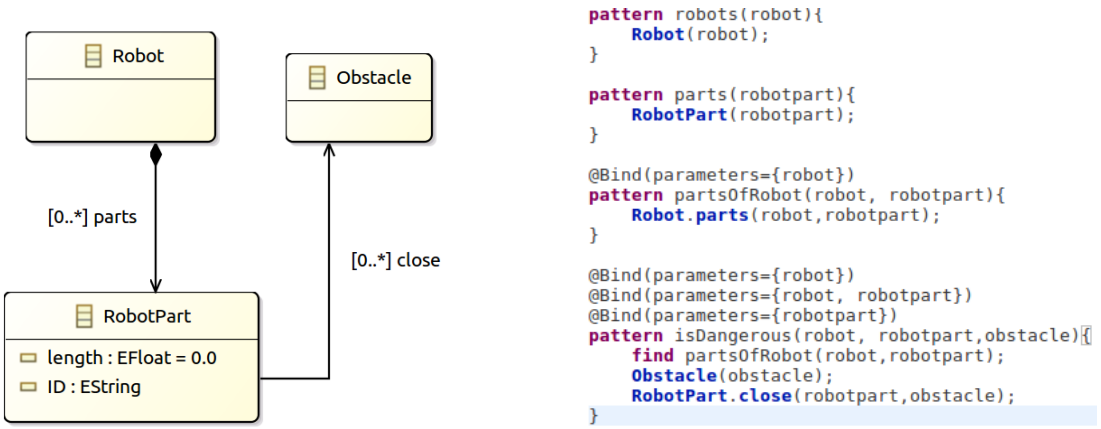

Figure 7. Context model and related patterns

The tool is presented by an example depicted in Figure 7.

- The metamodel of the context that specifies the types of objects in the context and their relations is given by a metamodel constructed in the EMF editor (left part of Figure 7) as a class diagram.
- The patterns to be matched that refer to the elements (instances) of the context metamodel are specified in a textual form using the VIATRA Query Language (right part of Figure 7). Various features are supported such as embedded graph patterns, transitive closure, expressions and type constraints.

From the context metamodel and the query patterns, the source code of the corresponding guard functions are generated with the help of VIATRA. The generated guard function (to be called by the monitor) is responsible for matching the context pattern with the observed context (where the sensors shall update the data structure generated on the basis of the context metamodel). As mentioned above, these guard functions can be referred to both in statechart models and in sequence diagram specifications.

Configuration dependency is handled similarly: instead of the context metamodel the configuration metamodel is used, while the patterns refer to configurations in which the events are acceptable. The guard function implements the matching between the configuration patterns and the observed configuration (the corresponding data structure generated on the basis of the configuration metamodel shall be updated during reconfiguration).

[1] http://www.eclipse.org/viatra/

3.1.3 The Main Novelties

In this section we highlight three main novelties of the monitoring infrastructure.

- *Supporting of engineering languages*. High level engineering languages are supported through existing and widely used modelling tools as inputs of the monitor synthesis (as presented in the preceding section). This way, instead of low-level mathematical formalisms, the engineers are provided property specification languages they are familiar with. The Yakindu Statechart tools, Papyrus and Eclipse Modelling Framework support the creation, editing, persisting and loading of the models in a standard way. The monitor synthesis is integrated into these modelling tools as an additional context menu item.
- *Monitoring the timing aspects*. In case of statechart diagrams (that are intended to specify complete behaviour) *time dependent behaviour* can be specified using transitions triggered by timeout events (in case of the statechart diagram presented in Figure 8, the transition labelled with "*after PlanningTO ms*" specifies a timeout where *PlanningTO* is the constant representing the timeout for the planning activity). In case of sequence diagrams (that are intended to specify conditional scenarios of event sequences) *time constraints* can be given in the form of time intervals with minimum and maximum durations between events (in case of the sequence diagram presented in Figure 9, the time constraints are given directly as durations, e.g., "*(5..10)"*). This is a user-friendly and integrated way of describing timing aspects to be checked by the monitor.
- Supporting the on-line verification of context and configuration dependency. Model based specification of the context elements (in the form of a context metamodel) and query based specification of context patterns is offered. An example is presented in Figure 7. These specifications are used to generate the source code of guard functions that perform the matching between the context patterns and the context observed by the robot (updated by its sensors). Configuration dependency is handled similarly. This way the engineer does not have to deal with the implementation of complex graph matching functionality,

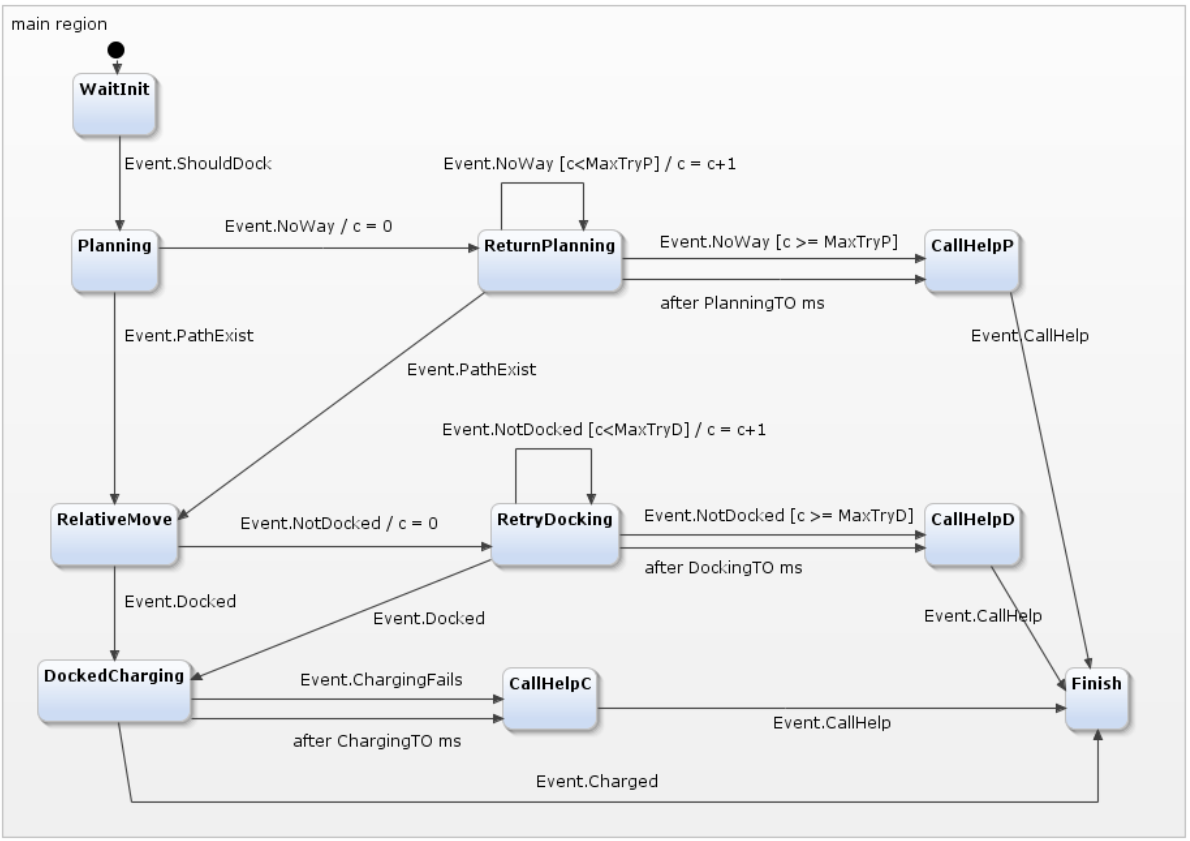

Figure 8. Timeout specified in a statechart diagram

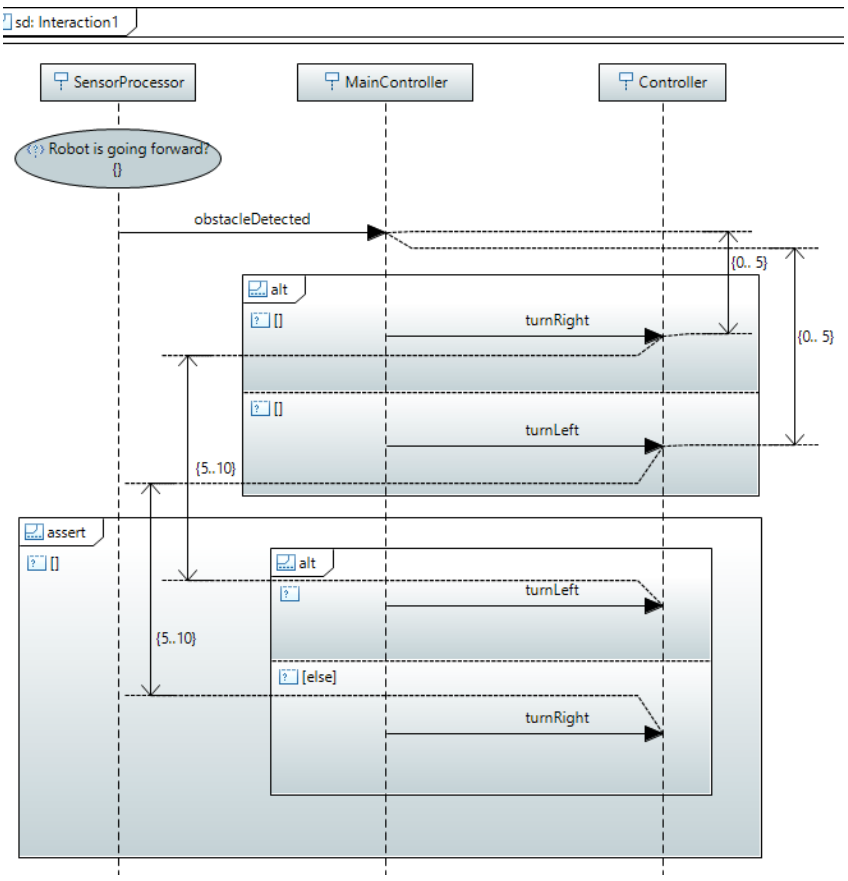

Figure 9. Time constraints specified in a sequence diagram

3.2 SWOT Analysis

The strengths, weaknesses, opportunities and threats of the on-line verification infrastructure are summarized in Figure 10 and detailed in the following subsections.

	Helpful	Harmful
Internal	• Precise property specification • Monitoring of timing • Monitoring context- and configuration-dependent behaviour • Systematic design of monitors with different strategies • Automatic tools • Independent monitoring **Strengths**	• Detailed semi-formal specification is needed • Limits of scenarios as properties • Overhead of monitoring • Separate tool for specifying context and configuration dependency **Weaknesses**
External	**Opportunities** • Complexity of classic V&V • V&V challenges in checking context-aware behaviour	**Threats** • Model-based design is not widespread • Testing and debugging of monitor components are difficult

Figure 10. The SWOT table of the on-line verification framework

3.2.1 Strengths

- *Precise property specification*: The properties to be monitored are specified in a precise way, using semi-formal engineering languages: statecharts, sequence diagrams based scenarios and temporal patterns. We assigned semantics to these languages, this way allowing the precise and systematic synthesis of the source code of the monitor components, It can be emphasized, however, that these languages are close to the engineering practice and does not mean “cryptic” formal mathematical languages like low-level temporal logics.
- *Monitoring of timing*: The property specification languages mentioned in the previous point are extended with language elements that allow the specification of timing. In case of statecharts, timeout events can be used; in case of sequence diagram based scenarios time intervals can be specified, while in case of temporal patterns clock variables can be introduced.
- *Monitoring context- and configuration-dependent behaviour*: The property specification languages are extended with the concept of guards that can be used to specify in which case an event is acceptable or not. The source code of the guard can be generated by a separate tool in which the context and configuration dependency can be specified. Namely, that context pattern or configuration pattern can be given that is to be matched by the monitor (in the guard function) with the observed context or the current configuration. This solution fits the model-based design approach and provides the complete synthesis of the source code of the monitor component.
- *Systematic design of monitors with different strategies*: The model-based solution offers systematic monitor design starting from property specifications and providing monitor synthesis, integrating the event-based property specification approach with the guard condition based context- and configuration dependency specification approach. The supported strategies include the use of complete behaviour specification (using statecharts), the light-weight scenario specification (using sequence diagrams) or the declarative property specification (using temporal patterns). Selection among

these strategies can be based on the information available on the required behaviour and the level of detail to be monitored.

- *Automatic tools*: The model-based design approach is supported by automatic tools for specifying and managing the properties (by widely used off-the-shelf modelling tools like Yakindu and Papyrus) and synthesis tools (developed in the project). The synthesis tools are integrated into the property specification tools. For designers who prefer textual languages over graphical languages, Xtext based intermediate statechart language is provided to start the monitor synthesis directly from such property specification.
- *Independent monitoring*: The monitor is designed and its interfaces are constructed to allow independence from the observed components. On the one hand, the specification of the property to be monitored in made independently of the design of the monitored component (i.e., not the same design is used for the implementation of the monitored component and its monitor). On the other hand, the monitor does not share variables or other state information with the monitored component, but observes events through ROS topics.

3.2.2 Weaknesses

- *Detailed semi-formal specification is needed*: If a designer is not familiar with the graphical or textual languages offered by the monitoring infrastructure then she/he has to learn these languages (and also the related modelling tools), Note, however, that these languages, as being included in the Unified Modelling Language, are nowadays part of the electrical engineering and software engineering studies.
- *Limits of scenarios as properties*: The scenario based property specification using sequence diagrams does not provide a complete behaviour specification, as its goal is only the specification of allowed or forbidden event sequence in case of a given condition. Accordingly, complete behaviour can be specified by several scenarios that are difficult to manage, Note, however, that for the purpose of complete behaviour specification the statechart language (also supported by the monitoring infrastructure) is offered.
- *Overhead of monitoring*: On-line verification by monitoring involves an unavoidable overhead as the monitor components need additional memory for code and data (resulting in memory overhead) and also CPU time for execution (resulting in runtime overhead). The overhead is optimized by the careful design of the monitoring algorithms (that evaluate the sequence of events observed by the monitor) and the implementation of the related data structure in the monitor. The monitor can be integrated into a system as an observer component that only accesses information by subscribing and listening to ROS topics, in this way it does not need instrumentation of the monitored component (reducing the direct overhead caused by the instrumentation). However, to access internal information and generate the related events for the monitor, instrumentation may be necessary.
- *Separate tool for specifying context and configuration dependency*: The languages that are used to specify the monitored properties in terms of event sequences, and the languages that are used to define the guard conditions in terms of context or configuration patterns, inherently differ. Namely, statechart diagrams, sequence diagrams and temporal patterns are used to specify event sequences, while class diagrams and related query languages are used to specify the guards. Accordingly, they need separate tools that are not directly integrated with each other (just the guard functions are referred by their names in the statechart and sequence diagrams), The integration is performed on a source code level as the source code generated for evaluating event sequences is linked with the source code generated by a separate tool for the guard functions.

