Grant agreement no.	**621447**
Project acronym	**R5-COP**
Project full title	**Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems**
Dissemination level	PU
Date of Delivery	03/03/2015
Deliverable Number	D34.10
Deliverable Name	**Languages and formalisms for expressing properties for on-line and off-line verification**
WP / Task related	WP34 T34.1
Author	BME
Contributors	BME: Istvan Majzik, Zoltán Micskei, Tamás Tóth, András Vörös, Dániel Darvas, Gergő Horányi, Zoltán Szatmári
Keywords	Property description language, safety rules, function contracts, temporal properties, runtime verification, incremental testing
Abstract	This deliverable aims at the selection and definition of description languages that can be used for (1) capturing the properties to be checked by on-line verification and (2) describing the relation of components, properties and test cases for incremental testing. These languages allow the formalization of capabilities and restrictions, safety rules, function contracts, temporal or trace-based reference behaviour, as well as test coverage with respect to components and specified properties.

Document History			
Ver.	Date	Changes	Author
0.1	11/11/2014	Initial structure of the content	I. Majzik (BME)
0.2	09/01/2015	Integration of contribution about incremental testing	Z. Micskei (BME)
0.3	21/01/2015	Integration of contribution about describing reference automata	T. Tóth (BME)
0.4	23/01/2015	Integration of contributions about existing solutions	All (BME)
0.45	26/01/2015	Integration of contributions about the overview of the languages	All (BME)
0.5	28/01/2015	Integration of contribution about code contracts	A. Vörös (BME)
0.55	30/01/2015	Integration of contributions about context and scenario modelling	Z. Szatmári, Z. Micskei, I. Majzik (BME)
0.6	03/02/2015	Integration of contributions about configuration description	A. Vörös (BME)
0.65	04/02/2015	Integration of contribution about CaTL, scenarios and patterns	G. Horányi, I. Majzik (BME)
0.7	06/02/2015	Integration of contribution about property description languages	D. Darvas, I. Majzik (BME)
0.75	11/02/2015	Integration of the additional contributions	All (BME)
0.8	13/02/2015	Modifications after discussion	All (BME)
0.9	16/02/2015	Document is ready for internal review	All (BME)
0.9r	27/02/2015	Internal review of the document	R. Lill (FAU)
1.0	03/03/2015	Corrections after the internal review	All (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 images

List of tables

List of Acronyms

CaTL	Context-aware Timed Propositional Linear Temporal Logic
CTL	Computational Tree Logic
EMF	Eclipse Modeling 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 Modeling 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, this way also supervising error handling and self-healing policies.

- Incremental testing of the behaviour of reconfigurable systems is relevant in the design phase (to check configuration possibilities) and in maintenance phases (to check the behaviour of a concrete reconfigured version). Incremental testing can build on the already existing test suites (that were developed for the previous versions) and the results of these previous testing activities.
- 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, *description languages are needed* to capture those properties of the system that characterise its correct behaviour. These formalized properties provide the basis for monitoring and incremental test selection:

- For incremental testing, the description of the relation of test cases, system components and properties is used by the *methods and tools for selecting, adapting and extending test cases from existing test suites* in an incremental way, in order to check the changed components or properties. Accordingly, the language captures the new requirements (formalized in scenarios), the changes in the context of the system (formalized in context ontologies and metamodels), and the changes in the internal components (formalized in architecture and capability models).
- For runtime monitoring, the description of the properties is used by the *methods and tools for monitor synthesis*, i.e., an automated construction of software monitors to check the specified system properties. Accordingly, the language captures safety and liveness properties (for monitoring safe and correct behaviour during execution), and function contracts (formalized using assertions, temporal and trace-based properties).

1.2 Purpose of Document

This deliverable aims at the selection and definition of description languages that can be used for (1) capturing the properties to be checked by on-line verification and (2) describing the relation of components, properties and test cases for incremental testing. These languages allow the formalization of capabilities and restrictions, safety rules, function contracts, temporal or trace-based reference behaviour, as well as test coverage with respect to components and specified properties.

As mentioned above, these languages form the basis of the following tasks of WP34:

- *Task 34.2: Incremental testing of behaviour*. In this task, a method will be elaborated that can be used for the selection, adaptation and extension of test cases. 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.

- *Task 34.3: Design of the monitoring infrastructure.* The monitoring infrastructure comprises the monitor components that perform the on-line verification and the mechanisms for accessing the monitored information. The algorithms of these monitors and procedures for their synthesis are derived on the basis of the semantics of the formalized properties to be checked.

1.3 Partners Involved

Partners and Contribution	
Short Name	**Contribution**
BME	Selection and definition of description languages
FAU	Review of the document

2 The General Context

This section describes the general concept of incremental testing and runtime verification in order to put into context the selection and definition of the description languages presented in the subsequent sections of the document.

2.1 Incremental Testing of Autonomous Robots

To create reconfigurable robotic systems, not only the development but also the verification and testing activities have to take into account reconfiguration and changes. Time and resources required for testing can be reduced if testing is performed incrementally.

In classical software engineering terminology, the re-use of previous tests and test results is denoted as “regression testing”. Here “incremental testing” is used to address the stepwise extension/change of the functional scope of the subject under test throughout successive testing phases. To increase efficiency in testing, the previous tests and test results are re-used and testing is focused on the changed part of the reconfigured system.

2.1.1 Basis of the Approach

In the preceding R3-COP project, BME developed a model-based system level testing method for testing the context-aware behaviour of an autonomous robot. The test goal is to check the safe execution of a robot mission (e.g., transportation of goods without collision) in various contexts (e.g., in the presence of obstacles, humans, other robots and various environment objects). Accordingly, test contexts (arrangements of objects, obstacles etc.) shall be constructed systematically. To do this, we model the scenarios (describing the requirements against the robot) and the potential contexts of the robot (environment object types with their relations and constraints). On the basis of these models, our tool generates systematically the models of test contexts in which the mission of the robot can be checked. These generated test context models can be mapped to the configuration of a real test environment, a simulated environment (like in ROS+Gazebo), or internal representation of perceived context of the robot in ROS (depending on the implementation). Various test generation strategies can be supported, like the generation of extreme context for robustness testing.

Figure 1 presents an example from an autonomous forklift. The left hand side depicts the context model representing that a forklift can move on segments and can interact with people, pallets and other forklifts. On the bottom left a scenario states that if a person moves close to the robot (it is in the so called “warning” range), then the robot has to react with an alarm sound. Another requirement is that if the person is too close (in the “danger” range), then the robot has to stop. From these models the test generator tool creates models describing test contexts (one of them is depicted on the bottom of Figure 1). Test context models place different objects and persons around the robot to verify that it can handle multiple, possibly conflicting requirements (e.g., when one person is in warning range, and another is in danger range).

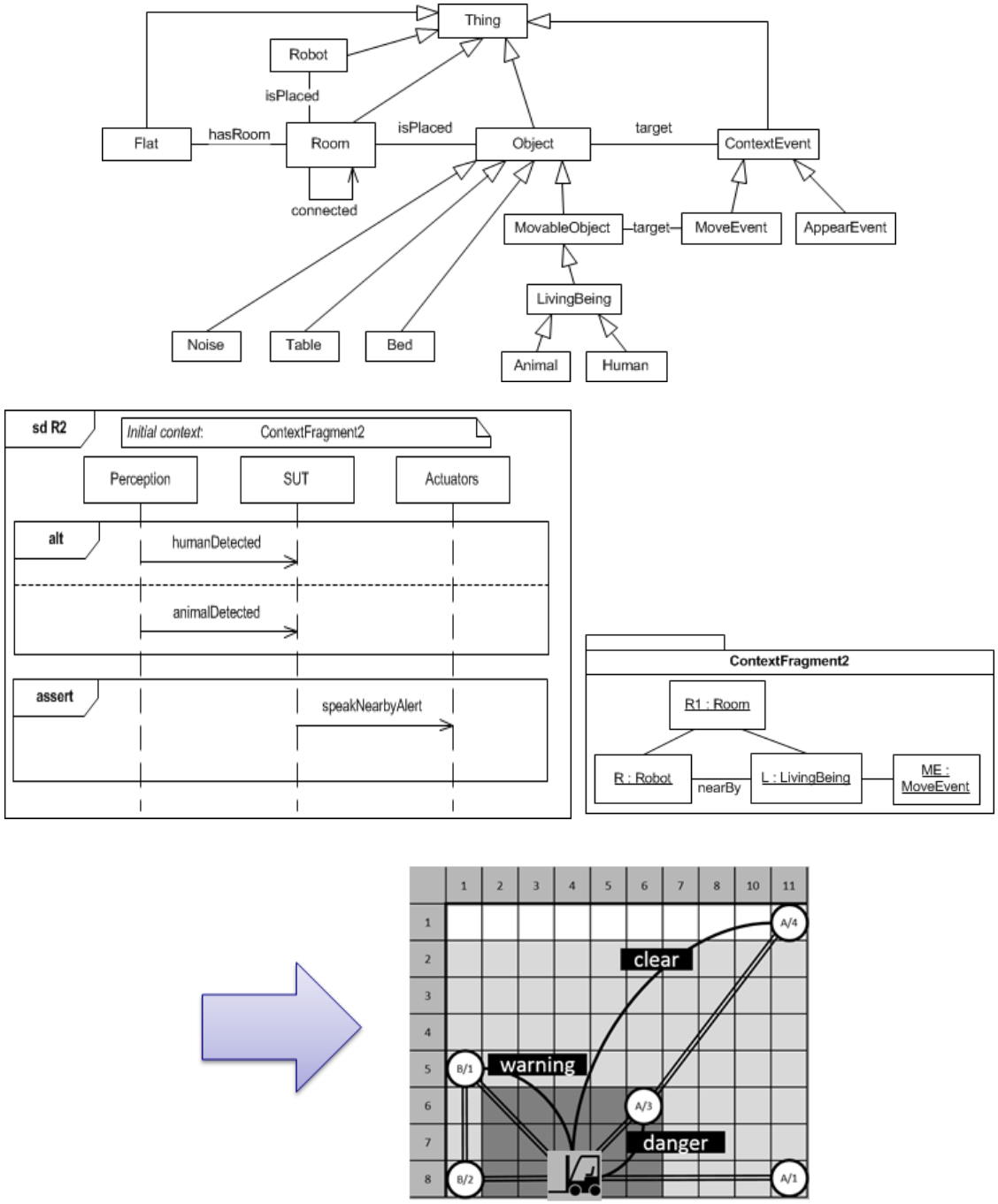


Figure 1. Creating test data (at the bottom) from context models (at the top)

2.1.2 Reconfiguration

In a reconfiguration, the context or the configuration of the robot can change. For example, a new type of object can appear in the environment, a requirement is modified, or a new type of sensor is added to the robot. In these cases the most basic strategy is to run all previous tests (called retest-all). However, this is not an optimal solution, as some of the previous tests are

- reusable (redundant): these tests are using only unchanged part of the system, thus it is unnecessary to run them, valuable test resources and time could be saved,
- retestable: these tests exercise the changed part of the system, thus they need to be re-executed,

- obsolete: these tests are not valid any more (e.g., a component of the robot has been removed),

Moreover, new tests may be needed, i.e., it has to be identified which new tests are required after a reconfiguration.

The following possible reconfiguration scenarios are investigated:

- The context or the requirements of the robot changed: In this case the related models are changed and it is identified (1) which previously generated test data are invalid now, and (2) which part of the new context model is not covered by the existing tests. Figure 2 presents an example: part of the context model is removed (e.g., the robot will be used in a different context), thus one of the previous tests is obsolete.