3.2.3 Opportunities

- *Complexity of classic V&V*: The verification and validation of context dependent and adaptive reconfigurable systems is a complex problem as it is difficult to predict the potential contexts and configurations in design time. Typically, classic V&V aims at checking the typical or critical scenarios estimated in design time, without guaranteeing the correctness or safety in case of other scenarios. The on-line verification of behaviour by monitoring offers a solution for the detection of erroneous events related to (1) the violation of assumptions about the system context, (2) violation of expected system properties due to operational faults, and (3) the deviation from expected effects of runtime actions like reconfiguration. A detected erroneous event may trigger an intervention into the system to perform corrections (to ensure safe behaviour) and recovery.
- *V&V challenges in checking context-aware behaviour*: As a specific aspect of the problems mentioned in the previous point, the complexity and diversity of the context of a system with context-dependent behaviour is a challenge for verification. Context modelling with hierarchic structure of the types of context objects (e.g., the concept of “furniture” covers the concepts of “table”, “chair” etc.) and abstract relations (e.g., “close to” and “far enough” in case of distance between obstacles and the robotic system) allows a compact specification of context patterns that influence the correct/safe behaviour of the system. In case of on-line verification, these context patterns are matched with the actually observed context by the monitor (in guard functions), providing this way a mechanism to detect diverse situations that may need corrective actions triggered by the monitor.

3.2.4 Threats

- *Model-based design is not widespread*: Although model-based design is considered as a way to address complexity and to provide understandability and unambiguity in the design of modern computer based systems, it is not used in all companies. As stable design tools are being available and the related training is being offered, this threat is expected to disappear. To address this issue, a textual property specification language is offered for designers who are not familiar with graphical (diagram based) modelling languages.
- *Testing and debugging of monitor components are difficult*: Monitor components, as the other components of a critical system, have to be tested and debugged/corrected in case of *their* design or implementation faults. The difficulty of these activities is due to the fact that monitors react to run-time faults, violation of assumptions about the context, and erroneous changes in configurations of the monitored system. Accordingly, the testing and debugging of the monitor components needs a specific environment (e.g., the use of fault injection tools, context emulation tools, reconfiguration support tools) in which the mentioned effects can be induced in the monitored system and the related reaction of the monitor can be checked. Relaxing the need for testing the monitor components embedded in their software context, specific simulation tools can be used that generate only the events to be checked by the monitor.

3.3 Assessment of Capabilities and Efficiency

In this section first practical experience of integrating the monitor into a demonstrator (WP42 PIAP Scout robot) is recalled then the overhead and performance of monitoring is measured.

3.3.1 Monitor Integration and Evaluation

Here we report the integration of the monitor generated by the source code generator tools into the Scout robot developed by the Industrial Research Institute for Automation and Measurements (PIAP). The goal of the monitor was verifying in runtime whether transitions between different states of the robot are allowed. If there is an error, the monitor informs the operator so she/he can take a proper action.

The specified properties

The monitor follows the state of the robot by receiving events about that change this state. Before a state transition, the monitor evaluates the event and decides whether it is allowed or not. If it is allowed then the state is changed. If not, a callback function is called and the operator is informed (see the *errorAction()* function in Section 9 Appendix A). The allowed event sequences are specified as state transitions in the form of a statechart diagram (Figure 11).

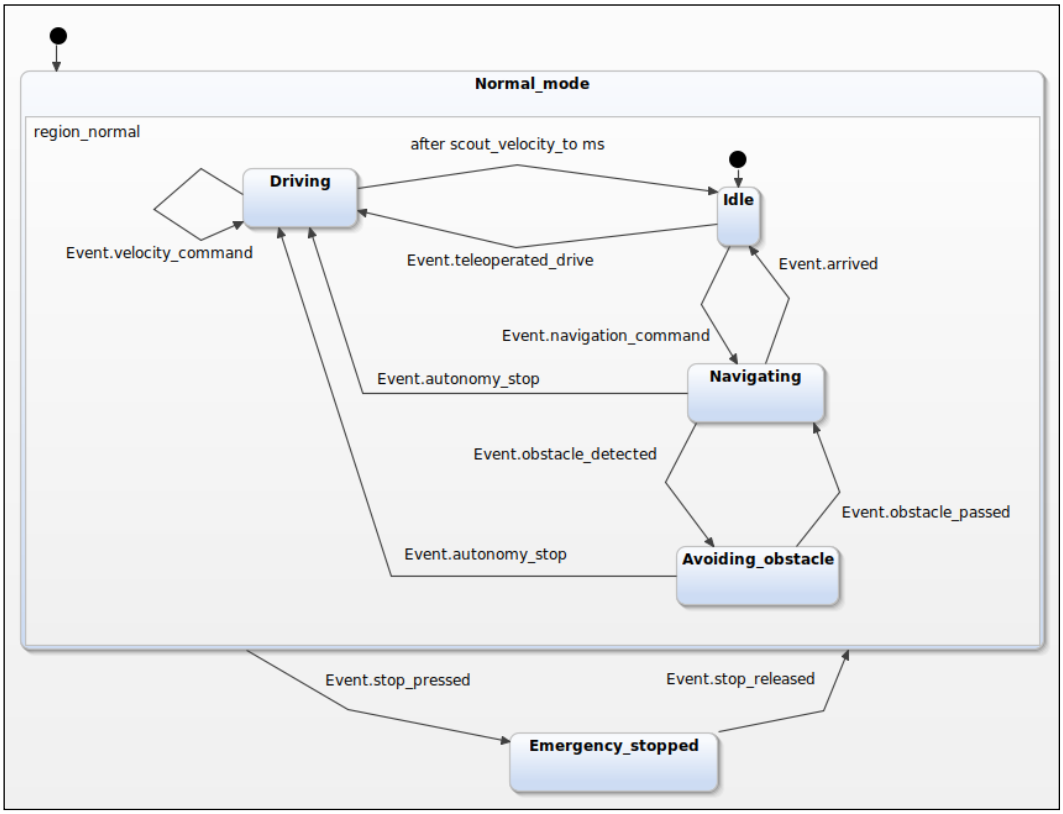

Figure 11. The reference statechart belonging to Scout

The events received by the monitor from the robot are summarized in Figure 12 (the names of the events that identify the change of states are self-explanatory). Note that in case of state *Driving*, a timeout *velocity_to* is used to transition to the *Idle* state.

Figure 12. The events observed and checked by the monitor belonging to Scout

The models were constructed using the Yakindu Statechart tool and source code of the monitor was generated in two iterations (correcting some specification and code generation mistakes during integration).

Overview of the integration

The monitor is running in a separate process on an operator's console and it uses ROS for communication with other components. The ROS topic */bmemonitor/status* topic is used for sending events from the console and from the robot, and the topic */bmemonitor/error* is used for sending information from the monitor to the operator. The integration is presented in Figure 13.

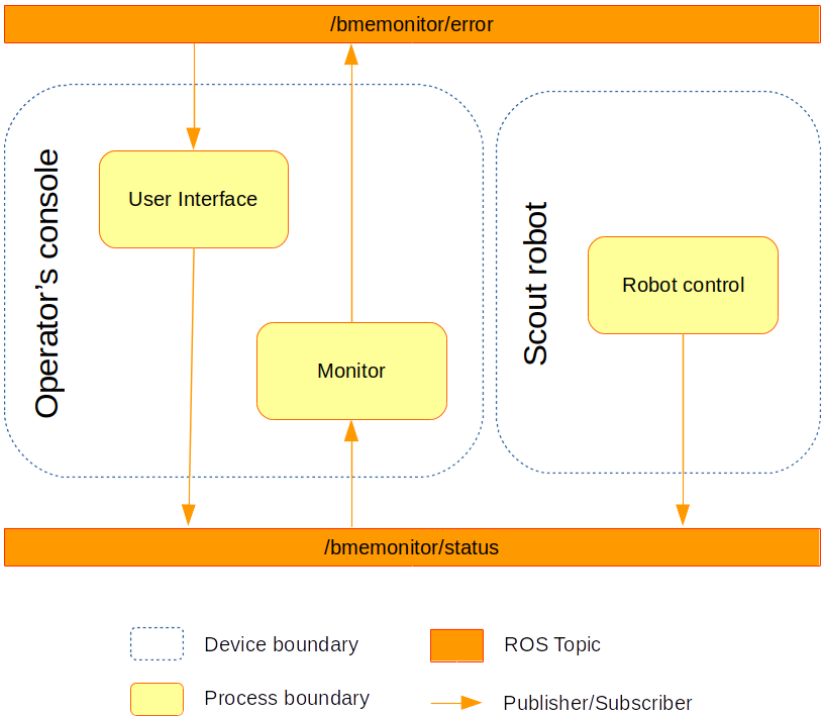

Figure 13. Integration of the monitor into Scout

Testing scenarios

The monitor with Scout was tested in two different scenarios in order to verify that the monitor can provide useful information when something is wrong.

Scenario 1: Emergency stop

In this scenario the robot was driving autonomously when the operator pressed the emergency stop button in case of a safety hazard. Despite the emergency situation, the autonomy module continued sending navigation commands requesting the robot to change its position.

The screenshot in Figure 14 shows that the monitor detected the error (incorrect behaviour with respect to the statechart reference model presented above) and the proper message was displayed for the operator.

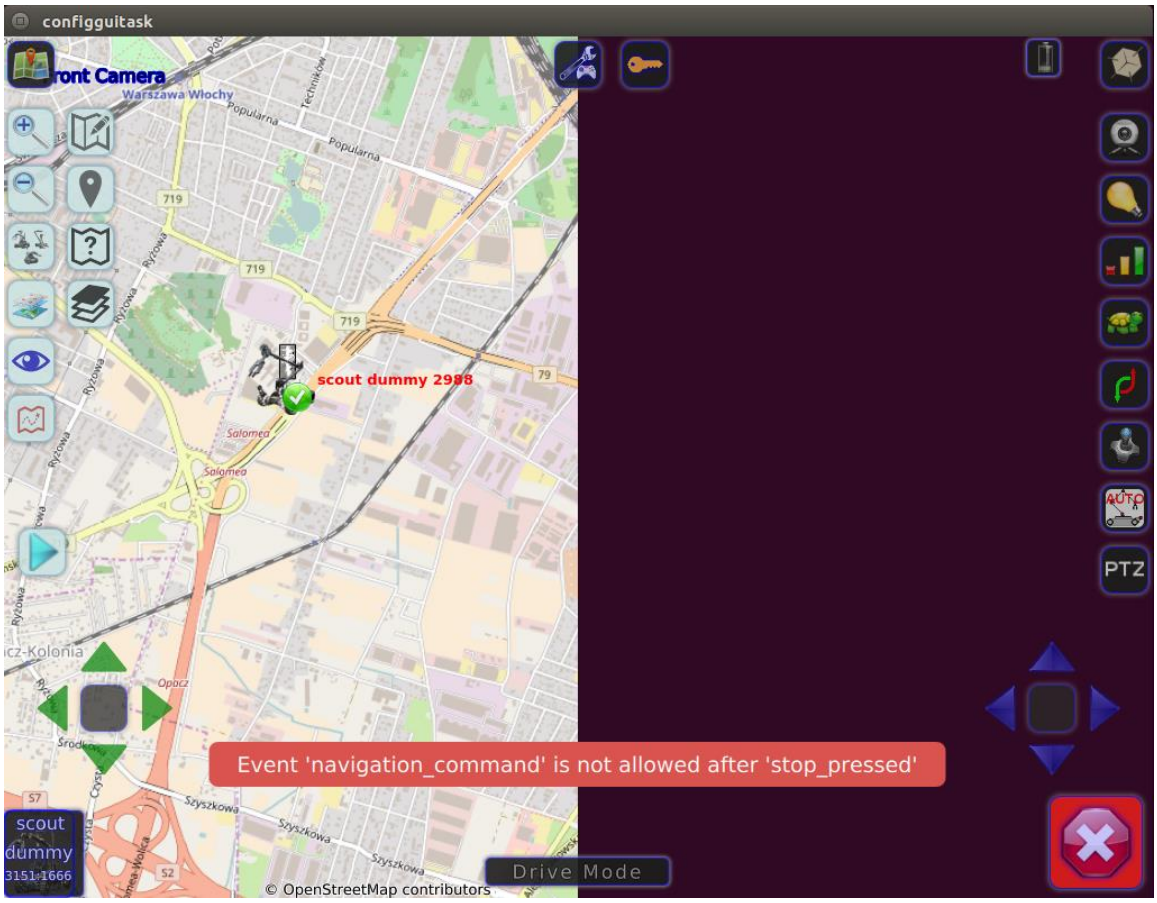

Figure 14. Error detected by the monitor in Scenario 1 (Emergency stop)

Scenario 2: Autonomy drive

In this scenario the Scout robot was driving autonomously but the operator decided to control the robot manually. In this case, the autonomy module should give up the control and switch to the idle state. However, in this scenario the autonomy module continued sending navigation commands. As previously, the monitor discovered the incorrect behaviour (with respect to the statechart model) and the callback function informed the operator (Figure 15).