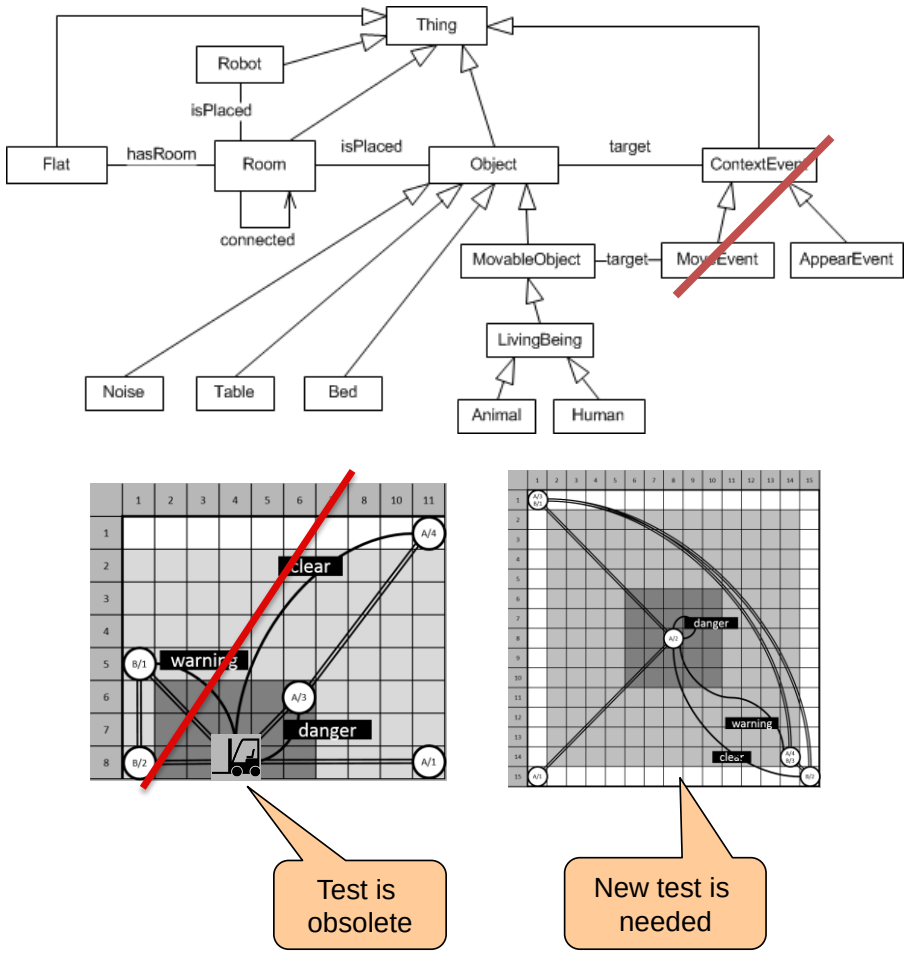


Figure 2. Change in tests due to reconfiguration in the context or environment

- The configuration of the robot changes: For example, a new type of sensor or navigation method is added to the robot. In this case, if there is a mapping between the tests and the components/skills exercised by these tests (e.g., when during a given test the robot uses a laser sensor then a mapping between the test and the laser sensor is recorded), then based on the description of the configuration the tests can also be classified. (Note that the description of the configuration could be obtained from the skill composer tool of WP 3.5.) Figure 3 presents a simplified example.

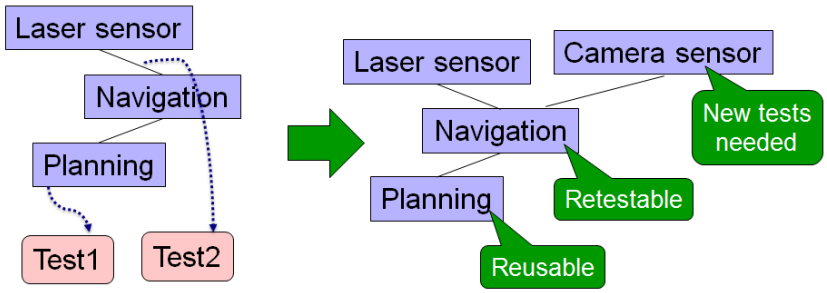


Figure 3. Effects of change in the robot configuration

2.1.3 Inputs and Outputs

In order to perform this kind of analysis in a demonstrator, the following inputs are needed:

- Description of the demonstrator's components: list of the major components and the dependencies between them (e.g., details of the architecture or the skill models developed in the project).
- Description of tests: mapping of existing tests to components (e.g., an integration test suite checking the communication between the sensors and the navigation module).
- Description of context: description of the environment of the robot used in testing.

The outcome is a method for

- the identification of the tests that need to be executed after a reconfiguration,
- the identification of those parts of the system which are not covered by the existing tests.

2.2 On-line Verification of Autonomous Robots

On-line verification by 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 etc.). Such kind of error detection is especially important in safety-critical systems, where one of the basic principles for assuring safe behaviour is reactive fail-safety: proper detection and handling of hazardous errors that occur in system components implementing a safety-related function. This principle appears, among others, in IEC 61508 (the generic standard for safety-related electronic systems).

Accordingly, on-line verification uses runtime monitor components that observe the behaviour of the robot components (the trace of states, events, actions, and the perceived context), detect the hazardous situations, and trigger a reaction to maintain safety (e.g., to stop the robot). In the typical case, these monitor components are implemented as additional software components in ROS.

In WP3.4, a technology is developed that allows automated construction of monitors by the synthesis of their source code. In our solution, the potential hazardous situations (the internal status of state variables, sequences of interactions among components, inputs and outputs) are specified using a high-level language: sequence diagrams, logic and temporal operators. On the basis of this specification, our tool automatically generates the source code of the monitor components that detect these situations.

A hierarchical structure of monitors is envisaged that is configurable in order to detect faults at the level of separate components (local monitoring) and at the level of the robot (system-level monitoring). The automated code generation offers optimized monitor code for a given configuration. In addition, instrumentation technologies are developed to support monitoring by accessing (1) local information about the states and actions to be observed by a local monitor, and (2) interactions among components to be observed by a system-level monitor.

Of course, the resource (CPU + memory) needs of monitoring depend on the number of variables/events to be monitored and the complexity of the situation to be detected by the monitor. Implementing efficient algorithms for detection and measuring overhead in concrete demonstrators is an important task in the project.

Note that monitors are useful not only in runtime (to detect operational faults), but also during testing: The monitors form part of the test oracle that decides whether the behaviour is acceptable considering the execution of a given test suite (i.e., a set of test traces is evaluated).

The following use cases of monitoring and runtime verification are supported:

- *Behaviour monitoring of software components*: In this use case the goal is to check the internal behaviour of the component, detecting in this way all errors that influence the states (state variables) and the control flow of the component. The monitored component is instrumented in order to send to the monitor information (signatures) that allow the identification of the internal states. The monitor receives this run-time information and compares it with the reference behaviour that defines the states and state transitions that are allowed (i.e., accepted by the monitor without detecting an error).
- *Trace-based monitoring of single components*: In this use case the goal is to check the externally observable behaviour of a component (when the instrumentation necessary for checking its internal behaviour is not possible). The monitor observes the timed sequence of inputs and outputs on the interface of the component (together with context and configuration related information) and decides whether the run-time sequence of these events is conformant with the reference information that is given as a set of allowed traces. Basically, reference traces capture safety properties ("something bad never happens") and liveness properties ("something good will eventually happen") restricted to the externally observable operation of a single component.
- *Trace-based monitoring of interacting components*: In this use case the goal is to check the interactions between components. The monitor observes the sequences of inputs and outputs on the interfaces of multiple interacting components (together with context related information) and decides whether the run-time sequence of these events is conformant with the reference information that is given as a set of allowed traces. Basically, traces represent here the correct execution of interactions (protocols) among multiple components.
- *Function contract monitoring*: In this use case the goal is to check contracts (simple conditions) expressed using the input and output parameters (variables) of the function interface of a component when a function call is performed. The monitor observes these parameters and checks the conditions in the form of executable assertions. In this case the scope of checking is a single function call, and the temporal ordering of the related states and events are checked separately (as presented in the previous use cases) or are not relevant.

In the following, two typical applications of these monitoring approaches are detailed (as examples), then the supporting technology is summarized.

2.2.1 Checking Safety and Liveness Properties

In this case the monitors observe the execution trace and evaluate the reachability of specified situations. Reachability is described using operators like "eventually", "always", "until", "potentially always", and "leads to", potentially with time information. An example reachability property that can be checked by a monitor is the following: "Whenever a state Stop is reached, it implies that no Speedup actions are executed until the event Restart is received". The source code of the monitors is generated in such a way that they observe and evaluate the execution trace on the basis of these expressions.

2.2.2 Checking Properties Formalized Using Scenarios

Scenarios are used to specify properties of interactions among components. An example scenario that can be checked by a monitor is the following: “If component B sends a message Start to component A, and C sends Resume to A (in any order), then B must receive Ready from C within 1 to 3 time units”. A scenario can be formalized by a Live Sequence Chart. It is composed of two parts (Figure 4): a so-called pre-chart (condition part) and a main chart (checked part). If the behaviour described by the pre-chart is encountered, then the behaviour described by the main chart must be matched to satisfy the scenario (otherwise an error is detected). The source code of the monitor is generated using an algorithm that matches the conditions and messages observed in runtime with the traces allowed by the scenario. Here the trace includes the local conditions of the components as well as the messages among them.

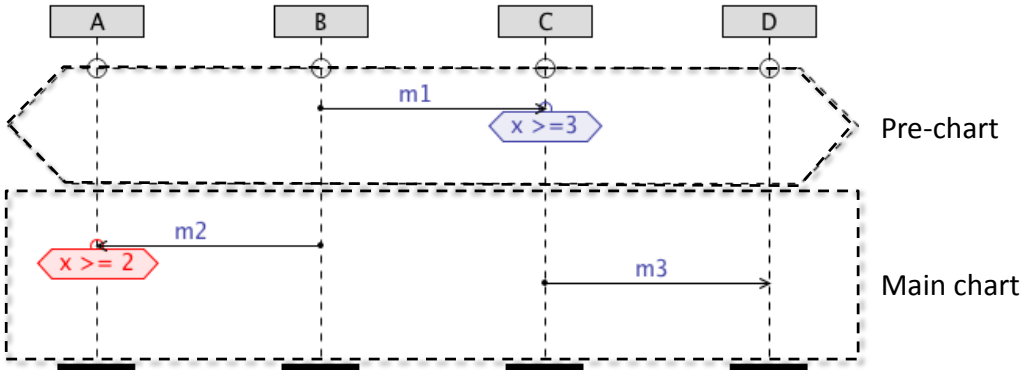


Figure 4. Live Sequence Chart with components A, B, C, D; messages m1, m2, m3; and conditions for variable x

2.2.3 Supporting Technology

On-line monitoring and verification is supported by the automated generation of the source code of the monitor components on the basis of the properties to be monitored. To do this, it is necessary to define the languages that are used to describe the properties relevant for on-line verification. Moreover, it is necessary to develop the algorithms to be used by the monitors to evaluate the properties. These algorithms will be realized by the monitor source code generator tool (Figure 5) that (together with the component instrumentation technologies) are made available to the developers.

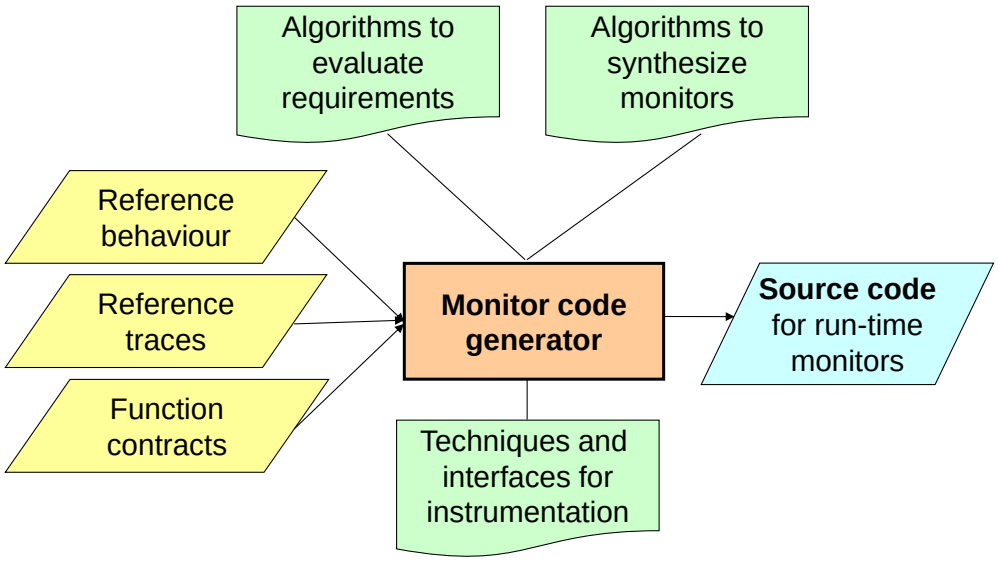


Figure 5. Supporting on-line verification by monitor code generation

2.2.4 Inputs and Outputs

In order to apply this kind of on-line verification in a demonstrator, the following inputs are needed:

- Description of properties (hazardous situations) to be monitored. These will be formalized and form the basis of monitor code generation.
- Interfaces to observe the trace of states/events/actions to be monitored, or source code of the components for instrumentation.

The outcome can be used in the demonstrator in the following form:

- The source code of monitor components (ROS components) that are able to detect the hazardous situation.
- Tools for specifying the hazardous situations and generate the source code of the monitors.

2.3 Overview of the Required Descriptions and Languages

On the basis of the use cases and general context presented in the previous sections, it can be concluded that several types of artefacts shall be captured to form the inputs for test analysis for incremental testing and for generating the source code of on-line monitors. These (types of) artefacts can be grouped into three categories as follows:

1. The so-called *common artefacts* that have to be captured both for test analysis and monitor code generation:
    - *Context elements*: As the verification of context-aware autonomous behaviour is addressed, the context elements include environment objects, their properties, the potential relations among them (e.g., abstract relations as “close to”, “lying on”), and the constraints among them (including physical constraints as well as domain-specific logic constraints).
    - *Scenarios*: Testing and on-line verification address safety and robustness of behaviour, expressed in the form of expected reactions of the autonomous system to its context and to the input events (messages and signals). Accordingly, the requirements are given in a generic and lightweight form as scenarios that include context, input events and observable output actions.
    - *Configurations*: As re-configuration is a key concept both for incremental testing and on-line verification, the (current) configuration of the system and its changes shall be captured. Configuration can be considered as a hierarchical structure of skills (from which an application is built), software components (that realize one or more skill), and hardware devices (that are used by the software components using specific interfaces).
2. The specific artefacts that are relevant for incremental testing:
    - *Tests*: Tests are captured as basic entities.
    - *Testables*: The term testable is a common artefact that includes everything that can be addressed (covered) by a test: Context fragments (relevant subset of context elements), requirements, configuration fragments (relevant subset of configuration elements), source code snippets, etc. are represented under this common term.
    - *Mapping*: The mapping is a relation “tests” among tests and testables.
3. The specific artefacts that are relevant for monitor code generation include the properties that are checked by the monitor:

- *Reference behaviour*: The reference behaviour of a monitored component gives the states and state transitions that are allowed (i.e., accepted by the monitor without detecting an error). The reference behaviour is needed in a behavioural monitoring use case, when the monitored component is instrumented in such a way that its internal states can be identified by the monitor.
- *Reference traces*: The set of reference traces gives the sequences of observable events (inputs, outputs of components) that are allowed. Reference traces are needed in the trace based monitoring use cases (trace based monitoring of single or multiple components) when the monitor is able to observe the inputs and outputs on the external interface(s) of the monitored component(s). The reference traces are usually given in a declarative form by specifying the temporal ordering among the relevant events (e.g., by using scenarios, or temporal logic operators like “after”, “before”, “until” etc.). Basically, reference traces capture safety properties (“something bad never happens”) and liveness properties (“something good will eventually happen”).
- *Function contracts*: Function contracts are simple conditions expressed using the input and output parameters (variables) of the function interface of a component. These are needed in a function contract monitoring use case, when the monitor evaluates these conditions in the form of executable assertions.

The following sections present the languages that were selected and adapted to describe the above mentioned three categories of artefacts.

3 Context, Scenario and Configuration Modelling

Context and scenario modelling formed the basis of the model based robustness testing approach that was worked out in the frame of the R3-COP (Resilient Reasoning Robotic Cooperating Systems) project and documented in its deliverables D4.2.1 “Models, Languages and Coverage Criteria for Behaviour Testing of Individual Autonomous Systems – Part I: Behaviour Testing” [69] and D4.2.2 “Behaviour Testing Strategies and Test Case Generation – Part I: Behaviour Testing” [70]. Accordingly, in the following subsections we recapitulate the context modelling (Section 3.1) and scenario modelling (Section 3.2) approaches that are reused in R5-COP in our task. Finally, in Section 3.3 we present the basic concepts of configuration modelling, that will be integrated with the output of the R5-COP work package WP13 “Dealing with Configurability”.

3.1 Context Modelling

To capture and process the information and requirements about the context (environment) of an autonomous system (AS) a model of the context can be used. The *context model* is a domain-specific description of the objects and events in the context that are relevant to the behaviour of the AS. Context modelling requires a formalized modelling language that supports precise representation and automated processing.

Based on the different aspects of the context, the model can be divided into two parts:

- Static environment description: The static part of the context model supports the representation of the environment *objects,* their *attributes* and *relations* (links). The objects are modelled using a type hierarchy (in other words, a dictionary-based taxonomy). Static elements of the context model can represent a snapshot (scene) of the environment at a concrete point of time.

  For example, in case of a household robot, the static context description should contain object types like room, furniture, table, chair, and their attributes like position, colour and size.

- Dynamic changes: To be able to represent an evolving context, dynamic changes should also be represented. This dynamic aspect is included in the context model by defining the concept of *changes* with regard to objects (i.e., an object appears, disappears), their properties (e.g., a property changes) and their relations. Changes as perceived by the AS are represented by individual context model elements called context *events* that have attributes and relations to static objects and their relations (depending on the type of the event). The actions of the AS are represented in a separate action model.

  For example, in case of a household robot, context events are moving of an object, or sounding of an unusual noise, while actions are starting the motor or speaking an alert.

The types of these model elements are captured in a type hierarchy in form of a *context metamodel*. Note that the metamodel can be systematically constructed on the basis of domain ontologies (like the existing RoboEarth or KnowRob ontology) [69]. A context model is formed by the instantiations of the context metamodel elements. In summary, a context model is able to capture (i) the relevant environment of the AS, (ii) its evolution in time, and (iii) the events perceived and the actions initiated by the AS.

In our approach, the construction of the context metamodel (either in a direct form or by constructing a domain ontology and mapping from it to a metamodel) is the task of the developer. To ease the programmatic manipulation of the context metamodel and context models,

the EMF (Eclipse Modelling Framework)[1] technology can be used for a concrete representation.

The metamodels are completed with domain-specific constraints that define various restrictions. Two types of constraints can be distinguished:

- *Well-formedness constraints* define constraints that must be satisfied by any context model, otherwise conceptual rules or the laws of physics are violated. For example, constraints expressing that “an object cannot hover” or “the sum weight of the objects placed on a table should be less than the carrying capacity of the table” are well-formedness constraints.
- *Semantics constraints* are derived from the requirements of an application, this way these are only preconditions or expectations about the context that can be violated in particular cases (e.g., when the robustness of an autonomous system is checked). For example, constraints expressing that “the chairs are around the table” or “there are two doors in the room” are semantic constraints.

The constraints can be captured using OCL, the standard Object Constraint Language. Several constraints can be mapped to context model patterns (however, there are limits to OCL constraints and this mapping, e.g., the lack of temporal operators and no quantification over infinite domains can be mentioned).

The structured context metamodel and the related constraints offer several advantages over an ad-hoc representation of context elements. Among others, it supports the automated generation of test data to test the behaviour of an AS in different contexts. The domain-specific constraints allow the automated generation of extreme contexts (as test data for robustness testing) that violate these constraints. Moreover, it is also the basis of the definition of precise test coverage metrics.

An example context metamodel and a context model of a household robot are presented in Figure 6 and Figure 7.

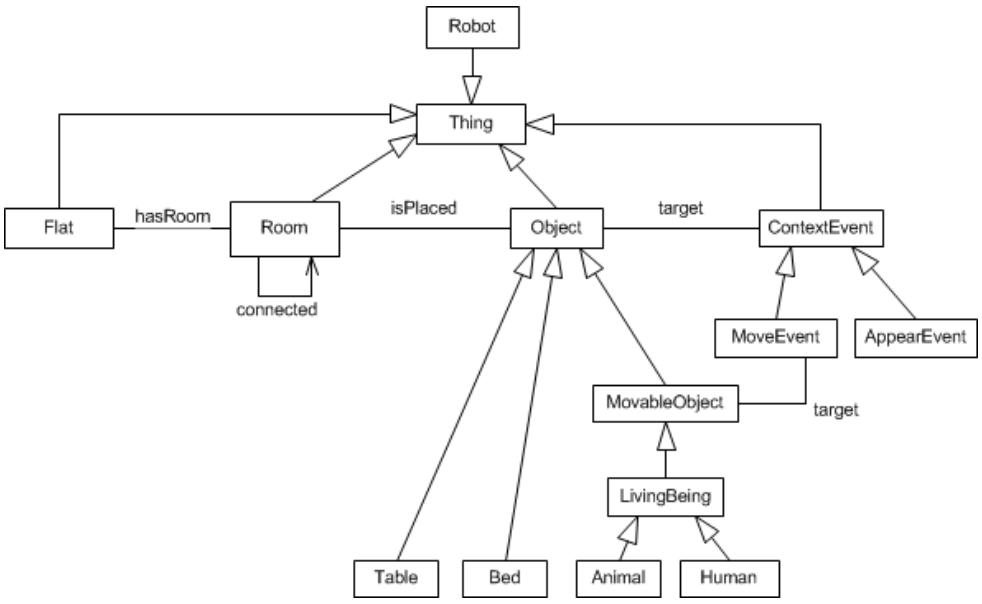


Figure 6. Context metamodel of a home environment

[1] http://www.eclipse.org/modeling/emf/

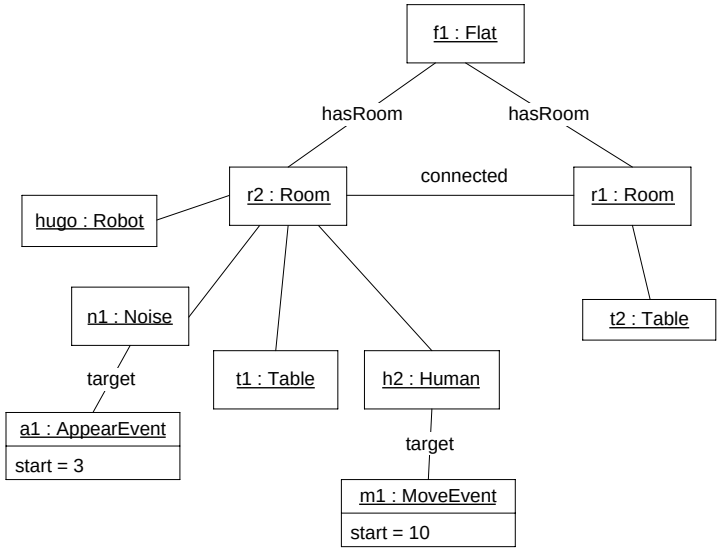


Figure 7. Context model (instance model) conforming to the context metamodel

3.2 Scenario Modelling

Scenarios are used to express requirements, which are later checked against execution traces. In the domain of autonomous systems, the requirements usually refer not only to events, actions and messages processed or initiated by the AS, but also to objects in the context (environment) of it (e.g., “if the robot is near to a human then it has to send an alert message”).

Each scenario has two parts:

- The *trigger part* captures the sequence of events and contexts that are considered as the “condition” part of a requirement, i.e., the scenario is only relevant if its trigger part has been traversed.
- The *assert part* captures the sequence of events, actions and contexts that describe what shall or shall not happen after traversing the trigger part.

These parts may have several fragments, like *opt* fragment for expressing optional behaviour, *alt* for expressing alternatives, and *neg* for a negative fragment that should not happen.

In the scenarios, context model fragments can be referenced as *initial context* of the scenario, *interim context* in the trigger part that should occur during execution, and *final context* (in the assert part) that shall be reached (matched) in order to satisfy the requirement.

Accordingly, the scenario language has two parts: a context view and a message view.

- Context view: The context view contains zero or more context fragments. A context fragment contains an instance model of the context metamodel, i.e., an arbitrary number of class instances and links between them. Instances of those classes that represent dynamic events can only appear in those context fragments that are used as initial context fragments in the requirements. A context fragment is represented as an object diagram (referenced with the name of the package containing it). The relevant part of UML’s abstract and concrete syntax (the metaclasses Package, InstanceSpecification, and Slot) was not modified. There are no other syntactic restrictions.
- Message view: Graphical scenarios are captured in the form of extended and restricted UML 2 Sequence Diagrams. Language extensions were added to refer to the context fragments. Restrictions were applied to simplify the language by eliminating

complex constructs that make the evaluation of execution traces too difficult. Figure 8 presents the subset of the model elements that can be used to represent requirements. Some of the elements of the original UML metamodel were removed in order to simplify the checking of requirements.