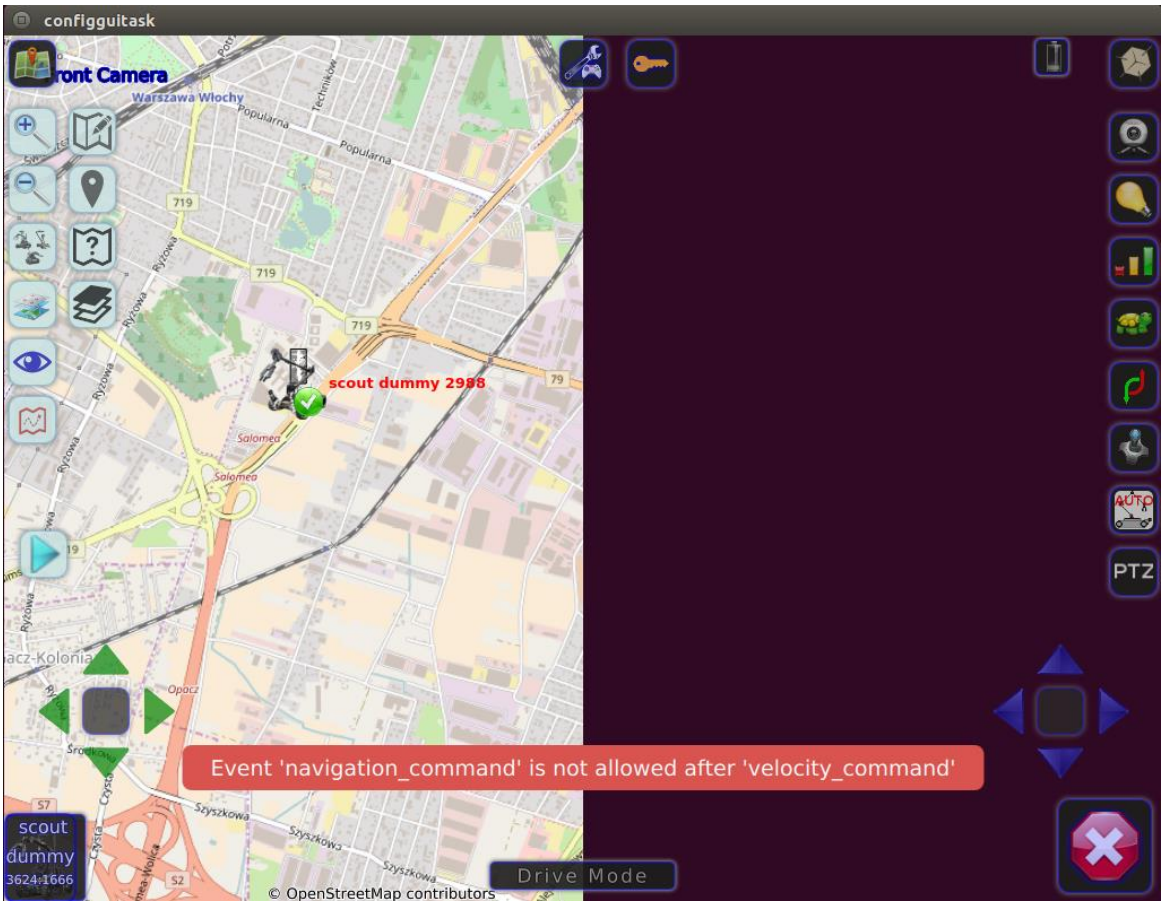

Figure 15. Error detected by the monitor in Scenario 2 (Autonomy drive)

Evaluation

PIAP concluded the integration and testing with the following evaluation:

- The monitor was able to correctly detect invalid transitions between states so for applications with well-defined states and transitions it may prove very useful. It is especially valid if the design of the monitored application uses a state pattern for its core functionality.
- In the case of Scout, the integration was cumbersome as the design did not use the state pattern and the events had to be formed artificially by adding new functionality to the application that was responsible for checking conditions and emitting a proper event. This extra code (instrumentation) introduced additional complexity.
- The integration was particularly difficult in case of the autonomy module, since its internal state was not instrumented, this way it was not known whether the autonomy module is active or not. If a request was sent by the operator to the autonomy module to stop, the operator did not know when it will emit the *autonomy_stop* event so that she/he can safely emit *velocity_command* to control the robot. The solution of this problem would need additional code.
- The overhead of monitoring depends on the way the monitor is integrated with the application. In this case there was a separate monitor process using ROS for communication (see Figure 13), and the related ROS libraries were responsible for the huge part of the memory footprint.
- In summary, the monitor may work very well with state based designs, but in case of Scout (not following this pattern by design) the added value of a state-based monitor did not fully justify the effort of difficult integration.

3.3.2 Performance and Overhead of Monitoring

In case of monitoring, one of the important questions is the overhead (in memory and runtime) and the performance of checking. In this section measurements results are provided that allow the estimation of these characteristics in case of the different monitoring approaches.

Performance of the statechart and scenario based monitors

The monitors generated on the basis of statechart diagram and sequence diagram based property specifications share the same intermediate formalism (the textual statechart model) for code generation, this way these have similar performance. In the following we refer to these monitors as statechart based monitors.

The internal execution engine of the monitor is triggered by the incoming events. To check whether the incoming event triggers a transition (i.e., the event is a valid successor of the previous event), the engine has to examine the outgoing transitions of the active state configuration. This means that the execution time is mostly determined by the number of outgoing transitions from the state configurations.

To measure the effects of different number of outgoing transitions, we have generated statechart models (as benchmarks) with the following characteristics:

- "*Complete graph*": the number of states is *n*, each with *n-1* outgoing transitions.
- "*Cyclic graph*": the number of states is *n*, each with *1* outgoing transition.
- "*Random graph*": the number of states is *n*, half of the states have *1*, and the other half of the states has more than *n/4* transitions.

The statechart based monitors were generated from these test models and were compiled using *g++* with the *-O3* flag set.

In the measurement setup, the monitor was driven by a test program where the calls to the evaluation function of the monitor were made directly from a loop. The measurements were done on an Intel Core i5-6500 CPU, where the monitor was executed on a dedicated core running at 3.2 GHz. Execution times were measured using the standard *C++11 chrono* library's high resolution clock functions. To minimize the possible effects caused by the operating system's interrupts and scheduling, the loop of the evaluation calls was executed one hundred thousand times (then the average execution time of the calls were calculated). Also the measuring function ran multiple times, and the outliers in the results were eliminated by averaging results in the 80 percentile.

The execution times for the test models can be found in Figure 16.

Results	n = 10	n = 100	n = 1000	n = 10 000
Cyclic	69 ns	66 ns	61 ns	62 ns
Complete	92 ns	371 ns	-	-
Random	76 ns	211 ns	-	-

Figure 16. Execution times (ns) of checking an event by statechart based monitors

The complete models for *n>=1000* would contain almost a million transitions. As the mapping tool from statecharts to the intermediate representation for monitor synthesis is not prepared to traverse such huge models, monitor generation is not currently supported for these model sizes. The same holds for the random models. Note that this model size is not realistic to be developed by hand.

The measurements were also performed on the monitor generated for the WP44 demonstrator (i.e., based on the statechart model given in Figure 8). In this case the average execution time of evaluating an incoming event by the monitor was *108 ns*.

We have also measured the execution time of guard functions that perform context matching. We used a context model that contained all together 50 modelling elements. The context patterns to be matched were similar to those of Figure 7. We have measured the average time needed for context matching: the maximum execution time was *150 ms* for the most complex context pattern (containing an expensive transitive closure pattern) and the minimum runtime was *10 ms* for the simplest context pattern.

Memory usage of the statechart and scenario based monitors

The memory usage of the monitor is affected by the number of states and transitions in the statechart model that specifies the property to be monitored.

Memory consumption of the monitors was measured using the same models as were used for the execution time measurements. The monitor executables were compiled using *g++* with the *-O3* flag specified. The *-O3* flag might not provide the best optimizations for memory usage, but developers typically opt for the speed increase versus the minor free memory gained.

The size of the used memory was measured with the Linux tool *ps*. The platform overhead of *g++* and Linux were taken into account by calculating the difference of the memory usage of the monitors and a simple one-line program realizing an infinite loop (which is typically over *1 MB*).

The memory usage of the monitors with different settings is presented in Figure 17.

Results	n = 10	n = 100	n = 1000	n = 10 000
Cyclic	4 kB	28 kB	40 kB	2356 kB
Complete	24 kB	1744 kB	-	-
Random	12 kB	921 kB	-	-

Figure 17. Memory usage of statechart based monitors

Note that the names of the events and states are stored as strings by the monitor for error reporting and logging purposes, thus the memory usage is influenced by the lengths of the names. For environments with ultra-low resources, the monitors may be modified by storing only IDs and references (that are used by the evaluation function), which would reduce memory consumption.

The measurements were also performed on the monitor generated for the WP44 demonstrator (Figure 8). In this case the memory usage of the monitor core was *4 kB*.

Performance of temporal pattern based monitoring

We have compared the execution time of our monitor implementation with the classic approaches of term-rewriting using the Maude engine (H&R, [25]) and the source code generation based on alternating automata (F&S, [26]). We have used the formulae and traces suggested in these papers. The evaluation trace was created by iterative repetition of the *(a; b; a; b; a; c; a; a; b; g; f; h; c; b; a)* event sequence. The measurements were carried out on a low-end platform (Intel processor core running at 2.2 GHz).

Figure 18 presents the execution times for evaluating two formulae on various trace lengths (note that the evaluation of the formulae requires the analysis of the entire trace), in comparison with the term rewriting approach [25].

	Formula 1 $G(b \rightarrow F\ c)$		Formula 2 $F(\neg G(b \rightarrow F\ c))$	
Trace length	**H&R term rewriting**	**Our approach**	**H&R term rewriting**	**Our approach**
1500	20	0.53	110	0.62
3000	40	1,10	220	1.25
4500	60	1.64	320	1.87
6000	80	2.20	420	2.64
7500	100	2.70	530	3.29
9000	120	3.70	640	3.84
10500	140	4.00	760	4.47
12000	160	4.50	860	5.50
13500	180	5.10	970	6.00
15000	200	5.60	1100	6.60

Figure 18. Monitor execution time (ms) in comparison with the term rewriting approach

Figure 19 presents execution times for evaluating three formulae in comparison with the alternating automata approach [26].

Formula	Trace length	F&S alternating automaton (BFS)	Our approach
F1	**1000**	78	0.57
	2000	54	1.13
	3000	76	1.84
	4000	99	2.46
	5000	123	3.22
F2	**1000**	82	0.66
	2000	52	1.39
	3000	73	2.07
	4000	94	3.40
	5000	117	3.66
F3	**1000**	876	0.95
	2000	1660	1.93
	3000	2377	2.96
	4000	3244	4.08
	5000	4034	6.73

Figure 19. Monitor execution time (ms) in comparison with the alternating automata approach

The explanation of the speedup is the effective implementation of sub-expression evaluation and the inherently programming-oriented nature of our approach: the entire solution is targeted for code generation, the data structures and algorithms seamlessly fit to C/C++ pro-

gramming languages resulting in a straightforward code generation step and a high performance application.

The code generation time was also measured using temporal logic formula of increasing length (i.e., increasing number of temporal operators by the conjunction of *(r => p U d)* expressions in context of a *G* operator). The longest formula contained 8+1 temporal operators, in this case the code generation was performed in still less than 1 second. Note that in case of typical safety patterns (see in D34.10) the number of temporal operators is less than 4.

Memory usage of temporal pattern based monitoring

The memory consumed by the monitor for evaluating a set of properties on a trace of *n* observed events involves the program code implementing the base algorithms (creating and managing evaluation blocks) and the memory used for storing evaluation block instances (see in D34.31). Since the base algorithms can be implemented in a few lines of code, the memory usage is dominated by the evaluation blocks. This is investigated below in case of C++ implementation.

An evaluation block instance maintains the values stored on its interfaces and the pointer to the previous instance. The "registers" on the interfaces of evaluation block instances can be implemented by bit vectors: storing the values of atomic expressions on the left interfaces require a single bit per expression while ternary values on the top and bottom interfaces need at least two bits per expression (storing bit patterns is supported by C bit-fields, the C++ bitset data type template, etc.). The pointer to the previous instance is obviously implemented as a programming language level pointer or reference construct. In case of the C++ programming language chosen for our prototype implementation, no explicit metadata or type information is stored, only in case of classes with virtual functions a single pointer to the virtual function pointer table.

Let us consider a 32 bit architecture where a pointer is 4 bytes long. In worst case (when different events are observed in each step), the number of bytes required for the evaluation of the expression "*Globally r implies p Until d*" with events *r*, *p* and *d*, on a trace of n steps is as follows:

$$1*(4 + 1 + 1 + 1 + 4) + (n-1)*(4 + 1 + 1 + 1 + 4) \text{ bytes}$$

For example, in case of 1000 steps it results in 11.000 bytes. In asymptotic aspects, the memory consumption is a linear function with the number of trace steps when different events are observed in each step of the trace (worsts case); otherwise repeated (successive) events can be checked by the same evaluation node this way the memory need is reduced.

Comparison of different monitoring approaches

We also performed direct comparison of three monitors:

- Monitor for checking the local control flow of the application. This is a reference case with an expected high overhead as the monitor checks each node of the program control flow graph (CFG) by instrumenting each branch-free statement block of the program (sending signatures identifying the node to the monitor that checks on the basis of the reference CFG whether the signature is allowed successor of the previous one).
- Monitor for checking statechart based property specification.
- Monitor for observing and checking the system behaviour defined as scenario diagram.

We measured the code overhead and execution time overhead of monitoring a control module implemented on a simple *mbed* NXP LPC1768 microcontroller platform[2] with ARM Cortex-M3 running at 96 MHz (the goal was the comparison of overhead and not explicit time

[2] https://developer.mbed.org/handbook/mbed-Microcontrollers

measurements). The local CFG monitor was deployed as a local process executed on the same microcontroller. The statechart and scenario based monitors were deployed on a separate microcontroller using the Ethernet based communication capabilities of the microcontrollers.

To get the highest overhead possible, as extreme reference case we modified the control module by removing the statements belonging to the interactions with other modules of the application, this way practically have only the program control flow skeleton of the module.

Memory overhead

The program code overhead is presented in Figure 20. The overhead turned to be acceptable: instrumentation for local control flow checking introduced 1.3% overhead, statechart and scenario based checking needed about 0.5% and 0.7% more code, this way the instrumentation for all checking possibilities resulted in less than 3% code overhead.