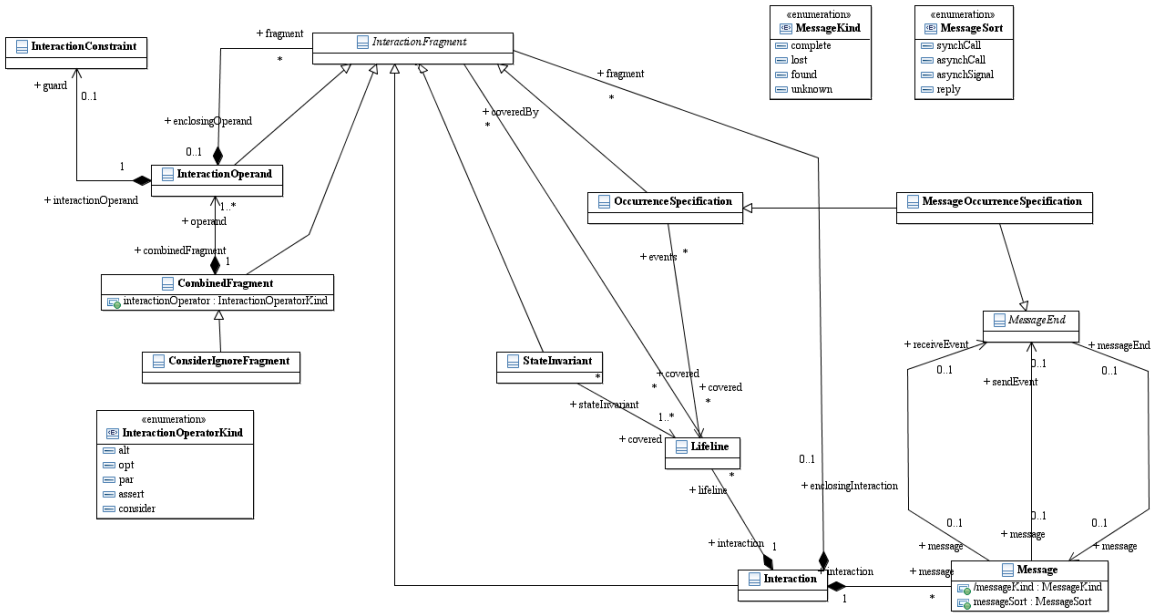


Figure 8. Reduced abstract syntax of the message view

In summary, the most important changes with respect to the original metamodel are the following:

- Gate and InteractionUse were removed to prevent references to other scenarios.
- The following operators were removed: seq, strict, loop, ignore, break, critical**.**
- Each scenario should have an *assert* fragment at the bottom of the diagram covering all the lifelines. This represents the mandatory part of the requirement, while the elements before the assert fragment form the trigger part of the requirement.

References to context fragments (used in initial, interim or final contexts) are expressed as StateInvariants placed on the Lifeline.

The concrete syntax of the language remains the same as the original, no new graphical notation was needed. In this way, existing UML modelling tools can be used to create scenarios.

The main purpose of the requirement scenarios is to evaluate execution traces (totally ordered sequence of contexts, events, and actions), and categorize them with respect to a given requirement scenario as *passed*, *failed* or *inconclusive*. The semantics was designed in a way that supports this overall goal: the operational semantics is defined by building one global *finite automaton* for the whole scenario. This automaton can be used as an *observer automaton* to categorize execution traces. The details of constructing this automaton for online verification purposes are described in Section 5.7.

Figure 9 and Figure 10 present two scenarios (R2 and R3) of a vacuum cleaner robot (SUT). R2 states that when a living being is detected nearby the robot then it has to be alerted. R3 states that if a noise is detected in the room then the robot should send a predefined alert. Here the requirements with respect to the initial contexts are described using context fragments (model instances conforming to the context metamodel).

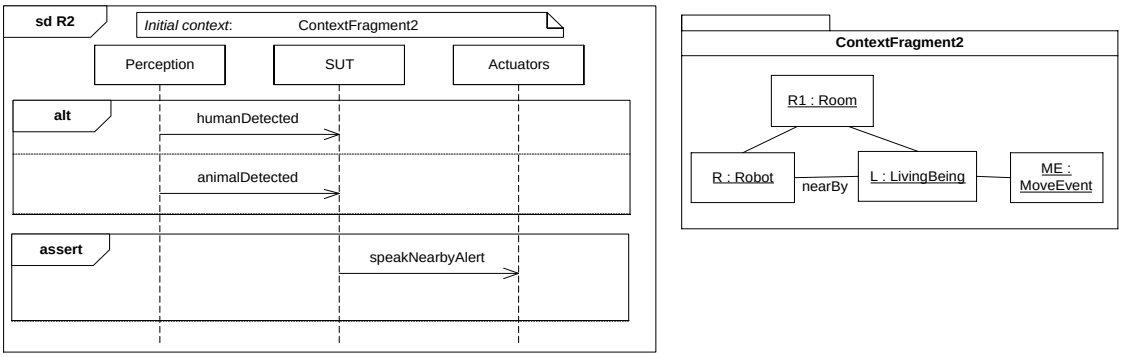


Figure 9. Example scenario model R2: Alerting a living being

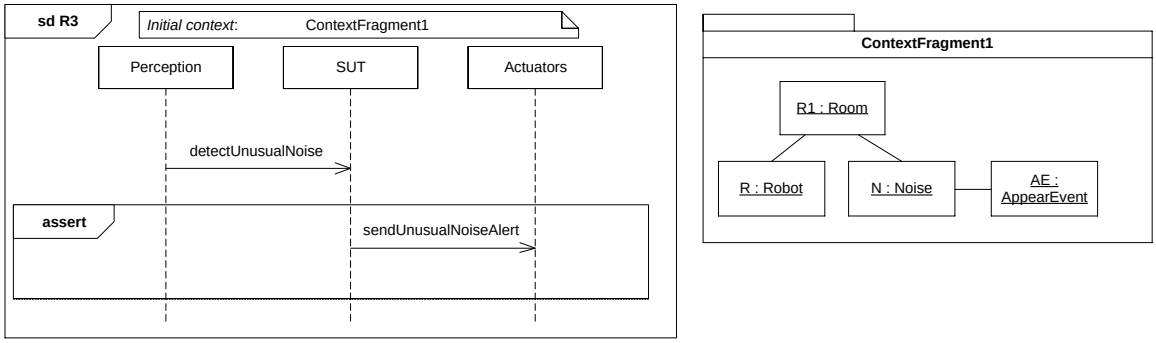


Figure 10. Example scenario model R3: Detecting unusual noise

3.3 Configuration Modelling

Modelling configurations is a complex task due to the large number of possible software packages, hardware elements and their combinations. The modelling approach is aimed to support the test generation by defining the artefacts, this yields additional requirements against the developed language.

In this section we introduce a compact language which can serve as the basis for a more complex configuration description language. Our goal is now to collect the basic concepts that are necessary for the purposes of on-line and off-line verification. This conceptual model will be merged and harmonized with the output of work package WP13 “Dealing with Configurability” in deliverables “Skill Model” and “Configuration models”.

The considerations behind the proposed language are the following:

- Simple skill modelling;
- Being able to define software and hardware components and their relations;
- Straightforward definition of complex building blocks of robots;
- Being able to define not only type but also cardinality constraints.

To support (incremental) testing of robot systems requires the ability to define the different skills of the robots and also the configurations implementing the skills.

The metamodel depicted on Figure 11 defines the basic concepts and their relations. The elements are organized into *Invertorie*s. From a high level point of view, two main concepts are modelled, *Skill*s and *Implementations*. Skills can be organized according to their dependencies (*depends* navigation). This way we provide a simple language to define the various

skills and their relations. The Implementation can either be a *ComplexComponent*, or a *ComponentType*. A ComplexComponent is composed of ComponentTypes through the *Composition* class. In Composition the cardinality of the relation, i.e., the required number of hardware and software elements can be defined. A ComponentType can be *HardwareCategory*, *SoftwareCategory, HardwareType* or *SoftwareType*. Categories can be organized into hierarchies to provide more accurate classification of the components. Software and hardware types are members of the categories: they constitute complex components together.

The metamodel was developed using the Eclipse Modeling Framework (EMF). Moreover, on the basis of the EMF technology, a simple model editor was implemented to support the quick development of configuration models (Figure 13).

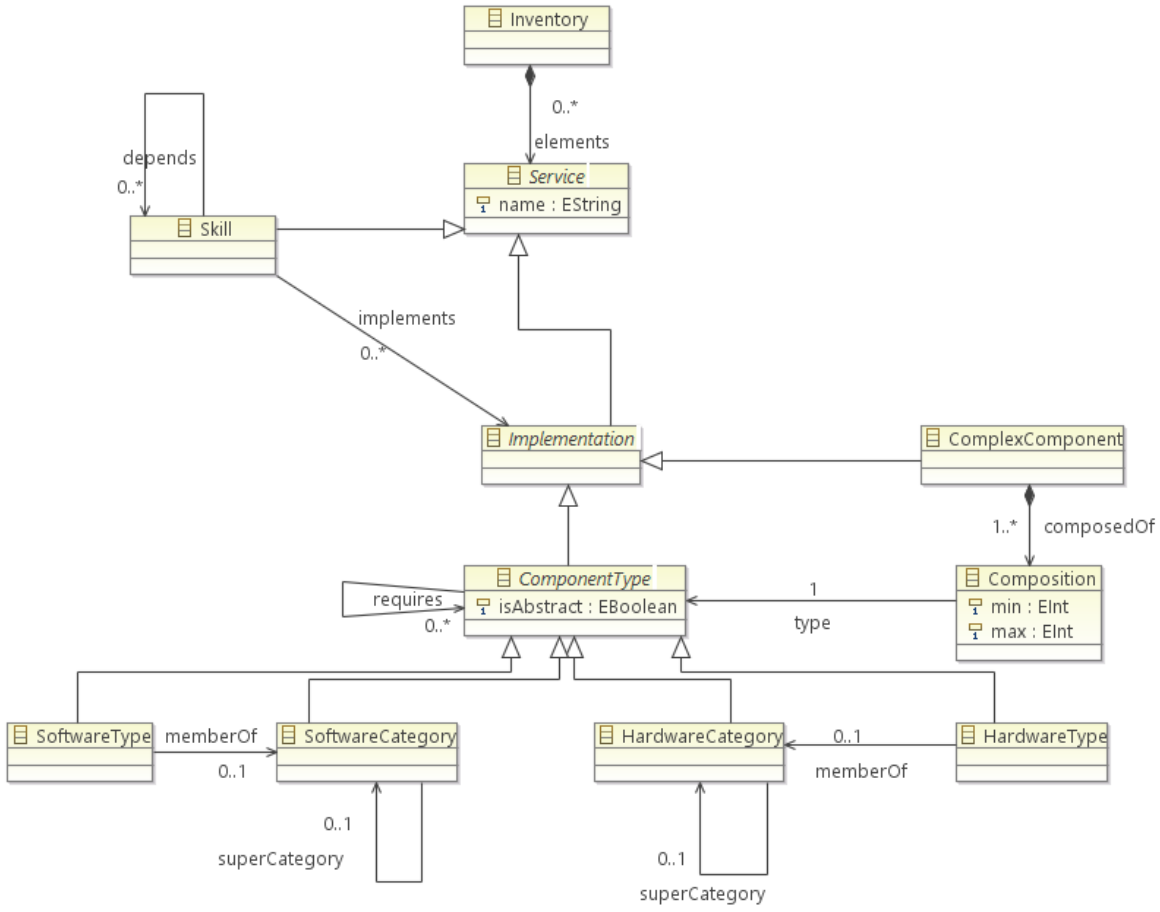


Figure 11. Basic concepts in the configuration and skill metamodel

To demonstrate the capabilities of the language, the configuration model of a hypothetic robot (inspired by Care-O-bot 3) is presented in Figure 12. Components of the robot were modelled by ComplexComponents built from various software and hardware parts. The specification of robot components means either defining categories of hardware or software parts, or giving a specific hardware or software type. For example, the implementation of the *Safe Navigation Skill* requires *laser scanners* (Hardware Category) and *Robot Navigation Software* (Software Type).