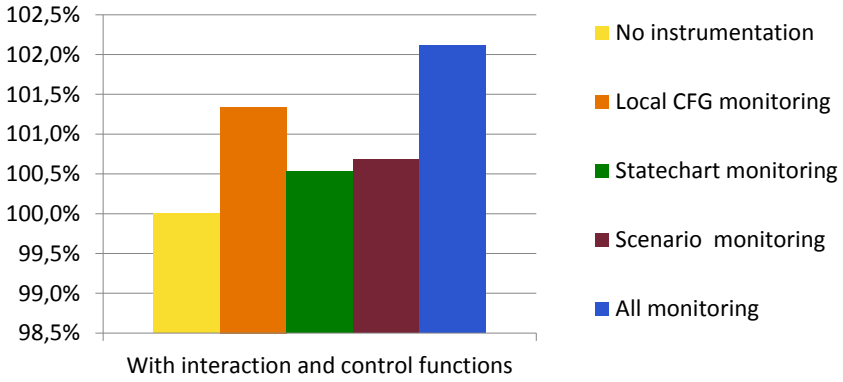

Figure 20. Program code overhead in case of various monitoring techniques

Execution time

Figure 21 presents the execution time overhead. In case of the control module that implements the control logic and performs interactions with other modules of the application, the run-time overhead for all checking was less than 12%. In the extreme reference case having the code skeleton only, the time required to provide information to the monitor modules dominated the execution time (especially in case of the local CFG monitoring) and the overhead reached far more than 100%. Note that in order to have the same scale on the vertical axis of diagrams in Figure 21, different number of state changes were measured in case of the full code and the skeleton code.

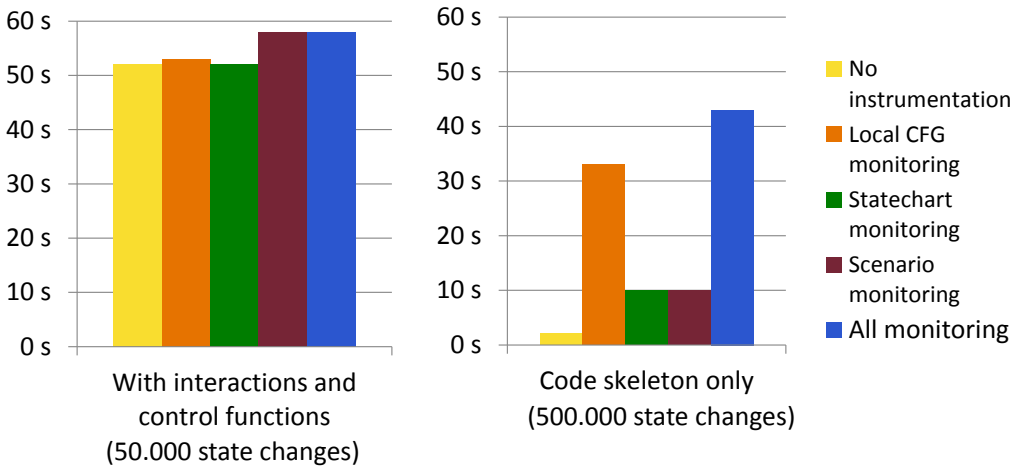

Figure 21. Execution time overhead in case of various monitoring techniques

4 Assessment of On-line V&V using SIL methodology

The topic of this chapter is assessment of the on-line V&V used in a Software-In-the-Loop (SIL) approach to check the collaboration between autonomous Rotorcraft Unmanned Aerial Vehicles (RUAV) and Wireless Sensor Networks (WSN).

Like described in the previous chapter, the goal for on-line V&V used in a Software-In-the-Loop (SIL) approach is the on-line verification of the behaviour of R5-COP systems (RUAV) by elaborating methods and tools for runtime monitoring. Nevertheless, in this case the technical approach is different from the automated construction of monitor components by the synthesis of their source code from high level property description. The SIL approach provides the simulation model of the RUAV, from which the source code for the real application can be automatically generated. As the simulation environment MATLAB/Simulink provides the simulation language Extended Finite State Machine (EFSM) and the possibility to transfer the code to real environment, then also the on-line V&V functionality is included in the simulated model and simultaneously in the software transferred to RUAV. This approach allows on-line V&V tools to be used in the full life cycle of the developed application, from SIL model to deployment in real time environment.

4.1 Summary of the Method and its Novelties

The task T43.2 (and T34.4) was devoted to building a SIL model for a demonstration task: Model of an autonomous robot cooperating with WSN. Such a model is built early in the application development process, when no real hardware or code is available. The idea was to implement on-line verification ideas and integration tools early, using the SIL model and to assess what advantages of such approach can be observed.

4.1.1 The SIL Methodology

Software-in-the-loop (SIL) is a system development methodology for embedded control systems, where special simulation model is built for hardware and environment simulation around the embedded code, which is, in this case, the system-under-test (SUT). Such SIL model then allows the real time embedded program (SUT) to be tested not only in the real environment but also with this model in simulated environment without using dedicated hardware.

In-the-loop means that the control system is in a closed-loop (inputs are driven by outputs, see in Figure 22). All types of the Field robots are good examples of such systems.

Such model based methodology can be used for a wide range of real time systems, but is also extremely popular for robotics [27] including field robots and RUAV [28]. In case of RUAV, the RUAV hardware is simulated and the environment, where the RUAV flies, is also simulated. For environment simulation many possible virtual reality simulation tools may be used. Mandatory is the camera simulation from this virtual reality, as it is one of the main tools for RUAV during a mission.

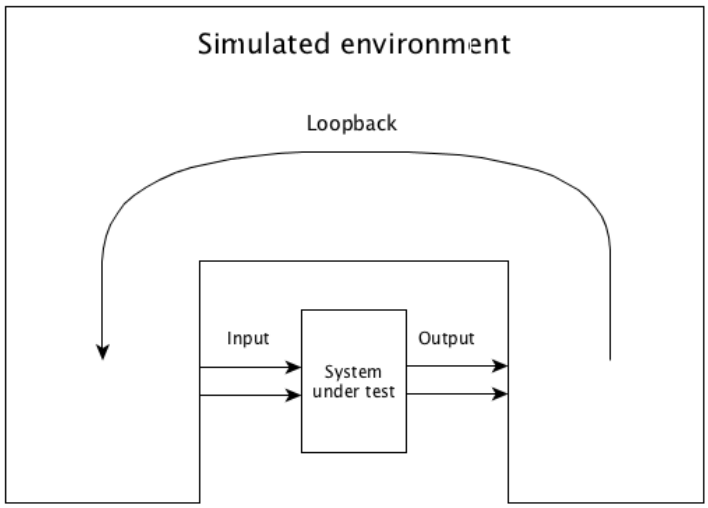

Figure 22. Closed loop control system

Usually such model based methodology includes different stages of producing and testing code: Model-in-the-loop (MIL), SIL, Hardware-in-the-loop (HIL), Processor-in-the-loop (PIL) [6]. To implement full loop (especially HIL platform) is an expensive and time consuming process, which is more suitable for large companies in aerospace and automotive industry using expensive chain of hardware and software tools (NI, dSpace [29]).

4.1.2 Implementation of on-line V&V in the SIL model

The solution offered by WP34.4 implies the construction of an external process verification mechanism: verification by observing processes from aside without intervention into the execution of these base processes. Events confirming process step executions are collected and verified. All events are detected by event agents and sent to monitors for verification. Agents are instrumented model components where the events occur. Monitors are separate processes in the SIL model. Agents and monitors are developed and implemented for different components and are part of SIL model.

The MATLAB/Simulink executes the SIL model in so called ticks. An execution of a tick can result with a transition to the next state (one step) or staying in the current state. During the execution of a tick, processes send information to other processes/components of the model including the hardware simulators and on-line V&V monitors. The simulators perform activities similar to that would be performed by real hardware and return new values of common variables. The loop can be run for long time to simulate continuous actions of RUAV, GCS (Ground Control Station) and WSN.

In Figure 23 a Monitor process is shown that checks several events:

- ROS message regularity,
- Hardware parameters of the motor (energy, status, flying time, etc.),
- Software (state consistency with motor parameters).

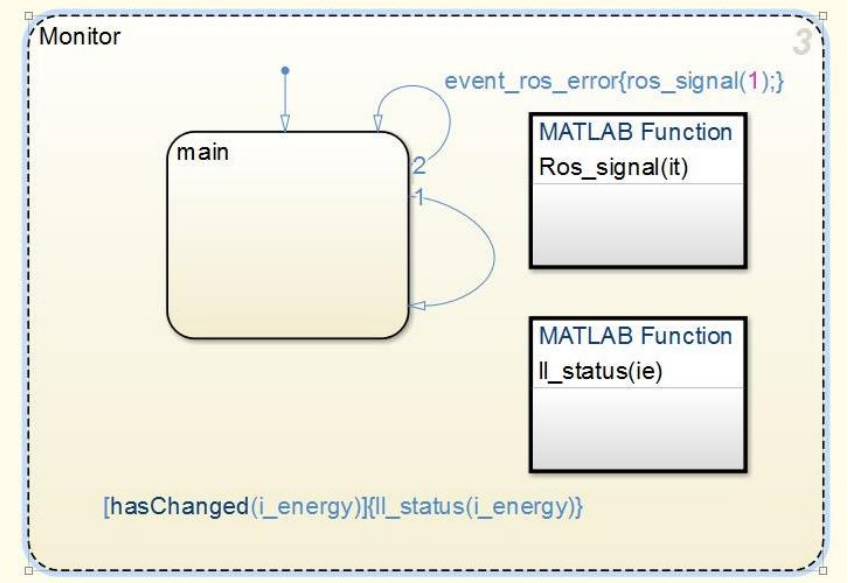

Figure 23. Example Monitor process

The RUAV embedded code includes error processing. It is available for the on-line checker (the Monitor) and it consists of:

- Error warning event propagation,
- Safe mode of operation (see in Figure 24), robust and safe execution in critical situations, including immediate landing or going home.

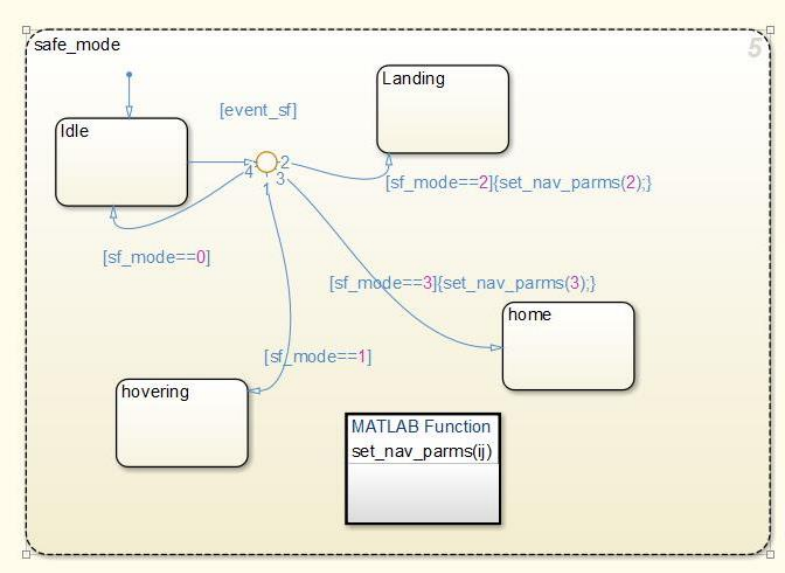

Figure 24. Safe mode operation process

4.1.3 Novelties of the Approach

The SIL technology was developed by large automotive and aero companies, at first for inner use of the companies, later also as commercial software. Those tools are expensive and mostly available for industrial companies. Use of MATLAB/Simulink for those purposes is popular in academic context.

If the SIL model is produced only for one product it can be too expensive, because the model is complicated and time consuming to develop. But if one has the technology ready, where the model is adopted for several products, then its use is very cost effective. For autonomous

devices requiring specially built test environments, SIL approach provides possibility to transfer large part of the field tests to laboratory environment, thus making large resource savings.

The implementation platform of our SIL approach is based on a State Machine, using for modelling the Stateflow language. That allows parallel processes, this way the embedded code does not extensively use processor (time) resources.

As the monitor is seamlessly integrated into the State Machine, it is allowed to use (in case of detecting an error) the built-in general error handling methods. For example, in case of serious errors, the State Machine can be put into Safe mode. Also for communication with GCS existing communication channel in ROS can be used. That allows also double checking of on-line verification on GCS, or saving events and using post mortem verification.

The main novelty of the SIL approach is that on-line verification tools are started to develop at the same time as the code is developed; on-line testing and monitoring is also performed from the start. Being in the Stateflow model, they are in the code during the testing and remain also after deploying. Tools are closely interconnected with existing code. On-line testing then covers hardware checking, environment checking, and software on-line verification.

4.2 SWOT Analysis

Figure 25 presents the SWOT analysis of the on-line V&V using SIL methodology introduced in deliverable D34.41 and implemented in the MATLAB/Simulink/Stateflow environment.

	Helpful	Harmful
Internal	• Universal methodology • MATLAB/Simulink as universal simulation platform • Large number of ready-made tools • Modelling by Finite State Machine • Parallel execution of processes • Automatic model transition to hardware **Strengths**	• High level model • Resources for monitors cause overhead • Restricted applicability for real time systems **Weaknesses**
External	**Opportunities** • On-line V&V model in one PC • On-line V&V model transition to hardware	**Threats** • Real time restrictions • Environment complexity restricts applicability of the methodology

Figure 25. SWOT analysis of the SIL methodology

Strengths

- The SIL methodology is universal and provides MATLAB/Simulink based universal solution for V&V for collaboration of several autonomous systems.
- The methodology is based on MATLAB/Simulink which is an universal simulation platform containing large number of ready-made tools.
- The simulation language is the Extended Finite State Machine (EFSM), which is used in this SIL methodology. It provides the possibility to develop high level abstract models and include there C code and modules developed by 3rd parties.
- The MATLAB/Simulink platform provides parallel processes, which allows executing on-line V&V in one process with basic processes.
- Agents are incorporated in code with automatic transfer of events to the monitoring process.
- The MATLAB/Simulink platform contains tools for transferring model to real time code.

Weaknesses

- The methodology is applicable to on-line V&V of autonomous systems on a high level.
- Lot of separate V&V monitors can be implemented, that results in overhead.
- The borders and limitations of the usage of SIL methodology are not well defined.

Opportunities

- The broad functionality of MATLAB/Simulink/Stateflow gives users the opportunity to make an on-line V&V model with relatively small resources, containing essential part of autonomous mobile system functionality. It can be simulated, executed, validated and verified on one PC without using real HW.
- MATLAB/Simulink contains several tools for transferring model based on-line V&V to real code, which subsequently can be used with real hardware.

Threats

- The simulation of autonomous systems and their collaboration in MATLAB/Simulink does not replace hard real time testing. For that one must do also hardware-in-the-loop testing and tests with real hardware.
- Environment complexity (weather conditions, visibility, etc.) and quality of its simulation can limit the truthfulness of the results and usage of this methodology.

4.3 Assessment of Capabilities and Efficiency

4.3.1 Usability

The SIL methodology provides many possibilities for developing autonomous systems. As first step, developing simulation model is provided by the Stateflow simulation language with high level of abstraction and with option to include C-code in the model. That allows using wide range of ready-made programs, decreasing the amount of necessary new code. The developed model for autonomous objects can be simulated in MATLAB/Simulink environment verifying the correctness of the model.

Other essential feature of the SIL methodology is a possibility of close integration of MATLAB with ROS, which is strongly enhanced in the last years. ROS provides standard interfaces for various devices and large number of open source components for processing the information from those devices. As the MATLAB/Simulink environment provides tools for using the ROS features, the usability of this SIL methodology is increased.

4.3.2 Overhead

On-line V&V using SIL methodology uses the Stateflow features to execute parallel processes. The on-line monitor is built as a parallel process, which executes in parallel with the basic functionality of the RUAV. In case of RUAV collaboration with WSN the overhead is small, but in more complex and more time critical applications the overhead can cause problems. Information about the environment processes, required for on-line monitoring, is included as part of standard interface signals, and respectively it is not a source of large overhead.

4.3.3 Efficiency

The SIL methodology provides several interlinked steps for application development. As a first step, the simulation model is built for MATLAB/Simulink environment. Interfaces between devices are provided by ROS, which is supported in MATLAB/Simulink environment. After the simulation for verification of the model, it is transferred to real environment. That is done automatically by MATLAB tools, keeping the functionality of the model. It is possible that the automatically generated code is not so efficient as a code written by experienced developer.

But such technics always allows returning to the SIL model, make changes there, re-test the model there and start a new code generation. That strongly increases the efficiency of making changes in the application and testing them. Slight decrease of the efficiency of the generated code for most applications is not serious (as in case of RUAV and WSN collaboration in our demonstrator).

We can conclude that in case of development costs, the solution of using SIL model with built in on-line V&V tools, is very effective as it uses Simulink with its well developed and approved platform, for providing simulation model on high level of abstraction, with automatic generation of real code.

5 Assessment of Incremental Testing

In this chapter the method is assessed that can be used for the selection, adaptation and extension of test cases in an incremental testing workflow.

5.1 Summary of the Method and its Novelties

The existing tools and approaches presented in the literature usually concentrate on one programming or modelling language as the input source for incremental testing. However, in R5-COP there could be multiple levels and types of reconfigurations and changes in which case incremental testing is needed. Instead of performing incremental testing separately for each of the change types, we could apply a unified approach, as basically they all can be handled in a similar way.

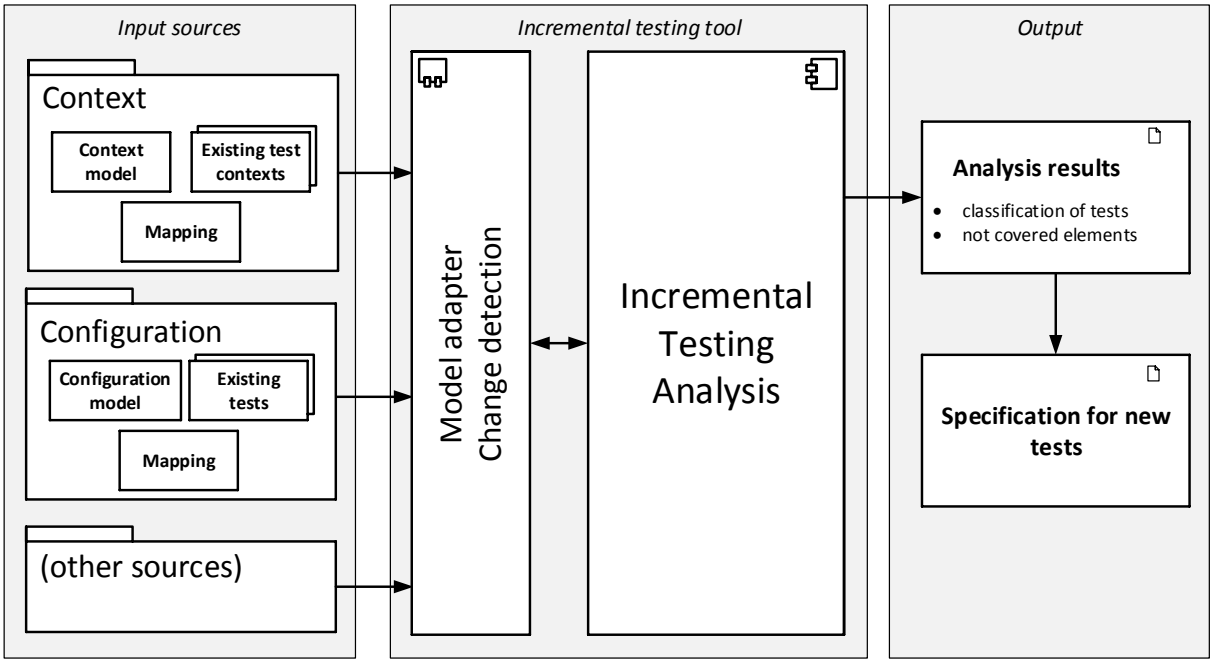

Figure 26. The incremental testing methods

Accordingly, we developed a common, general incremental testing approach, and connected the specific test types (test contexts from context models, module/integration tests for components, etc.) using special adapters to this core. Figure 26 depicts the approach.

- The *incremental testing analysis* component is the central element of the approach. It defines a general model for representing the tests and tested elements. The regression testing algorithms (test selection or coverage identification described in deliverable D34.20) work on this general model.
- A *model adapter* is responsible for connecting the different sources, like context or configuration models and tests to the general analysis component. This adapter is developed for each source type and is responsible for converting the models and tests to the internal representation of the analysis component. This component is also responsible for detecting changes in the sources.
- The *outcome* of the analysis is a classification of tests and the coverage information of the source elements (e.g., to detect that there is a class in the context model that is not present in any of the existing test contexts). This information can be used later to create new tests either manually or automatically.

In summary, this method is driven by the analysis of the new requirements (formalized in scenarios), the changes in the context of the system (formalized in context metamodels), and the changes in the internal components (formalized in architecture and capability models). The gaps in the coverage of the existing test suites are identified, which drives the adaptation of existing test cases and the generation of new test cases to cover the changes.

The main contributions and novelties of the work are the following:

- A *general concept* of test analysis was introduced and the corresponding languages to capture tests (test cases) and testables (context and configuration elements) and their mapping were defined.
- A *tool* was designed that can perform the incremental testing analysis. Using model adapters the core incremental analysis component is independent from the actual domain, and only the light-weight adapters had to be created when new types of artefacts and related changes has to be handled. A tool implementation was also developed that is based on the Eclipse framework, the de facto modelling environment widely used in industry.
- *Evaluation* of the applicability and scalability of the method and the tool was performed. The evaluation used context and capability model used in WP42 and inspired by the DHS-NIST-ASTM International Standard Test Methods for Response Robots (ASTM International Standards Committee on Homeland Security Applications; Operational Equipment; Robots E54.08.01).

5.2 SWOT Analysis

Figure 27 presents the SWOT-based analysis of the incremental testing method.

	Helpful	Harmful
Internal	• Core methodology is domain-independent • Automated incremental testing analysis tool • Usable at different testing levels **Strengths**	• Attributes of tests and testables are currently limited • Tool is in prototype phase **Weaknesses**
External	**Opportunities** • High cost of retesting everything • Reconfiguration is frequent • Rapidly changing requirements and context	**Threats** • Model-based culture not widespread • Lack of unified representations

Figure 27. SWOT-based analysis of behaviour testing method

5.2.1 Strengths

- *Core methodology is domain-independent*: The core of the incremental testing methodology (the metamodel, the impact analysis algorithm and tooling) is generic and is independent of the actual robot and its domain. Therefore, only a subset (the model adapters) has to be created for each new use case.
- *Automated incremental testing analysis tool*: The incremental testing is implemented in an automated tool that categorizes existing tests cases with respect to a change,

and can help to select a necessary subset of all the tests. The inner working of the method is hidden from the user, only the changed parts of the model have to be marked, and the tool computes automatically the test classification.

- *Usable at different testing levels*: The method and the tool can be used in many different settings; tests can represent “virtual world” descriptions for simulators or manual test setups for real world environments. Depending on the actual robot use case and testing priorities, input models can represent the different configurations of the robot or the context in which the robot operates, broadening the applicability of the method.

5.2.2 Weaknesses

- *Attributes of tests and testables are currently limited*: Currently a basic relation is captured in the models, namely that a given test case “tests” a testable (a module or a context element). However, further attributes could be added to enhance the test descriptors, e.g., cost or duration of the tests. The metamodel was designed to be flexible, thus such changes could be incorporated. The test selection and classification algorithm has to be adjusted, similarly to the algorithms presented in the literature [30].

- *Tool is in prototype phase*: The tool is currently in prototype phase with a basic user interface. Users not familiar with the Eclipse framework and its editors could require more time to create the models. However, domain-specific graphical editors could be easily created to support users not familiar with the Eclipse modelling technologies.

5.2.3 Opportunities

- *High cost of retesting all*: Running all tests (either simulated or real) for every modification is extraordinarily costly, and in several cases it is not even possible (e.g., executing all tests on the standardized NIST test stands requires days). Therefore, identifying and running only a subset of the required tests could offer significant time and resource savings.

- *Reconfiguration is frequent:* With today’s modern robots, reconfiguration could be a frequent activity; hence retesting new configurations is necessary.

- *Rapidly changing requirements and context:* Not only the configuration of the robot, but its requirements and operational context could change rapidly, which makes retesting a non-optional activity that requires support and advanced methods to be cost-effective.

5.2.4 Threats

- *Model-based culture is not widespread:* The incremental testing method is based on creating good context and configuration models for the application domains. This activity has to be supported by modelling experts as domain experts usually do not have the necessary modelling experience. Before the developed test approach could be applied, first the model-based thinking has to be accepted in the company or team responsible for the verification of autonomous systems (e.g., models are not just visual documentation of already written code). Unfortunately, model-based approaches are not yet a common industrial practice. However, in certain industries model-based approaches are gaining a lot of traction (e.g., AUTOSAR in the automotive domain).

- *Lack of unified representations:* The configuration and context models and their adapters have to be created for each new robot as currently there are many different possible representations. However, the skill model developed in R5-COP could offer a common representation that can be applied in several domains.

The goal of this assessment was to collect the advantages and limitations of the methods and tools developed for incremental testing of reconfigurable autonomous systems. The SWOT analysis identified several strengths (e.g., generic approach, automated tooling), but has also found limitations. Some of the limitations were found because the developed tools were only prototypes, and were only applied in the first case studies with the demonstrators.

5.3 Assessment of Capabilities and Efficiency

This section summarizes the final capabilities of the incremental testing methods and the lessons learnt about its usability and efficiency.

5.3.1 Application in Demonstrators

The incremental testing method has been applied to the WP42 mobile robot demonstrator by PIAP. PIAP recently introduced NIST-standardized test stands into its test process. The NIST guideline [31] describes how to make the test rooms comparable. The guidelines use ASTM (American Society for Testing and Materials) standard objects for describing the lay-out of the rooms. These include different types of terrains and obstacles. Various terrains exist especially for mobility exercises, such as ramps, steps, sand, gravel or mud. Obstacles can be used to test different capabilities of the robot like gaps in the floor or signs on the wall.

BME visited PIAP in Warsaw to discuss their testing process and observe the test environments for their robots. After the meeting in the first iteration, we considered the test stand elements (e.g., ramp or wall) as elements of the context model and created a mapping model between robot modules and context elements, e.g. that a ramp tests the capabilities of the motor. This approach was presented in detail in deliverable D34.20. The incremental testing tool was able to select from different test room layouts those ones that are relevant for a change.

However, after revisiting the models with the experts from the PIAP, we concluded that a different modelling approach would be more suitable for their use case. As they have only 3 fixed test lanes (lane 1, 2 and 3 from [32]) then number of different, possible layouts is limited. Variability and numerous testing combinations are introduced by performing different exercises on a fixed lane.

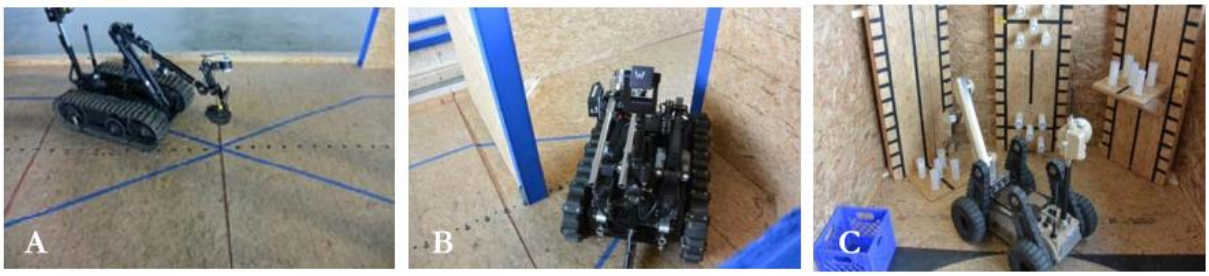

Figure 28. Exercises on NIST test lane [32]

For example, Figure 28 presents parts of test lane 1. Basic manoeuvring and pattern recognition can be tested with the line following exercise in the beginning of the lane. More advanced manoeuvring can be inspected in the middle of the lane by a tight turning. Finally, manipulator dexterity can be tested in the end zone by grasping and rotating objects. The full lane offers much more exercises and testing combinations (e.g. varying lightning, robot movement directions, placement of objects, signs to observe, etc.). However, not all exercises are needed for each reconfiguration (change in the robot's configuration or skills).