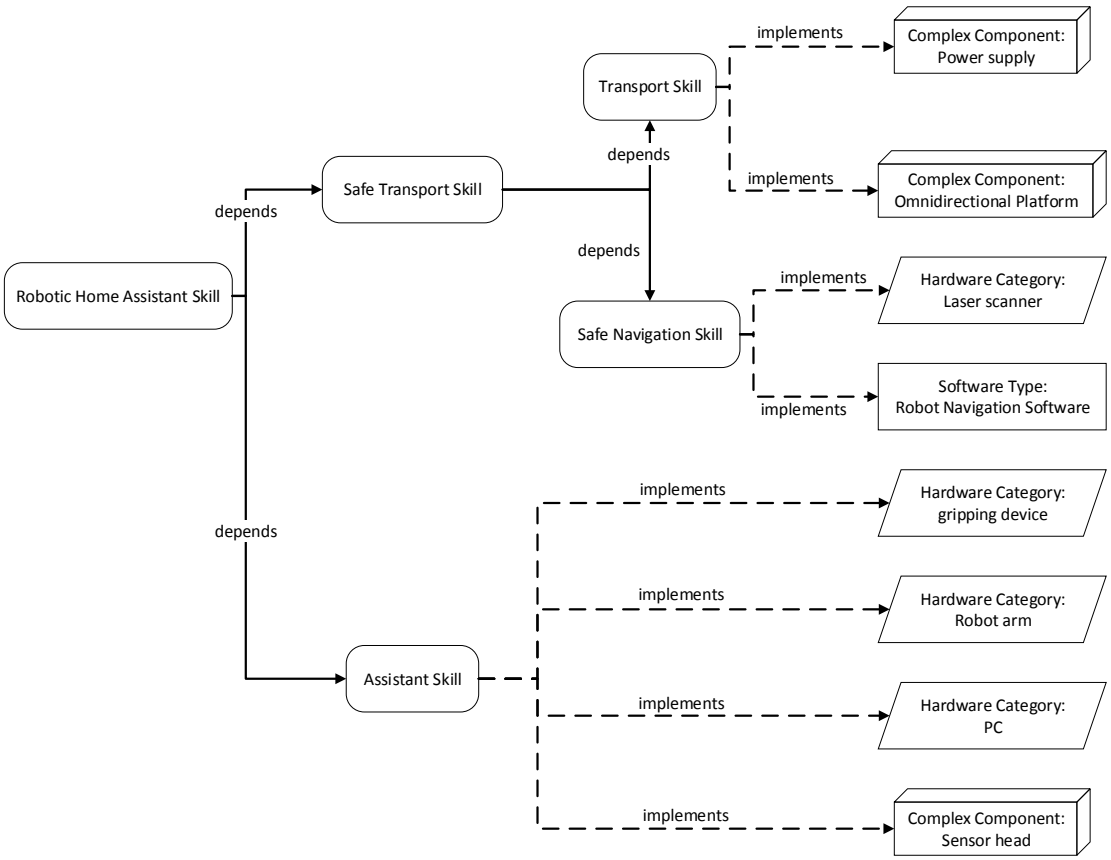


Figure 12. An example skill model

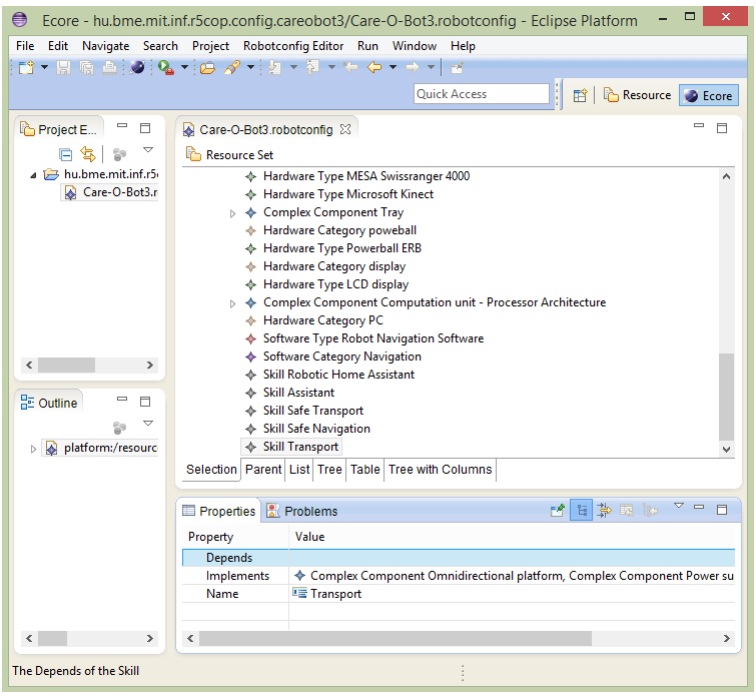


Figure 13. A simple configuration model editor

4 Describing Artefacts for Incremental Testing

4.1 Background[2]

The kind of incremental testing approach planned in the project is usually referred as regression testing in the literature [1]. Regression testing is defined as “selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or the component still complies with its specified requirements” [3]. Regression testing can be performed on any testing level (i.e., module, integration, etc.), and it can cover both functional and non-functional requirements. However, rerunning every test after each of the modifications is resource and time-consuming, thus a trade-off must be made between the confidence gained from regression testing and resources used for it. For this reason, several regression testing techniques were proposed over the years, e.g. to select only a subset of the regression test suite, what is relevant for the current change, or to identify those new parts of the system, which are not covered by existing tests.

The research in the field of regression testing focused on the following problems:

a) *Regression test selection*: select only tests from the regression test suite that are affected by changes.

b) *Test suite minimization*: find a minimal subset of test cases that preserves the coverage with respect to a certain criterion of the original test set.

c) *Coverage identification*: identify those parts of the system that need additional tests due to the change.

d) *Test prioritization*: optimize the order of tests according to some criteria, e.g. to run those tests first which are more likely to uncover bugs or which need less time to run.

e) *Test suite execution*: automatically execute the test in an efficient way.

For regression test selection techniques the basic idea is similar to the one used in build systems (e.g. the make tool), namely that at each build only those files need to be recompiled that have been changed or depend on a file that have been changed. Similarly, to reduce the size of the regression test suite, and thus reduce the time and resources needed to execute it, one can select only those tests that work on changed parts of the system. Rothermel and Harrold published a detailed survey paper about regression selection techniques [4]. They evaluated several techniques according to their inclusiveness, precision, efficiency and generality. The surveyed techniques consisted of linear equation, symbolic execution, path analysis, dataflow, program dependence graph, system dependence graph, modification based, cluster identification, slicing, graph walk techniques, etc. Each technique had its strength or weakness; some were able to uncover more errors, while some computed the selected tests very fast. A more recent survey can be found in [2].

As the test suite grows and changes, some tests become redundant. Test suite minimization techniques remove test cases from the tests suite to retain only a minimal number of test cases, while providing the same level of coverage than the original test suite. However, care must be taken, because removing too much test cases can reduce its fault detection effectiveness.

Changes in the system can introduce new parts, which are not exercised by existing tests. Coverage identification can map these parts of the system. Simple approaches can use code coverage analysis tools to uncover changed portions not touched by existing tests. More

[2] Adapted from previous work: Zoltan Micskei (BME). „Evaluation of existing methods and principles in MBT”, BME, 2009.

advanced approaches typically use some sophisticated data structure, e.g. dependence graphs that capture also data and control dependencies in the system.

Test prioritization techniques can have several goals. One can optimize the order of the test suite to increase the rate of fault detection, code coverage, or the rate at which high-risk faults are detected. Rothermel et al. analysed in [5] nine test prioritization techniques (e.g. random, prioritize in order of coverage statements, etc.). Their conclusion was that even simple approaches (which are quite easy to implement and inexpensive) can improve the rate of fault detection. However, the performance overhead of more sophisticated approaches was still a bit high.

Test suite execution techniques concentrate on the automatic execution and evaluation of test cases. These techniques moved into the practice over the years, as most of the current testing tools have these functionalities.

Running a set of regression tests is usually part of the automatic build procedures of popular, modern software development processes. However, industrial testing tools and platforms used nowadays (both commercial and open source ones) usually concentrate on just the automatic execution of tests, collection of results, and creating test reports when talking about regression testing. These tools usually do not perform test selection or minimization on the regression test suite.

On the other hand, several academic tools were reported to support research on different regression testing techniques. The drawback of these tools is however that they are usually not available to the public or not maintained any more.

4.2 The Envisaged Approach

The existing tools and approaches usually concentrated on one programming or modelling language as the input source for incremental testing. However, as we have seen in the previous sections, in R5-COP there could be multiple levels and types of reconfiguration. Instead of performing incremental testing separately for each of the change types, we could apply a unified approach, as basically they all belong to the same problem.

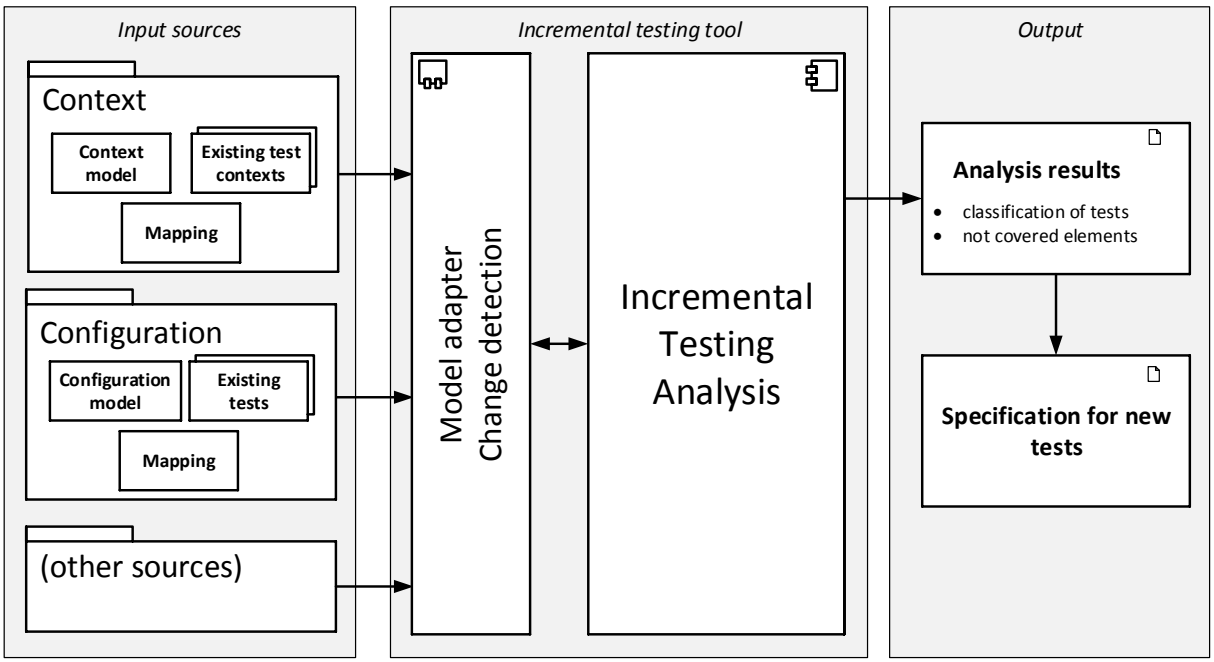


Figure 14. Overview of the incremental testing methods

We recommend to develop a *common, general incremental testing approach*, and connect the specific test types (test contexts from context models, module/integration tests for components, etc.) using special adapters to this core. Figure 14 depicts the approach in detail.

- The incremental testing analysis component is the central element of the approach. It defines a very general model for representing the tests and tested elements. The regression testing algorithms (test selection or coverage identification described in the previous section) 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 should be 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 as described in Section 2 and the coverage information of the source elements (e.g. there is a class in the context model that is not present in any of the existing test contexts). This information can later be used to create new tests either manually or automatically.

The next section will detail the required description formats for these elements.

4.3 Description for Incremental Testing

4.3.1 General Model for Incremental Testing

The core incremental testing analysis component needs a very general model to represent the inputs from the various sources. It should be able to describe elements in the context of the system or dependencies between the various components of the system.

Thus the following general model was designed that can incorporate the basic concepts of incremental testing. (The modelling diagram was created using Eclipse EMF, that is why the types are EMF-specific, like EString or ELong, but it can be easily converted to other basic types, if needed.)

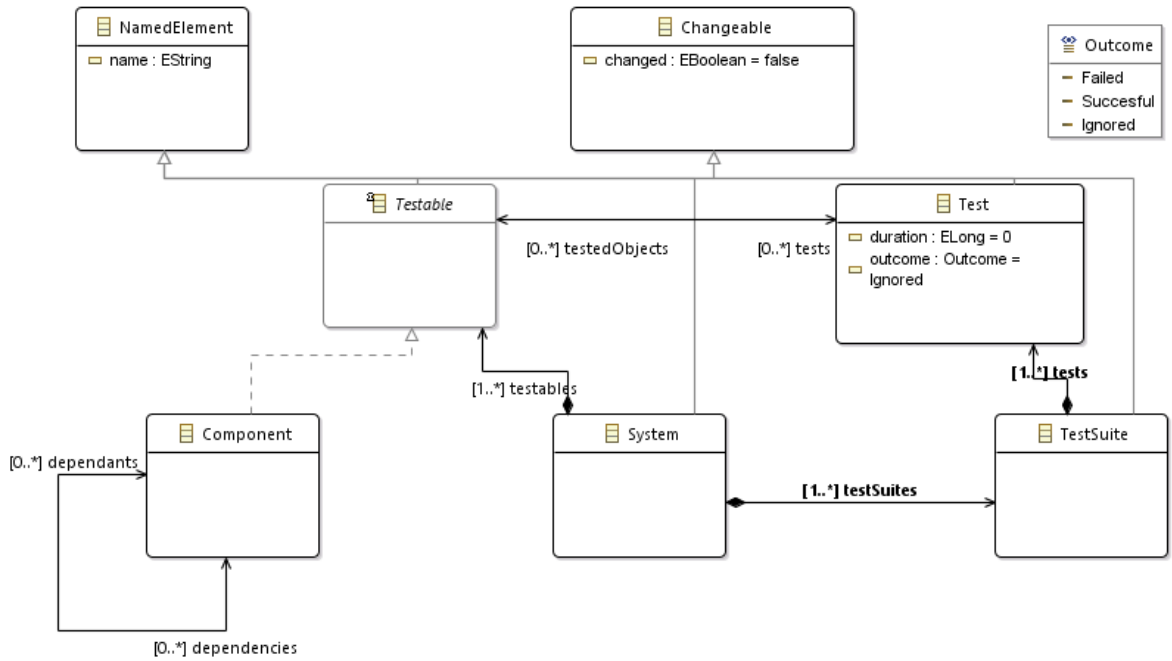


Figure 15. Metamodel for describing incremental testing

The elements in the model are defined as follows.