Therefore, in the second iteration in the modelling we focused (1) on the exercises and tasks for a given lane and (2) on the skills and components of the robot (Figure 29).

Figure 29. Modelling in the demonstrator: context (left) and configuration (right)

Focusing on exercises and tasks: Figure 30 presents the artefacts created for modelling the context of the robot. In the central part the context metamodel can be seen with elements like lane, exercise or tasks (tasks are a valid combination of exercises along a path for the robot). In the lower left part an excerpt from the mapping model is depicted connecting the exercises in the test lane and the skills of the robot.

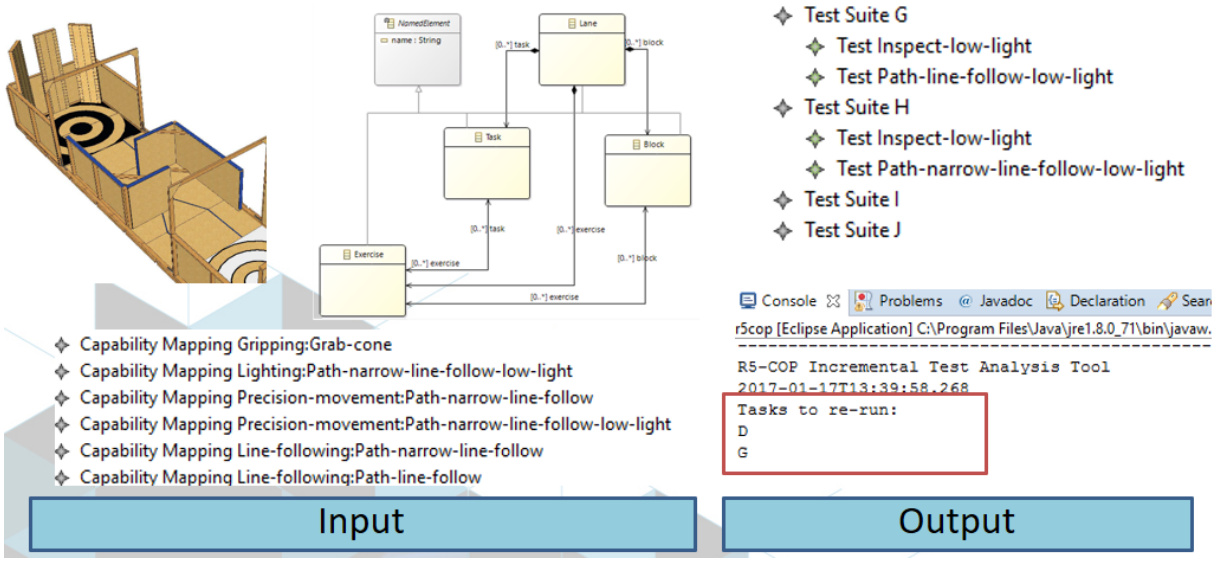

Figure 30. Context models created for the demonstrator and result of test analysis

Focusing on skills and components: Figure 31 presents the artefacts created for modelling the configuration of the robot. The revised metamodel in the centre part contains just high-level elements (mount point, skill component). An instance model of this metamodel should capture the actual configuration of the model (lower left). After changing some part of the model (e.g., modifying a component), the tool calculates all affected elements (e.g., skills affected by the component), and based on the context model descriptions selects those tasks that need to be retested (at the minimum). As it can be seen from the tool's output in the lower right part, the tool can detect that with the currently modelled tasks some of the skills cannot be tested and records these uncovered elements. Moreover, the tool can be parametrized to select only one test for a given component or select all the tests relevant for the component. This can be useful to balance the testing effort and the confidence gained from testing.

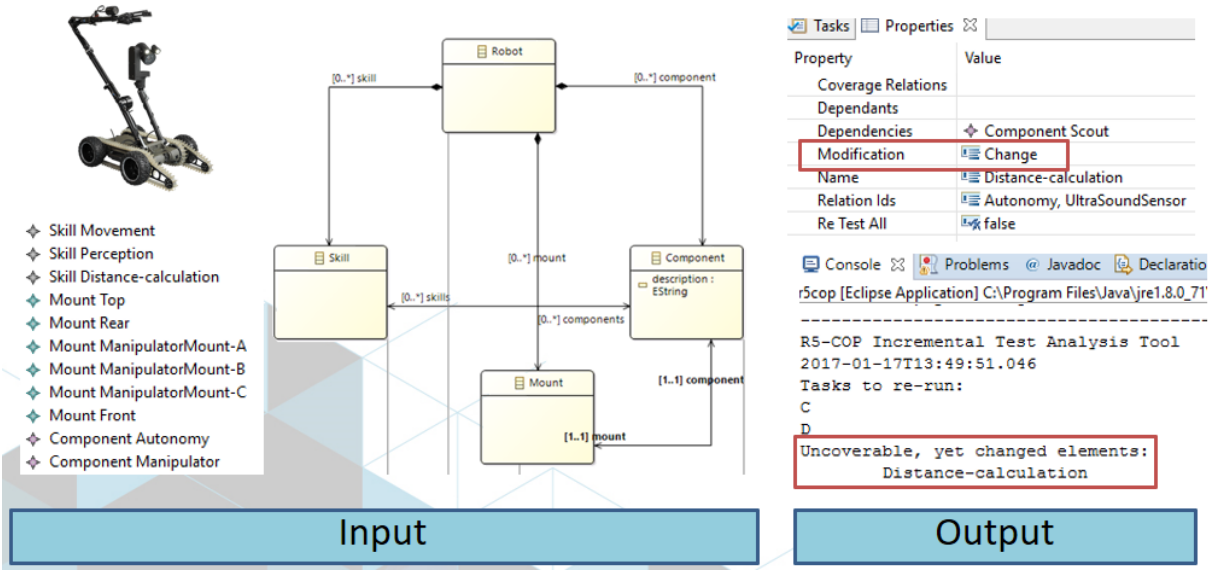

Figure 31. Configuration models created for the demonstrator and result of test analysis

5.3.2 The Test Classification Framework

This section briefly summarizes the capabilities of the tools developed for test classification based on previous experiments.

Support to the designer/user: The usage of the test classification framework can be separated into two distinct phases:

I. Define what to model, create metamodel to capture concepts and create initial instance models (by robot designers).

II. Modify the instance models according to the current change or reconfiguration, calculate test classification (by robot user).

The first phase is done usually in collaboration between modelling and domain experts (like in the case of the demonstrator with BME and PIAP), while in the second phase the created models can be modified and the tool can be used without extensive modelling or Eclipse expertise.

The following steps need to be performed in the first, preparation phase:

1. Creating the domain metamodels for test context, configurations or any other testing artefacts relevant for the test classification. Models need to be created as plain EMF models either with the Ecore tree editor or with Sirius graphical editor. Figure 32 presents an example screenshot for creating the robot configuration metamodel.
2. Creating the queries and code for transforming metamodel elements to the internal test model representation (this defines what is considered a test or a testable element in the domain metamodels and how are they connected). Transformations are currently defined using the VIATRA model transformation framework[3].
3. Finally, initial instance models could be created to show how the metamodels can be used later by the tool users.

In the second phase, the normal tool usage is as follows:

1. Creating an initial checkpoint (CP) representing the current state of the models (this can be achieved by just pushing a button, see Figure 33).

[3] http://www.eclipse.org/viatra/

2. Modifying the model to perform changes: adding or deleting elements, modifying element properties, etc.
3. After all changes are done, calculating a new checkpoint.
4. Finally, calculating the differences between the two model checkpoints: the tool will identify changed model elements and select necessary tests to cover those elements. This is again done by pushing a button; the user of the test classification tool does not have to be familiar with the internal workings of the tool.

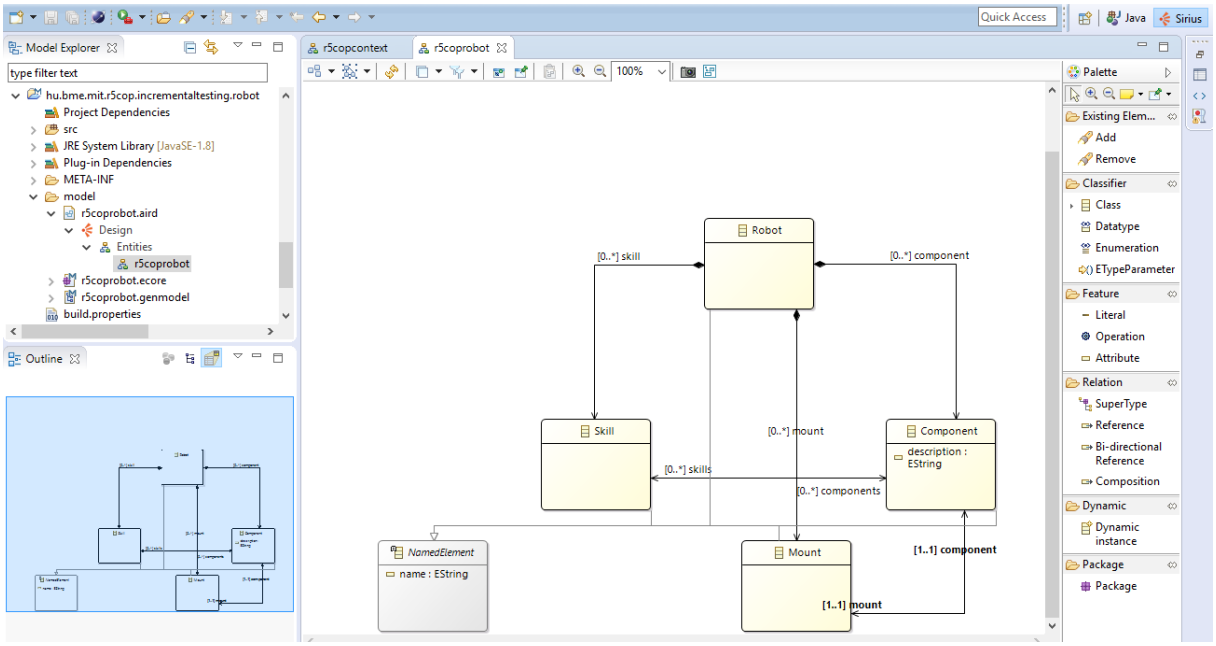

Figure 32. Creating the robot configuration metamodel

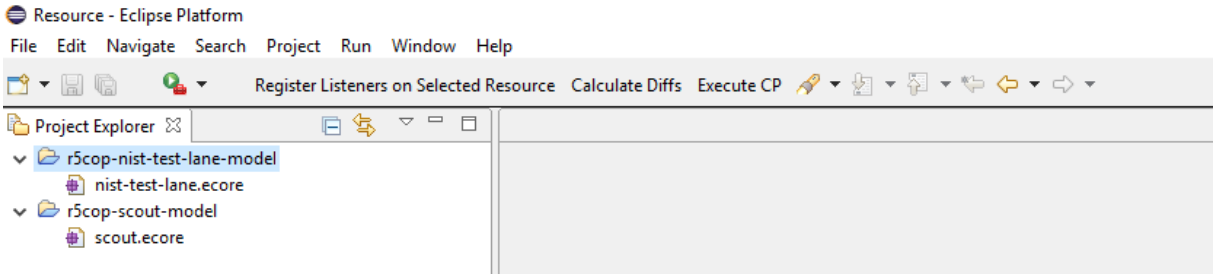

Figure 33. User interface of the tool: “Execute CP” and “Calculate Diffs” buttons

Genericity: The test classification framework offers common tools for different models. The domain metamodels can represent anything relevant from a testing point of view. For example, in case of the PIAP demonstrator models can represent tasks and exercises in a physical test room, while for other robots models can also represent test context data for simulated or real environments (see in Section 5.3.3).

Scalability: Scalability of tools was analysed in deliverable D34.20 in detail. Figure 34 presents some of the main findings. The tool was able to handle models with 500 elements and changes with 30 elements in seconds. The models created for the demonstrators consisted of usually 30-50 elements, therefore this order of magnitude for the handled elements is more than satisfactory for the envisioned use cases. Moreover, the underlying technologies (Eclipse EMF, VIATRA and VIATRA Query) proved to scale to thousands of model elements, therefore we see no threats towards scalability of the tooling.

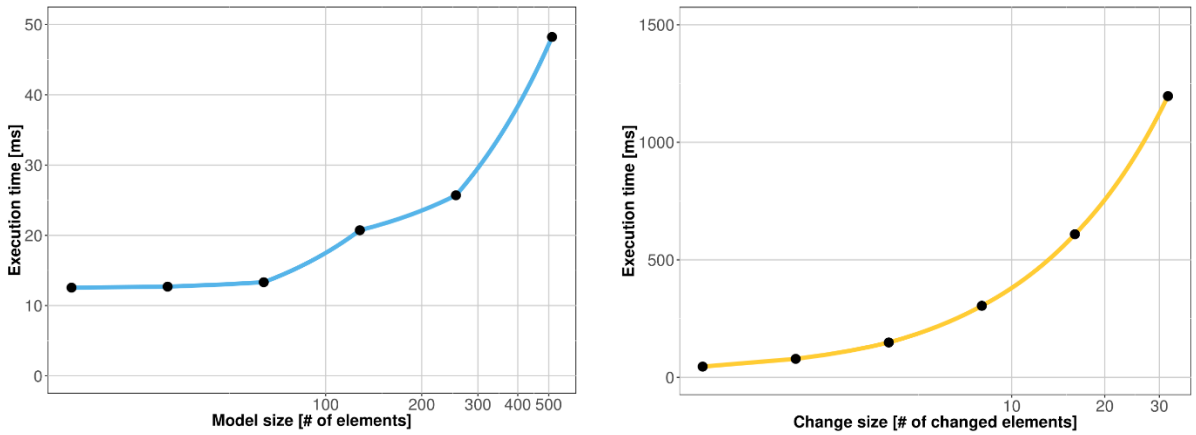

Figure 34. Scalability assessment of the incremental testing tool

5.3.3 The Test Context Generator Tool

In Section 5.3.2 we mentioned that the Test Classification Framework can be used together with test context models by analysing the changes and triggering the generation of new test contexts. In this section we present the related Test Context Generator tool that constructs test context models specifying setups for real or simulated environments. Figure 35 overviews the main steps of the test context generation approach.