- **Test**: Represents a test case with the last outcome and execution time of the test. Execution time will be used for cost calculation and outcome may be used for additional algorithms (e.g. run most frequently failing tests first). Tests also have reference to the object they exercise.
- **Test Suite**: A Test Suite is used for grouping purposes, e.g. unit and integration test suites can be represented separately.
- **Testable**: Testable is an abstraction of an object, which is executed or checked by an arbitrary number of tests.
- **Component**: An implementation of Testable. It allows dependencies to be specified.
- **System**: Root element for the model, a System is composed of its testables and test suites.
- **Changeable**: abstract base class to be able to mark objects ”changed”. This flag will be used by the incremental testing algorithm, and is planned to be edited by the model adapter components.

The model could be easily extended with further specializations of these general concepts. Components, packages, context elements, source projects all belong to the Testable role, while test contexts, integration or unit tests can all be classified as tests. The fundamental thing that has to be captured for incremental testing is the hierarchy between the testables and the many-to-many relation between tests and testables.

4.3.2 Using Context Models as the Source for Incremental Testing

When the reconfiguration is performed in the environment or application domain of the robot, it can be reflected with changes in its context model. In this case, the incremental testing analysis should find those test contexts, which

- contain instances of modified context model elements,
- are invalid, because they contain instances, whose type has been deleted.

Moreover, the approach should identify not covered context model elements.

As seen in Section 3 context models are basically class models, which describe the environment of the robot under tests. The requirements for using contexts models as input sources for incremental testing are the followings.

- Context models should be specified as UML class diagrams or Eclipse Ecore models.
- Test contexts should be specified as instances of the context model.
- The mapping of tests and testable is not required to be given separately, as the instanceOf relation between an instance objects and its meta-element can be used for this purpose.

Thus, in case of context model, it is relatively straightforward to use them as inputs for the envisaged incremental testing approach.

4.3.3 Using Configuration Models as the Source for Incremental Testing

When the reconfiguration is in the capabilities or components of the robot, then it can be captured with changes in the configuration or skill model. As the skill model of R5-COP is still in development, we could not yet use directly it, but the following general requirements can be formulated.

- The configuration model (components, skills, etc.) should be given as a graph-based model, preferably a UML or Ecore model. It should describe the hierarchy and dependency relations between the configuration elements.
- The list of test projects, test suites or individual test cases should be specified.
- The mapping of tests and configuration elements needs to be specified. A relation should exist between a test and a configuration element, when
    - the test checks directly the element (e.g. a module test is written for a given component),
    - the test needs the given element for its execution (e.g. an integration test requires also the service provided by the component to start).

If the configuration model is given as a UML element, then the list of tests and the mapping can also be incorporated in the model. Otherwise, the mapping can be specified in a textual format, e.g. an XML file.

4.3.4 Description of Change and Reconfiguration

So far the models describe only one given context or configuration. However, a crucial part is to include the changes induced by a reconfiguration of the systems. Thus, the concept of "change" should somehow appear in the descriptions used in incremental testing. There are three fundamental ways to achieve this.

1. *Annotate the model*: annotate the source models with tags or stereotypes describing new, changed or deleted elements.
2. *Trigger-based support*: if the modelling environment supports hooks and triggers to notify about model manipulation, then the changes can be detected in this way.
3. *Calculate diff between models:* if an old and new version of the model is given, then the difference can be calculated

The first solution could be quite cumbersome. For a simple model, annotating it by hand could be done once, but maintaining the annotations through several changes in a large model is not preferable.

The second solution could only be used, if the input sources (context or configuration models) are created in a modelling tool, and the tooling supports change detection. Such functionality exists for instance in the Eclipse-based modelling tooling.

The third option can be used without any special modelling environment support and it does not require extra effort from the user either. However, calculating differences in large graph models in not trivial.

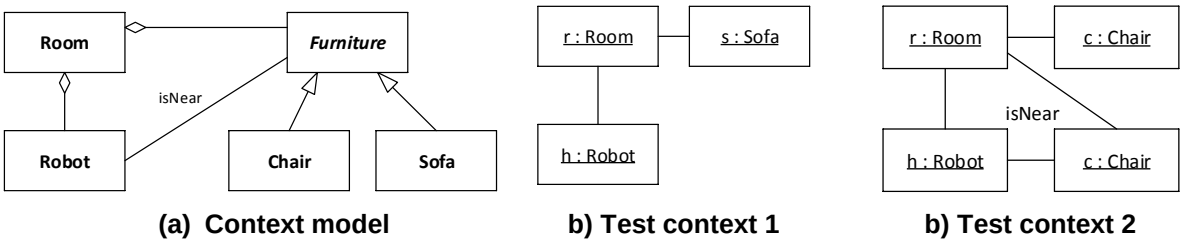
4.4 Detailed Examples Using the Defined Descriptions

The following examples present the approach for incremental testing. The examples are simplified in order to ease understanding, but the descriptions would use the same formats for larger models and demonstrators.

4.4.1 Context Models and Test Contexts

The following figure depicts the simplified context of a robot, which works in a household environment. The robot can move in rooms, which contains chairs or sofas. The robot can be near to a furniture, in that case it should avoid collision with the furniture when moving.

Two test contexts were created for this robot. In the first one there is only a sofa far apart from the robot, thus it should not watch for collision. While in the second one, there are two chairs, one of them being near, thus this situation could be more complex for the robot.



(a) Context model **b) Test context 1** **b) Test context 2**

Figure 16. Example context model and test contexts

The context model is given as a class model, and the test contexts are instance models. We plan to use the open-source, industry standard Eclipse EMF modelling environment, thus these models should be preferably in EMF or Eclipse UML2 format.

The mapping between the test contexts and the context model are defined by the instance of relations, i.e. a test contexts covers a model element, if there is an object that is the instance of that element. Thus all the non-abstract classes and the associations in the model are covered in the two test contexts.

This coverage is represented, when the input is parsed into the general model. The classes of the context model are the testable elements, and the test contexts are related to those elements, for which they contain an instance of it.

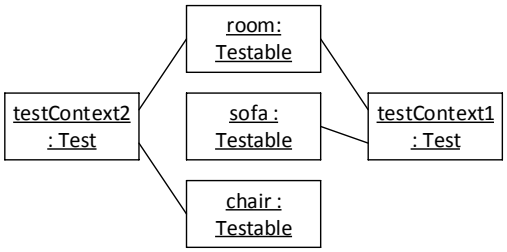


Figure 17. Representation of the context model and test contexts

Note. In this example only non-abstract classes of the model are imported, abstract classes and associations are left out. In some cases, associations could illustrate also vital requirements, thus they should also be covered. The incremental testing approach is general enough, it can be used in this situation (the important association could be represented as a Testable, and mapping of test contexts could be calculated).

Let us imagine a reconfiguration: the robot is redeployed in a different setting, for example a hospital. The context in which it should operate has changed. The new context model reflects this change:

- the attributes of the Chair model element is changed (marked with yellow colour)
- the Sofa model element is deleted (red)
- a new element, HospitalBed is added (green).

(In a real implementation instead of colours the change is captured through the activity or history of a modelling tool. Colour is used here only for illustrative purposes.)

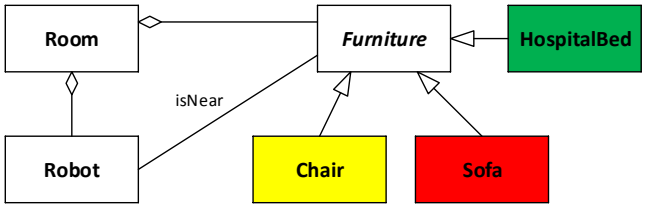


Figure 18. Context model after reconfiguration

When this changed context model is supplied to the incremental testing analysis, it detects the following:

- As the Chair testable element has been changed, all the test contexts related to this element should be retested. In the example this is test context 1.
- The Sofa testable element has been deleted, thus all the test contexts connected to this element are obsolete now. This affects test context 2.
- There is a new testable element, namely HospitalBed, which is not covered in any of the existing test contexts. This information could serve as input for generating new test contexts.

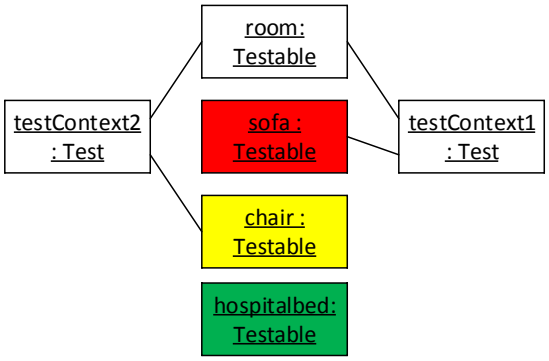


Figure 19. Incremental testing analysis of the reconfiguration

The example showed how context models and test contexts could be parsed into the general incremental testing model, and how the analysis could help testing after a reconfiguration.

4.4.2 Configuration Models and Tests

The next example shows how the incremental testing approach could be applied to configuration models. The following component hierarchy is inspired by the perception-related packages of ROS.

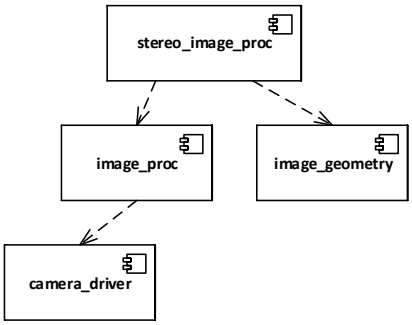


Figure 20. Example configuration model

The camera_driver handles obtaining the raw image from the camera, image_proc provides image rectification and colour processing, and stereo_image_proc performs processing the image of two cameras using the help of the image_geometry component.

The following tests are currently available for these packages:

- DirectedTest: unit test for image_geometry, which tests some basic calculation.
- ImageProcTest: unit test for image_proc, which directly loads a raw camera image stored in a file.
- StereoProcTest: an integration test, which starts the whole perception stack and processes the raw image given to the camera driver.

This information can be represented with the incremental testing model in the following way.

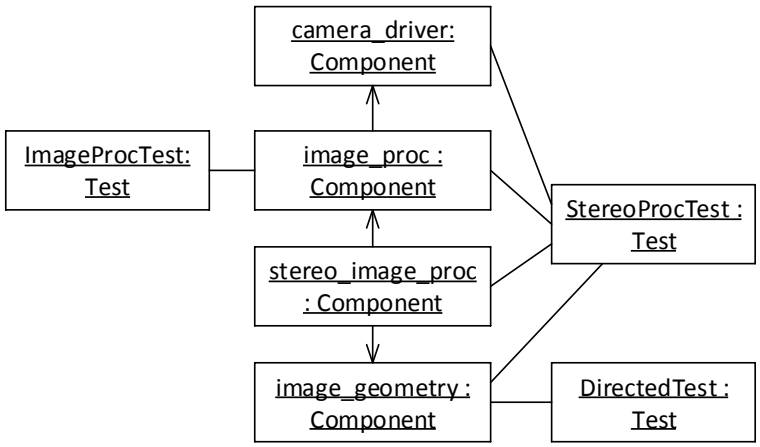


Figure 21. Representation of the configuration model and test contexts

In this model instances of Components are used (Component is an implementation of Testable), because we want to capture dependencies between the different packages.

Now let's imagine two modifications. In the first, only the image_geometry component is changed. In the second, the camera_driver is modified.