In the first step of the test context generation workflow, the test engineers have to define the following artefacts:

- *Context information*, i.e., the relevant environmental and physical configuration that form the context of the system under test.
- *Coverage criteria* which specify the required properties of the generated test suite, for example to cover potential types of obstacles that can occur in the environment.
- *Test objectives* to express the properties which should be satisfied by the test suite, for example the obstacle avoidance in case of multiple obstacles with the minimum path of the robot.

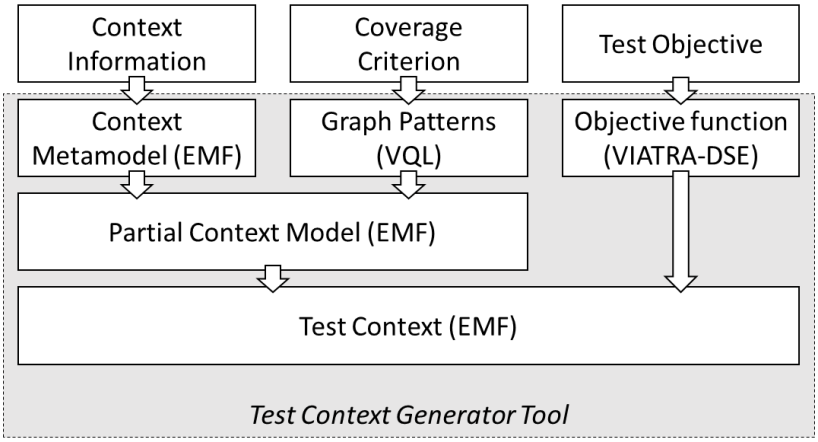

Figure 35. Test context generation approach

Context information defines the types of objects of the environment such as furniture, obstacles, actors etc. and their properties. In addition, context information defines also the basic

constraints how all these objects can be arranged. The test generator tool supports the definition of the context information in the form of an EMF metamodel in the Eclipse framework.

Coverage criteria can be provided for the system under test in order to define which parts from the context model should be included in the test suite (while the other parts will only be covered in an ad-hoc manner). Coverage criteria can be expressed with the help of the VIATRA Query language by defining patterns: the models resulting from the test generation will cover the possible instantiations of the prescribed patterns.

Test objectives define the property of the test suite so that the test generation algorithm will try to minimize the value of this property. This helps guiding the test generation algorithm to provide relevant and low cost tests for the system. A test objective can be defined as result of a *test objective function* which is specified using the VIATRA Query language and Java. Various kinds of test objective skeletons are provided, this way even complex objective functions can be defined with relatively small effort.

As the next step in the test context generation workflow, the partial context models are generated. This set of models is conformant to the context metamodel and they fulfil the coverage criteria. From this set of partial models, the test generation algorithm generates the test contexts according to the test objective function. The result of the procedure is a set of test context models which satisfy the coverage criteria and minimize the objective function.

As an example, let us consider a test context generation problem in case of the WP44 demonstrator (MIR robot that shall autonomously drive into an elevator).

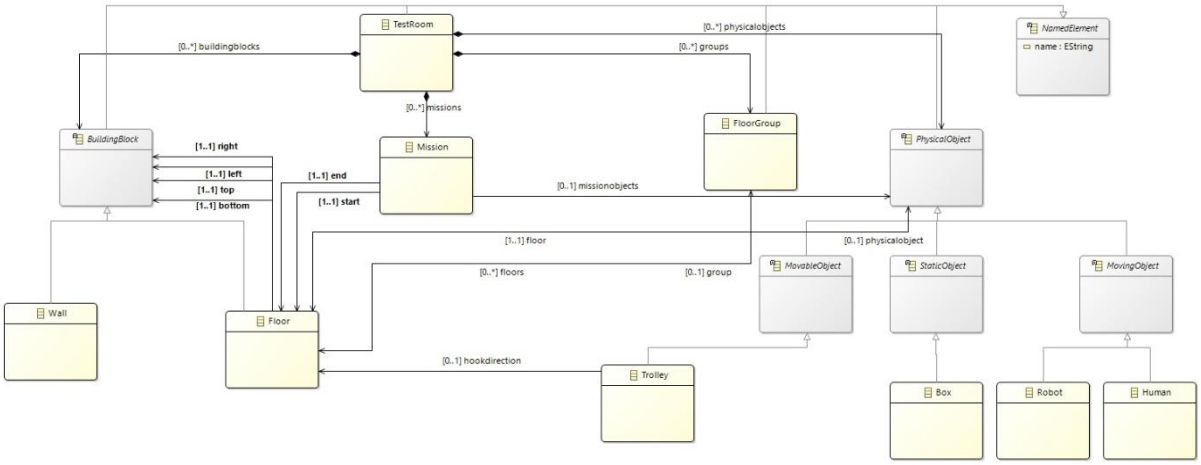

Figure 36. Test context metamodel

Figure 36 presents the context in form of an EMF metamodel. It defines the basic building blocks of the environment such as floors, walls and obstacles such as boxes, robots, trolleys and humans. In addition, containment and other structural constraints are also represented in the metamodel.

On the basis of this metamodel, we can specify coverage criteria with the help of the VIATRA Query language. As for now our goal is to cover all the arrangements in an elevator (into which the robot shall drive), we defined the VIATRA Query pattern depicted in Figure 37. This pattern will help us generate possible elevator configurations in which there are two boxes (obstacles) in the elevator at various places.

```
pattern initElevatorConfigurations(tr : TestRoom, b1 : Floor, b2 : Floor, end : Floor){
    find elevatorFloorPairsForPOs(tr, b1, b2);
    find elevatorEndPoints(tr,end);
    end != b1;
    end != b2;
}
```

Figure 37. Coverage criterion expressed as context pattern

As test objective function, we show the generation of test context models which try to minimize the number of places that were not reached by the robot, in other words, to maximize the trajectory of the robot by placing objects into its way. In Figure 38, the definition includes the following: (1) generate test models trying to maximize the length of the robot trajectory to reach its goal, (2) use boxes to prevent the robot in reaching its goal, but (3) limit the number of boxes.

```
dse.addObjective(Objectives.createCompositeObjective("Composite")
    .withObjective(new StartEndAccurateDistanceObjective("StartEndDistance"), 1000)
    .withObjective(Objectives.createConstraintsObjective("PhysicalObjectCount").withSoftConstraint(IsPhysicalObjectQuerySpecification.instance, 10))
    .withObjective(new NumberOfPhysicalObjectsObjective("NumberOfBoxes", 4), 100)
    .withComparator(Comparators.HIGHER_IS_BETTER))
```

Figure 38. Definition of the test objective functions

On the basis of the context metamodel (Figure 36), the coverage criterion (Figure 37) and the test objective functions (Figure 38), the test context generation algorithm produces more than 126 different test setups. Some of them are depicted in Figure 39. The graphical syntax of the test contexts can be interpreted as follows: the robot pictogram represents the initial place of the robot (to start its mission), the elevator space is at the bottom of the figure (8 places) with an elevator sign representing an admissible goal place, boxes represent physical obstacles the robot has to avoid. All the test contexts have different elevator configurations.

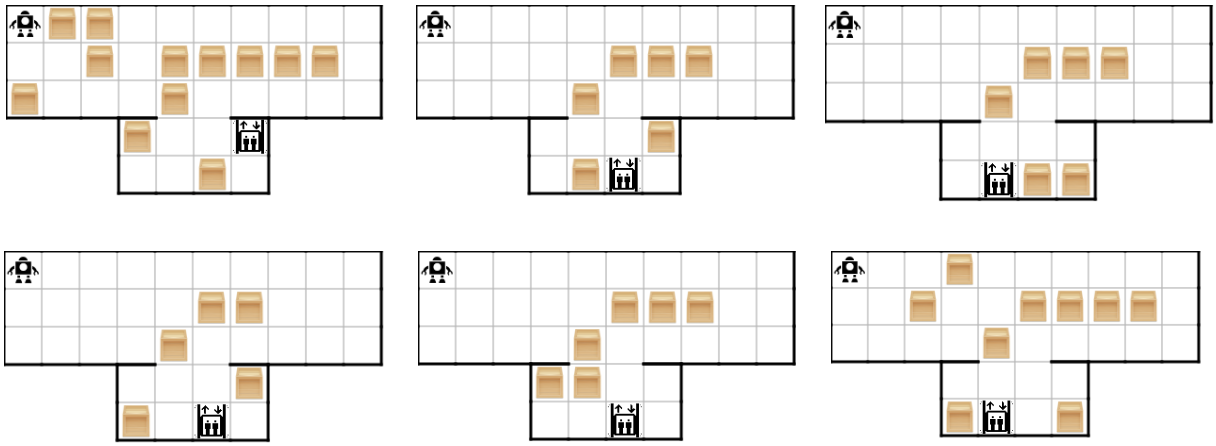

Figure 39. Test context models

Using different test objective functions various other test goals can be supported. The generated test contexts can also be inspected by test engineers to select situations which seem to be interesting for simulator based testing or testing in a real physical environment. In case of changing the metamodel, the existing test suite can be classified by the Test Classification Framework in order to identify obsolete test contexts.

5.3.4 Efficiency of Incremental Testing (An Example)

In order to show the efficiency of incremental testing, we present an example that emphasizes the importance of test selection in case of changes in the robot.

The context contains three main blocks as shown on the left of Figure 29. The middle element is a path that can have six different tasks on it:

- line following,
- narrow line following,
- zigzag,
- low light line following,
- low light narrow line following,

- low light zigzag.

Each of them exercises a slightly overlapping set of skills. In the start and end blocks, exercises are placed as well that must be finished for successful testing. These can be the following three:

- inspection,
- inspection in low light conditions,
- grabbing a cone.

A full test execution in this context requires re-testing all combinations of these exercises, which would mean re-testing of 54 tasks (3 different tasks on start and end blocks, and 6 different tasks on the middle block). This would require significant amount of time and effort to perform, even not counting the fact that changing the layout of the test room also requires a measureable effort.

The example *instance model of the context* can be found on Figure 40.

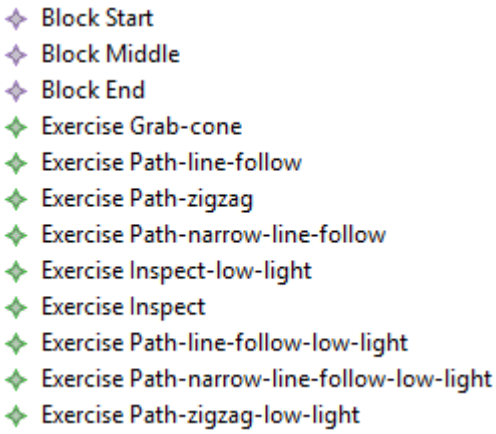

Figure 40. Example context instance model

The presented approach requires an *instance model of the robot* on which components can be changed that trigger re-execution of some tests. Then, these components must be mapped to *skills* of the robot. These skills are connected to specific exercises in the context. For example, the cone grabbing exercise uses the manipulator arm with the gripper. For the purpose of this example we used a robot instance model with various components and skills (e.g., gripping skill – gripper arm, distance calculation skill – ultrasound sensor, etc.). Also, for example purposes, we created a mapping between the skills and the exercises found in the context. The mapping connects gripping skill of the robot with the cone grabbing exercise.

Let us consider a development scenario, when the gripper on the robot is changed. Without incremental test selection, this would trigger re-execution of all 54 tasks. However, employing our approach on this problem may tackle this by reducing the number of tests to execute. The approach involves two different kinds of test selection procedures as mentioned previously. In the current case this would mean the following.

- Without test selection: *re-execution of 45 tasks*
- With test selection (re-test one): *re-execution of 1 task (*e.g., inspect – zigzag – grab)
- With test selection (re-test all): *re-execution of 18 tasks* (all tasks where grabbing can be an exercise on one of the blocks).

6 Standardization Aspects

In this chapter the standardization and certification related aspects of the methods are summarized. Standardization is especially important in case of safety-critical systems where the development process shall typically follow the requirement of standards in order to assure certification.

In general, safety certification follows two complementary approaches:

- The *standard-based approach* means that the designer is recommended or required to follow certain guidelines. These specify the development and verification & validation techniques that should be used, the intermediate artefacts to be produced (specifications, designs, test plans etc.), the kinds of reviews, tests, and analyses that should be performed, and the corresponding documentation.
- The *safety case approach* provides an *argument* that a system is safe. The notion of "safe" is made precise in suitable *claims* about the system and its context, and the argument is intended to support these claims, based on *evidence* concerning the system, its design, implementation, verification and validation. The approach can be applied recursively, so that claims about subsystems can be used as evidences in an upper level argumentation.

Note that these approaches are not fundamentally different as the prescriptions of standards and guidelines can be considered as constructing a generic safety case: the required documentation of the processes and artefacts for a given system provides the evidence for an "instantiation" of this generic safety case, and the argumentation is implicit in the standards-based approach. Standards are often considered as conservative and not well-tuned to novel characteristics of systems like context-awareness, adaptiveness, etc. An explicit safety case can be customized very precisely for a given system, and may provide assurance at lower cost than a standards-based approach. However, systematic processes and well-defined artefacts are needed to provide confidence in the soundness of a given safety case – this is where systematic procedures and techniques based on formal models come into consideration, like in our case model-based incremental testing, and on-line verification with monitors constructed on the basis of formalized properties.

In the following first the role of incremental testing and on-line verification in standard-based approach is discussed (Section 6.1 and 6.1), then the on-line verification is considered as providing an evidence in a safety case, leading to the idea of "runtime certification" (Section 6.3).

6.1 Incremental Testing in Safety Standards

The notion of "incremental testing" is not included explicitly in standards. However, in case of changes and modifications in a system, *regression testing* is mentioned as one of the related techniques. Efficient regression testing needs similar test classification techniques that are developed in WP34.