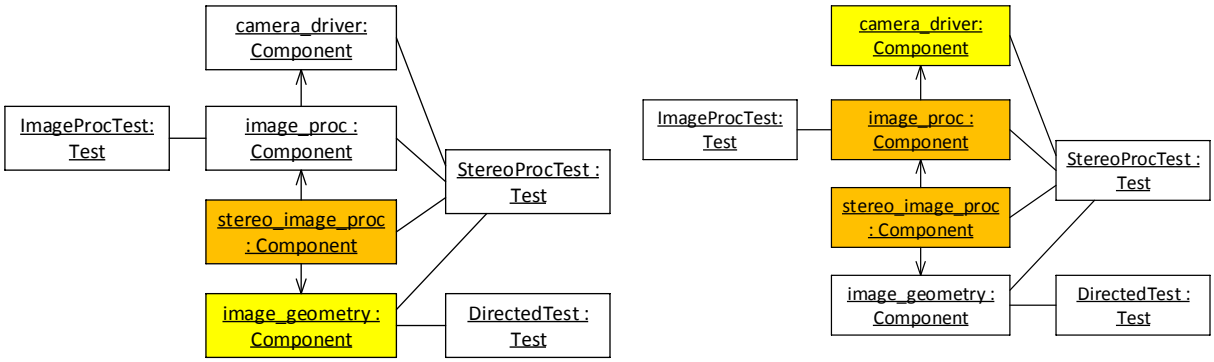


Figure 22. Effect of the two reconfigurations

In the first change the image_geometry component is modified, but because the stereo_image_proc component depends on it, it is also affected (illustrated with orange colour on the figure). Thus, the DirectedTest and StereoProcTest test should be rerun, but the ImageProcTest could be omitted. Similarly, if the camera_driver component is changed, then StereoProcTest (direct useage) and ImageProcTest (indirect usage through) should be executed.

The approach can be extended with using further attributes of tests in the selection (e.g. costs, execution time, previous outcome, etc.), and defining different strategies (e.g. selecting the minimal number of tests covering the changed elements, selecting all affected tests, etc.). These strategies and their applicability to the R5-COP project and its demonstrators will be investigated in Task 34.2 next.

5 Describing Properties for On-line Verification

This section presents the languages selected or defined for describing the properties for on-line verification. Subsection 5.1 provides an overview on the role and required basic properties of these languages; subsection 5.2 presents a categorization of existing solutions and identifies the new challenges; while the subsequent subsections describe the languages in detail.

5.1 Overview of the Languages

In Section 2.3, the main types of artefacts relevant for on-line verification were identified and categorized. The *common artefacts* are captured by the languages that are described in Section 3. Their role in monitoring can be summarized as follows:

- *Context elements*: Context modelling (see in Section 3.1) is used to represent context fragments that are included in scenarios (as initial, interim or final context), and context fragments that capture relevant parts of the perceived context in reference traces to be monitored.
- *Scenarios*: Scenario modelling (see in Section 3.2) is used to represent requirements as the sequence of contexts, input events, interactions, and output actions (in a form similar to message sequence charts). For monitoring purposes, a scenario is filtered to get the sequence of events and actions that is relevant to the component that is monitored.
- *Configurations*: Configuration modelling (see in Section 3.3) is used here to identify the software components (with their interfaces) that are monitored.

This section will describe the languages to capture the properties to be monitored. On the basis of the use cases of monitoring (see in Section 2.2) and the identified artefacts (Section 2.3), the following basic requirements towards the languages can be summarized:

- *Behaviour monitoring*: The monitored property is given as reference behaviour. The language for capturing the reference behaviour shall be able to describe the states and state transitions that are allowed (i.e., accepted by the monitor without detecting an error).
- *Trace based monitoring of single or multiple components*: The monitored property is given in the form of reference traces of observable inputs and outputs on the external interface(s) of component(s). The language for capturing the reference traces shall be able to describe the allowed temporal ordering of events, actions, and perceived contexts.
- *Function contract monitoring*: The monitored property is given in the form of function contracts as local conditions evaluated on the parameters of a function call (note that temporal ordering of calls and states are checked separately). Accordingly, the language for capturing function contracts shall be able to specify assertions expressed using the input and output parameters (variables) of the function interface of a component.

The required property description languages are used at two levels of formalization (the rationale is described in Section 5.2.3 on the basis of Section 5.2.2.11):

- *Formal languages*: For automated monitor source code generation purposes, it is required to use formal languages with precise syntax and semantics. However, mathematical languages with formal semantics are not easily used by developers (consider, for example, temporal logics or formal automata), this way it is more convenient to offer so-called engineering languages.

- *Semi-formal engineering languages*: There are languages that are regularly used as requirement description and behaviour modelling languages by software developers in analysis and design phases. These languages typically have graphical syntax (diagrams) but lack formal semantics that is necessary for automated code generation. This problem is solved by providing a mapping from the engineering language to a formal language (assigning this way a formal semantics to the language).

On the basis of these considerations, languages are offered on two hierarchical levels: engineering languages (describing checked properties given by the developers) and formal languages (used mainly for internal formal specification for monitor synthesis). These two levels and the corresponding mappings are presented in Figure 23. The properties to be checked are related to state-based event-driven behaviour. To identify the sequences of internal states, instrumentation of the component is necessary. The traces of events (like messages among components) can typically be observed on the external interfaces of the component on the basis of monitoring the inter-component communication (e.g., message channels). These alternatives are presented at the bottom of Figure 23.

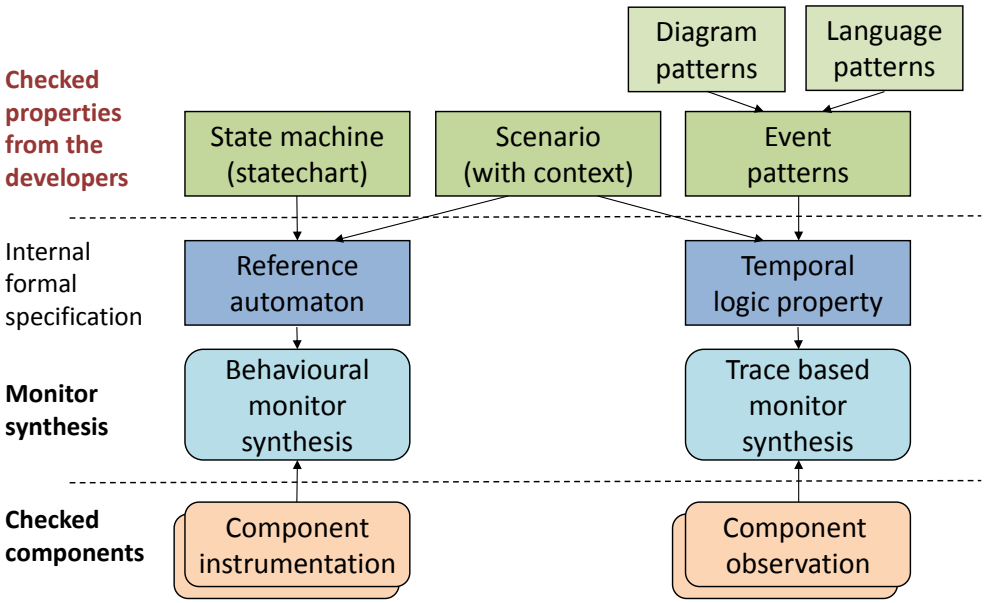


Figure 23. Languages used to describe properties to be monitored

The selection of these languages is explained in Section 5.2 where the existing approaches and general requirements for property description languages are analysed. The role and use of these languages can be summarized as follows.

Formal languages for automated generation of monitor code include automata, temporal logic languages and contract languages:

- *Automaton language*: It is used to represent reference behaviour in a formal way. The automaton is considered as an observer automaton that evaluates the sequence of states and state transitions observed by the monitor. Behaviour that differs from the reference automaton is considered as erroneous.
- *Temporal logic language*: It is used to represent reference traces. A new variant of linear temporal logics called Context-aware Timed Propositional Linear Temporal Logic (CaTL) is defined that (besides having the usual temporal operators as “next”, “until”, “eventually” and “globally”) includes observable events, actions, perceived context, configuration, and also time. This way the context-aware real-time behaviour of reconfigurable systems can be monitored.
- *Contract language*: This language is used to capture function contracts that are mapped to executable assertions.

The engineering languages are used and mapped to formal languages in the following way:

- *Statechart model*: The statechart model offers a higher level representation of reference behaviour, having convenient constructs to represent state hierarchy, concurrent regions, and composite state transitions. A restricted subset of UML 2 statechart (state machine) model is used with a formalized semantics that maps a state machine defined by the statechart to a formal automaton (represented by the automaton language).
- *Scenario model*: As the scenario model captures requirements, it has a central role in specifying the properties to be monitored. Its formal semantics is given by defining an observer automaton that decides on the run-time sequence of events, actions and contexts. On the one hand, using this formal semantics, it is straightforward to map a scenario to a formal automaton (represented by the automaton language). On the other hand, it is also possible to characterize the allowed sequences of a scenario by temporal logic operators, this way the scenario is mapped to the temporal logic language.
- *Event patterns*: The low level mathematical operators of temporal logics can be replaced by more intuitive graphical or natural language constructs. Moreover, using these intuitive construct, the typical and often used temporal properties can be defined as building blocks, this way defining a pattern library with composition rules.

After summarizing the existing solutions (Section 5.2), the next subsections will present the related languages and mappings in detail:

- The automaton language is defined in Section 5.3.
- The temporal logic language is defined in Section 5.4.
- The contract language is defined in Section 5.5.
- The statechart based modelling of reference behaviour is summarized in Section 5.6.
- The scenario model is recapitulated (from Section 3.2) in Section 5.7.
- The idea of event patterns is introduced in Section 5.8.

5.2 Existing Solutions and New Challenges

This subsection gives a concise overview on the existing typical runtime verification techniques, supporting tools, characteristics of existing property description languages, and summarizes the new challenges.

5.2.1 Runtime Verification Techniques and Supporting Tools

The implementation of runtime verification needs several techniques that can be categorized in the following way:

- *Property specification techniques*: Solutions are given for the specification of the properties to be monitored, typically based on execution history or contracts. The solutions that focus on “specificationless” common properties like atomicity of execution or freedom from race conditions can also be mentioned here.
- *Verification techniques*: Algorithms are given for the monitoring of temporal specifications, contracts, or specificationless properties.
- *Instrumentation techniques*: Solutions are given for systematic modification of the source code or object code of the components to be monitored. Low overhead is a typical requirement for instrumentation.

- *Feedback techniques*: These techniques give solutions for initiating the error handling when the run-time verification detects an error.

These techniques with typical solutions are presented in Table 1.

Technique	Solution category	Solutions
Property specification techniques	Execution history based techniques	Temporal logics
		Regular expressions
		State machines
	Contracts based techniques	Contracts
		Assume-guarantee interfaces
	Specificationless properties	Common properties
		Freedom from race conditions
		Atomicity
		Serializability
Verification techniques	Monitoring temporal specifications	Incremental[3] model checking
		Monitoring specific logics
	Design-by-contract monitoring	Contract primitives
		Comment-based frameworks
		Trace assertions
	Specificationless monitoring	Data race checking
		Atomicity and serializability checking
		Linearizability and refinement checking
Instrumentation techniques	Observation techniques	Active instrumentation
		Passive instrumentation
	Interaction techniques	Synchronous approach
		Asynchronous approach
	Implementation techniques	Aspect Oriented Programming
		Tracematch event patterns
	Overhead reduction techniques	Residual typestate analysis
		Removing instrumentation points
		Control-theoretic approaches
Feedback techniques	Alerting	Generation of alerts
	Recovery routines	Forward recovery
		Backward recovery
		Switch to fail-safe mode
	Diagnostic facilities	On-line diagnostics
		Off-line diagnostics
	Runtime enforcement	Blocking of events
		Delaying events

Table 1: Techniques and example solutions for runtime verification

[3] Here incremental model checking means that not a static model of behaviour is checked but a runtime evolution of the observed behaviour, using specific techniques that focus on the efficient handling of the incrementally observed new states and events.

The architectures of monitoring can be categorized in the following way:

- *Basic monitors* observe and check single systems or components using the same platform as the monitored component. Specific implementation techniques are in-line and out-line monitoring.
- *Distributed monitors* are used as separate monitor nodes in distributed systems in order to minimize the dependencies between the monitor and the monitored components (nodes). In case of simple bus-monitor architectures, the same data bus is used to interconnect the monitor with the observed nodes and the nodes themselves, while in case of single process monitor architectures a dedicated monitor bus is used to interconnect the monitor with the observed nodes. In distributed process monitor architectures several monitor components are applied that are interconnected by a monitor bus.

These types of architectures with references to typical examples are listed in Table 2.