IEC 61508, the basic standard for functional safety of electrical / electronic / programmable electronic safety related systems includes the concept of *regression validation* that is recommended (R) for safety integrity level (SIL) 1, and highly recommended (HR) for SIL 2, 3, and 4 (see Part 3, Requirement 7.8.2, Table A.8 – Modification, and Part 7, C5.25).

In case of software modification, *regression testing and verification* is required, and regression validation is used to ensure that valid conclusions are drawn from regression testing. It is admitted that complete regression testing of large or complex system usually requires much effort and time. When possible, it is desirable to restrict the regression testing to cover only the system aspects of direct interest (e.g., affected by a modification). In this partial regression testing scenario it is essential to have a clear understanding of the scope of the partial testing and to draw valid conclusions regarding the tested status of the system.

Regarding our approach (Section 5), the following can be highlighted:

- Our approach to test classification directly supports the requirement to have a clear understanding of the scope of the partial testing. Having performed the impact analysis, that is necessary part of regression validation, the model based test classification framework can precisely represent and automatically select the test cases that are linked to the software parts that are impacted (and this way these tests shall be re-executed). Tests classified as redundant can be omitted this way reducing the efforts and time of re-testing.
- The role of test classification framework is extended: not only the changes in software parts, but also changes in the context and configuration can be handled and the related tests can be classified. This way regression testing is supported even in the case when there is no change in the software source code but in the environment assumptions and component configuration.
- Considering the properties for systematic safety integrity, it is noted that evaluation of results and regression test suites is a key benefit of model based testing (MBT). In our solution, MBT approach was followed also in case of context and configuration modelling and the related test generation and test classification. As rigorous modelling approach was applied with regard to context and configurations, objective evidence of coverage is possible. By using the test classification and executing tests classified as re-testable, coverage is retained even in case of changes.

6.2 On-Line Verification in Safety Standards

On-line verification is a classic method that is included in safety standards. IEC 61508, the basic standard for functional safety of electrical / electronic / programmable electronic safety related systems contains several requirements and techniques that are related to the concept of on-line monitoring.

For the control of random hardware failures:

- Part 7, Technique A.1.1. Failure detection by on-line monitoring. To detect failures by monitoring the behaviour of the E/E/PE safety-related system in response to the normal (on-line) operation of the equipment under control (EUC).
- Part 7, Technique A.6.4: Monitored outputs: To detect individual failures, failures caused by external influences, timing failures, addressing failures, drift failures and transient failures.
- Part 7, Technique A.9.3: Logical monitoring of program sequence: To monitor the correct sequence of the individual program sections.
- Part 7, Technique A.9.4: Combination of temporal and logical monitoring of program sequences: To monitor the behaviour and the correct sequence of the individual program sections.
- Part 7, Technique A.9.5: Temporal monitoring with on-line check: To detect faults by temporal monitoring.

For achieving software safety integrity:

- Part 3, Requirement 7.2.2.8: The software safety requirements specification shall consider (among others) software self-monitoring.
- Part 3, Requirement 7.4.2.7: The software design shall include, commensurate with the required safety integrity level, self-monitoring of control and data flow. On failure detection, appropriate actions shall be taken.
- Part 3, Table A.2 for software design and development - software architecture design: Diverse monitoring techniques are recommended (R) for SIL 2 and SIL 3, and (on separated computer) highly recommended (HR) for SIL 4.

- Part 7, Technique C.3.4: Diverse monitoring: It is used to protect against residual specification and implementation faults in software which adversely affect safety.

It is noted that diverse monitoring techniques (with independence between the monitor and the monitored function in the same computer) increase software complexity.

Considering properties for systematic safety integrity, the application of diverse monitor techniques on the same computer are characterized (with medium level of rigour, i.e., with objective acceptance criteria that give a high level of confidence that the property is achieved) as providing freedom from intrinsic design faults, simplicity and understandability, predictability of behaviour, and verifiable and testable design by implementing the minimum safety requirements.

Regarding our approach (Section 3), the following can be highlighted:

- Our approach implements diverse monitoring because the monitor is generated on the basis of a property specification that is separated from the design models. Using our monitor, checking of events related to program sequence, inputs and outputs can be achieved, combined with temporal monitoring. Our method even extends this by offering the monitoring of context dependency and configuration dependency.
- Another characteristic of our approach is the use of engineering languages with formal semantics, to specify the properties to be monitored and to form the basis of automated monitor synthesis. According to the standard, the application of formal methods is characterized with completeness and correctness with respect to the safety needs to be addressed, freedom from intrinsic specification faults including freedom from ambiguity, understandability of safety requirements (as the languages are user- and application-friendly), and capability of providing a basis for verification and validation.

6.3 Run-Time Certification

If the overall argumentation in a safety case is sound, it allows focusing on the evidences and assumptions that support the argument. The validity of certain kinds of evidences and assumptions can be assured by monitoring these at runtime. Moreover, if these assumptions and properties are formalized then construction of monitors can be automated, leading to the idea of *runtime verification*.

Runtime verification can be used in the construction of evidence and argumentation with respect to explicit safety goals. This approach of *runtime certification* is a relatively recent initiative [22]: monitors that guarantee certain properties can be considered as evidence for the assurance case.

This approach is especially usable in case of adaptive robotic systems operating in dynamic environments, where the monitors can detect any anomalies, invalid assumptions, or violation of essential safety properties. Thus reconfigurable, resilient, reasoning robotic systems (R5-COP systems) present a new application area for run-time verification and certification, especially focusing on reconfiguration and fault handling policies. The methods developed in WP34 may effectively support this approach.

7 Conclusions

R5-COP progressed beyond the state-of-the-art by the provision of new V&V techniques that may also support safety certification. In concordance with the Technical Annex (Description of Work), the main results are as follows:

- *On-line (runtime) verification of reconfigurable systems*. The difficulties of verifying a great variety of interactions and the adaptive behaviour are resolved by proper integration of design-time and run-time verification activities: design-time verification techniques are extended with novel run-time verification techniques that focus on monitoring those properties, especially robustness and safety ones, that cannot be guaranteed by design time verification due to unpredictable environment, variability of interactions, and run-time faults. The design time modelling and requirement specification formalisms are adapted and extended to express the properties to be monitored and this way to support the automated synthesis of monitors that are responsible for application monitoring, state evaluation, checking reconfiguration processes, and supervising adaptive fault handling. The on-line verification open ways towards runtime certification, in which certification related safety goals, evidences and arguments are supported by the on-line verification activities.
- *Retesting of reconfigured systems*. In the case of reconfigurable systems, test optimization regarding the re-testing of a new version can be supported by systematically re-using and adapting existing test cases that were developed for another configuration of the same system. Selection, adaptation, and extension of test cases are supported by a novel model based method that is able to analyse the changes in requirements (formalized in requirement models), the changes in the context of the system (formalized in context models), and the changes in the internal components (formalized in configuration models). On the basis of this incremental analysis, redundant test cases, and gaps in the existing test suites are identified according to predefined coverage metrics, and this way the test adaptation and generation of new tests are triggered. This new method can be used in the design phase (to check configuration possibilities) as well as in maintenance phases (to check the behaviour of a concrete reconfigured version).

8 References

[1] Harel, D. and Thiagarajan, P. S.: Message sequence charts. In UML for real, pp 77-105. Kluwer Academic Publishers, 2003.

[2] Damm, W. and Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design, 19(1):45-80, 2001.

[3] Autili, M., Inverardi, P. and Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Eng., 14(3):293-340, 2007.

[4] R3-COP Consortium: Deliverable D4.2.1 “Models, Languages and Coverage Criteria for Behaviour Testing of Individual Autonomous Systems – Part I: Behaviour Testing”. April 30, 2013.

[5] R3-COP Consortium: Deliverable D4.2.2 “Behaviour Testing Strategies and Test Case Generation – Part I: Behaviour Testing”. October 31, 2013.

[6] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C.: Property Specification Patterns for Finite-state Verification. In Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP), pp 7-15. ACM, 1998.

[7] About Specification Patterns. http://patterns.projects.cis.ksu.edu/ (accessed on January 6, 2015).

[8] Pnueli, A: The temporal logic of programs. Foundations of Computer Science, 18th Annual Symposium, pages 46–57, 1977.

[9] Misra, J. and Roy, S.: A Decidable Timeout based Extension of Propositional Linear Temporal Logic. ArXiv preprint, (1012.3704):1–29, 2010.

[10] Pintér, G. and Majzik, I.: Automatic generation of executable assertions for runtime checking temporal requirements. In Proc. of the 9th IEEE Int. Symposium on High-Assurance Systems Engineering (HASE 2005), pp 111–120, IEEE CS, 2005.

[11] Decker, N., Leucker, M. and Thoma, D.: Monitoring modulo theories. Int. Journal on Software Tools for Technology Transfer, pp. 1–21, Springer, 2015.

[12] Bauer, A., Leucker, M. and Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput., vol. 20, no. 3, pp. 651–674, 2010.

[13] Bauer, A., Leucker, M. and Schallhart, C.: Monitoring of real-time properties. In Proc. 26th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2006), LNCS 4337. pp. 260–272, Springer, 2006.

[14] Barringer, H., Rydeheard, D. E. and Havelund, K.: Rule systems for run-time monitoring: From Eagle to Ruler. In Proc. 7th Int. Workshop on Runtime Verification (RV 2007), Vancouver, Canada, March 13, 2007, LNCS 4839. pp. 111–125, Springer, 2007.

[15] Bauer, A., Leucker, M. and Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Software Eng. Methodology, vol. 20, no. 4, p. 14, 2011.

[16] Horányi, G., Micskei, Z. and Majzik, I.: Scenario-based Automated Evaluation of Test Traces of Autonomous Systems. In Proc. Workshop on Dependable Embedded and Cyber-physical Systems (DECS@SAFECOMP 2013), Toulouse, France, 2013.

[17] Messmer, B. T., Bunke, H.: Efficient Subgraph Isomorphism Detection : A Decomposition Approach. Knowledge Creation Diffusion Utilization, 12(2):307–323, 2000.

[18] Horányi, G.: Monitor synthesis for runtime checking of context-aware applications. Master’s thesis, Budapest University of Technology and Economics, 2014.

[19] Hélene Waeselynck, Zoltán Micskei, Nicolas Riviere, Áron Hamvas, Irina Nitu: TERMOS: a Formal Language for Scenarios in Mobile Computing Systems. In Proc. 7th International ICST Conference on Mobile and Ubiquitous Systems (MobiQuitous 2010), Sydney, Australia, 6-9 December 2010.

[20] J. Klose: Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior. PhD thesis, C. v.O. Universitat Oldenburg, 2003.

[21] B. Dutertre and M. Sorea, Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol using Calendar Automata. In Proc. FORMATS/FTRTFT'04, Grenoble, France, September 2004.

[22] J. Rushby: Runtime Certification. In Proc. 8th Workshop on Runtime Verification (RV08), Springer Verlag Lecture Notes in Computer Science, vol. 5289, pp. 21{35. 2008.

[23] O. Sokolsky, K. Havelund, I. Lee: Introduction to the special section on runtime verification. In: International Journal on Software Tools for Technology Transfer, June 2012, Volume 14, Issue 3, pp 243-247, Springer Verlag, 2012.

[24] D. W. Pickton, S. Wright: W'at's SWOT in strategic analysis?. Strat. Change, 7: 101–109. DOI: 10.1002/(SICI)1099-1697(199803/04)7:2<101::AID-JSC332>3.0.CO;2-6, 1998

[25] K. Havelund, G. Rosu: Testing Linear Temporal Logic Formulae on Finite Execution Traces. Technical report, RIACS, 2000.

[26] B. Finkbeiner, H. Sipma: Checking Finite Traces Using Alternating Automata. In Formal Methods in System Design, 24(2):101-127, 2004.

[27] http://se.mathworks.com/company/user_stories/festo-develops-innovative-robotic-arm-using-model-based-design.html

[28] M. Saggiani, R. Pretolani, B. Teodorani, F. Zanetti: Developing a hardware-in-the-loop simulator for the rotary wing unmanned aerial vehicle, University of Bologna, School of Engineeting Forli, http://sine.ni.com/cs/app/doc/p/id/cs-12188

[29] K. Lamberg, P. Wältermann: Using HIL Simulation to Test Mechatronic Components in Automotive Engineering. dSPACE GmbH, Paderborn, http://www.dspace.de/ftp/papers/HdT00_e.pdf

[30] [25] S. Yoo, M. Harman. Regression testing minimization, selection and prioriza-tion: a sur-vey. STVR 22:67-120, 2012. DOI: 10.1002/stvr.430

[31] [26] National Institute of Standards and Technology. Guide for evaluating, purchasing, and training with response robots using DHS-NIST-ASTM international standard test meth-ods. Technical report [Online]. Available: http://www.nist.gov/el/isd/ms/upload/DHS NIST ASTM Robot Test Methods-2.pdf

[32] [27] National Institute of Standards and Technology. Counter-Improvised Explosive Device Training Using Standard Test Methods for Response Robots. Document 18370 [Online], Available: https://www.nist.gov/document-18370

9 Appendix A

The integration of the monitor into the Scout robot needed the following functions:

```
std::mutex mutex;
std::map<std::string,std::shared_ptr<bool>> timers;
extern std::unique_ptr<ros::Publisher> publisher;

void timer(std::shared_ptr<bool> active, int timeout, std::function<void()>
callback)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(timeout));
    {
        std::lock_guard<std::mutex> lock(mutex);
        if (!*active)
        {
            return;
        }
    }
    callback();
}

void setTimeout(const char* event, int timeout)
{
    if (timers.find(event) != timers.end() && *timers[event])
    {
        std::cerr << "Overwriting timer: " << event << std::endl;
        cancelTimeout(event);
    }
    timers[event] = std::make_shared<bool>(true);
    std::string name(event);
    auto callback = [name](){ evaluate(name.data()); };
    std::thread t(timer, timers[event], timeout, callback);
    t.detach();
}

void cancelTimeout(const char* event)
{
    if (timers.find(event) == timers.end())
    {
        std::cerr << "Timer does not exists: " << event << std::endl;
        return;
    }
    std::lock_guard<std::mutex> lock(mutex);
    *timers[event] = false;
}

void errorAction(const char* current, const char* last)
{
    if (publisher)
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "Event '" << current << "' is not allowed after '" << last << "'";
        msg.data = ss.str();
        publisher->publish(msg);
    }
}
```