Architecture	**Solution category**	**Solution examples**
Basic monitors	Off-line monitoring	[6] [7]
	On-line monitoring	General on-line monitoring [8] [9]
		In-line monitoring [10]
		Out-line monitoring [11]
Distributed monitors	Bus-monitor architectures	[12] [13]
	Single process monitor architectures	[14] [15] [16] [17] [18]
	Distributed process monitor architectures	[19] [20]

Table 2: Architectures for monitoring with examples

The existing tools and languages that support monitor synthesis and instrumentation can be categorized as follows:

- Monitoring temporal specifications: The properties to be monitored give temporal ordering of events, actions and messages, typically in a declarative form by using temporal logic, event patterns, and evaluation rules.
- Monitoring state machine specifications: The properties to be monitored refer to states and allowed transitions among them.
- Design-by-contract specifications: Conditions to be verified are given by code contracts or trace contracts.
- Instrumentation tools: The instrumentation tools support the systematic modification of the source code or object code of the components to be monitored.

Typical tools and frameworks in these categories are given in Table 3.

Tool category	**Example tools and frameworks**
Monitoring temporal specifications	MOP Framework [21]
	Eagle [22]
	RuleR [23]
	Java PathExplorer [24]
	Java-MaC [25]
	Java LTL\X monitor [26]
Monitoring state machine specifications	RMOR [27]

Tool category	Example tools and frameworks
Design-by-contract specifications	Eiffel [28]
	Spec# [29]
	Code Contracts [30]
	TraceContract [31]
	JML [32]
	Jass [33]
Instrumentation tools	J-Lo [34]
	Hawk [35]
	AspectBench [36]
	Larva [37]
	InterAspect [38]
	Clara [39]

Table 3: Typical tools and frameworks for on-line verification

5.2.2 Property Description Languages

This section gives a short overview of widely-used formal specification languages, focusing on the higher level languages that were intended to be used by engineers (instead of mathematicians). Language elements or basic ideas of these mentioned approaches will be reused or referenced in the languages selected for specifying the monitored properties. The limitations of these existing solutions and the extensions necessary for our purposes are discussed at the end of the overview in Section 5.2.3.

5.2.2.1 Statecharts

Statecharts are extended, hierarchical finite state machines first introduced by Harel in [40]. He defined his statecharts as FSMs extended by “hierarchy, concurrency and communication”. The UML State Machine formalism is a widely-used object-based variant of Harel's statecharts[4]. There are, however, some features that are not inherited by UML State Machines; also some possible extensions exist (e.g., parameterized states, overlapping states, probabilistic statecharts). The original formalism has support for delays and timeouts using implicit timers: conditions like “10 sec in state X” are allowed and also the timeout can be indicated graphically. The UML standard defines similar time representation, e.g., “after (15 sec.)” as transition trigger.

A semi-formalisation of the UML State Machines can be found in the corresponding ISO standard [41]. It specifies the syntax of the language, but the semantics are not defined precisely (there are missing pieces, for example the time semantics). There are several approaches to the formalization of the semantics. Among others, the paper [42] can be mentioned that presents the formalization of the semantics of a subset of UML Statechart Diagrams. Their approach is to map the statecharts to extended hierarchical automata and define the formal semantics this way. A more extended formalization (covering all concepts that are relevant from the point of view of the practice) is found in [62]. This formalization is used as the basis for automatic source-code level implementation of specified behaviour and runtime error detection (comparison of the runtime behaviour with the reference one provided by the State Machine model).

[4] Another variant with different semantics is the STATEMATE variant [90].

5.2.2.2 VDM-SL, VDM++, and Z

The Vienna Development Method is a formal method for specification and development [43], developed initially by IBM in the 1970s. Its main element is the VDM-SL (Vienna Development Method Specification Language) language, which is one of the two standardised formal specification notation maintained by the ISO working group of Formal Specification Languages (JTC1/SC22/WG19). VDM-SL relies on set theory. Also, functions are defined along with pre- and postconditions. It is a “wide spectrum specification language” [43], as it is suitable both for high-level, abstract specification, and for low-level, detailed specification.

VDM++ is the object-oriented version of VDM-SL, thus here the structuring is based on classes, not on modules. RAISE is an evolution of VDM [44]. Also, the COMPASS Modelling Language (CML) is an extension of VDM.

VDM-SL is not directly supporting time, but VDM++ and its real-time extension called VDM-RT cope with time [45]. Its semantics is based on a discrete clock and time costs associated with each VDM construct.

The Z Notation [46] is one of the two standardised formal specification notation maintained by the ISO working group of Formal Specification Languages (JTC1/SC22/WG19). The Z, similarly to the B Method, also relies on set theory. An attempt to use UML diagrams (class, state and object diagrams) together with Z specification is described in [47]. Here UML can be considered as a graphical interface for the Z formalism. The Z notation does not support time explicitly.

5.2.2.3 RSML

RSML [48] is a requirements specification language for embedded systems, based on Statecharts by Harel. The state and its refinement concepts are similar to UML Statecharts. The main difference is that the guard expressions are expressed using so-called AND/OR tables that is a tabular format of the disjunctive normal form of the expression. Its goal is to help the description of complex expressions.

The paper [49] provides a good overview of the method, along with a detailed example from the TCAS II airplane collision avoidance system. This paper focuses on the slicing of state machines in order to help the users in understanding the specification by focusing on a chosen scenario.

RSML^{-e} is a different version of RSML that supports rigorous specifications of the interfaces between the environment and the control software. The paper [50] introduces RSML^{-e} and presents a translation to NuSMV for model checking purposes.

SpecTRM-RL is a successor of the RSML language. An introduction is found in [51]. This language had tool support (Eclipse-based tool called SpecTRM), but apparently it is not maintained anymore. It is claimed that timing constraints may also be specified as conditions in the tables (i.e., conditions on the state transitions) in SpecTRM-RL [51].

5.2.2.4 Visual Timed Event Scenarios

The authors of Visual Timed Event Scenarios (VTS) decided to create a brand-new formal notation to express properties to be checked, instead of extending existing ones. A VTS describes an event pattern, where events and the precedence and temporal distance between them are defined. Restrictions can be also included (e.g., forbidden events). The formal syntax and semantics of VTS is defined in [52], also a translation to timed automata for verification purposes is described. An example VTS can be seen in Figure 24. Arrows denote precedence among events, and also temporal distance between events is illustrated.

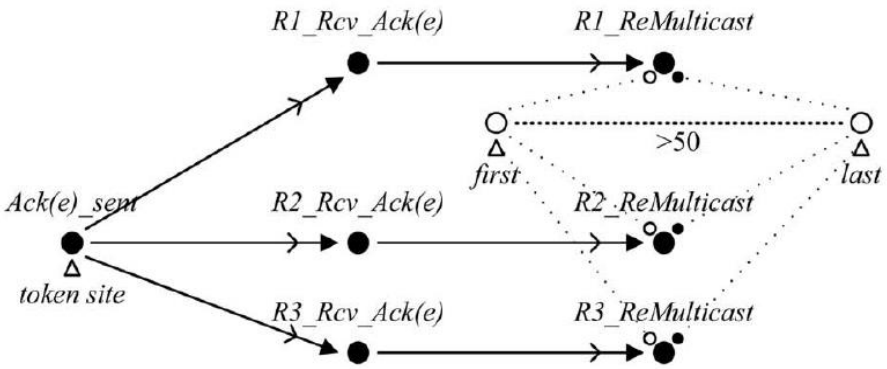


Figure 24. Example VTS specification [52]

5.2.2.5 Graphical Interval Logic

Graphical Interval Logic (GIL), presented in [53], is a formal specification notation for concurrent software systems. The aim is to provide a formalism that can be easier-to-use for the software engineers than temporal logics. The formulas are similar to informal timing diagrams. To illustrate the structure of GIL diagrams, an example can be seen in Figure 25 (the notation is not introduced here in detail).

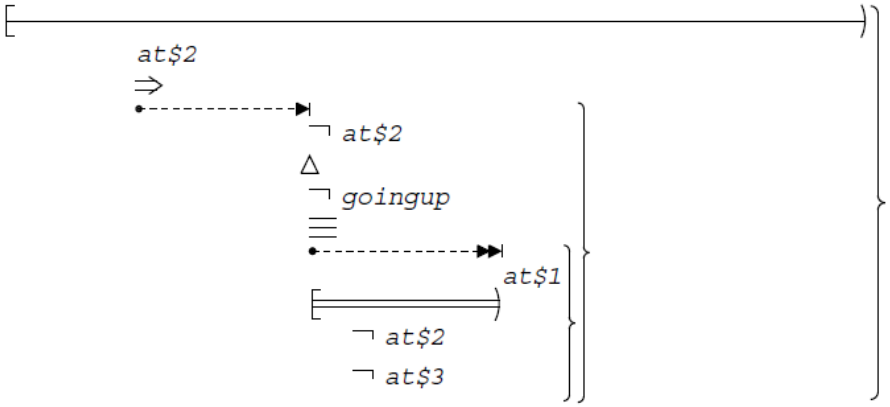


Figure 25. Example GIL specification [53]

5.2.2.6 PSL

The Property Specification Language is a general assertion language for hardware description languages [54]. PSL is typically used to describe assertions that are required to hold. PSL is composed of 4 layers: Boolean, temporal, verification, and modelling layer. The first contains Boolean expressions; the second is composed by temporal operators. The verification layer defines how the temporal expressions should be handled (if it is an assertion, an assumption, a restriction, etc.). The modelling layer is only used in complex models. PSL comes in different flavours to adapt mainly the Boolean and temporal layer to the syntax of the programming language used with.

This language is now defined in the IEEE 1850-2010 standard [55], containing the formal syntax and semantics.

5.2.2.7 Sequence Charts (MSC, LSC. PSC)

Message Sequence Charts (MSC) is a visual formalism to capture system requirements in the early design stages [56]. In UML, it is the basis for Sequence Diagrams. Although for the first sight MSCs provide an intuitive way to present the possible scenarios of a system, they have many weak points: its expressiveness is weaker than temporal logics, difficult to distinguish between possible and necessary behaviour, etc. [56]. The MSCs can be grouped into high-level MSC (HMSC) that is a finite state machine whose states are labelled by MSC showing the possible connections (asynchronous composition) between the scenarios. MSCs are widely used in the industry and are standardised in an ITU recommendation 87. An example MSC is presented in Figure 26.

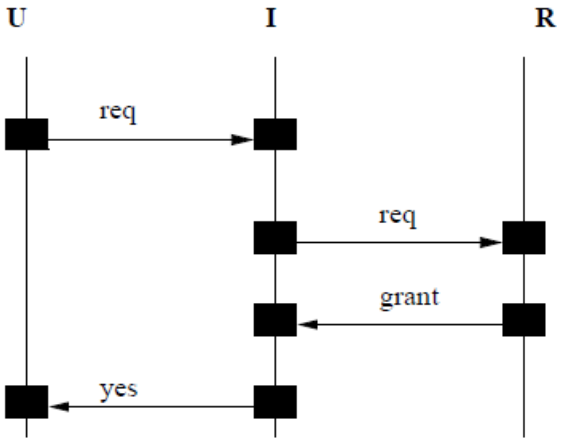


Figure 26. Example MSC [56]

The Live Sequence Charts (LSCs) introduced in [58] extends MSCs by making possible to clearly distinguish between possible, necessary and forbidden behaviour. Also, it is often misunderstood if an MSC specification is a collection of sample behaviours, or is a complete set of allowed behaviours [58]. For this purpose, hot and cold elements are explicitly distinguished, corresponding to necessary and possible behaviours, respectively. LSCs are also extended by “preconditions” (pre-charts) and looping constructs.

The work [56] also mentions briefly that timing constraints can be included in LSCs. The play-in/play-out approach is also presented: this is a LSC specification method, where the LSCs are automatically built based on the developers interaction with a skeleton.

An example LSC can be seen in Figure 27 At the top, the participants of the scenario are given with their lifelines drawn from the top to the bottom. Then pre-chart (conditions and messages that shall be observed in order to start checking this scenario) is given in a dashed fragment, while at the bottom part three embedded fragments specify the required sequences of messages and conditions.

The Property Sequence Charts (PSCs) were introduced in [59] to provide a scenario-based language for expressing temporal properties. Similarly to LSCs, the possible behaviours (“regular messages”), the mandatory behaviours (“required messages”), and the forbidden behaviours (“fail messages”) are distinguished. Also, a strict ordering operator is introduced. Furthermore, multiple constraints/restrictions can be assigned to the messages. [59] presents the syntax and the operational semantics of PSCs by mapping them to Büchi automata. The main advantage of PSC with respect to LSC is claimed to be its ability to specify chain constraints.

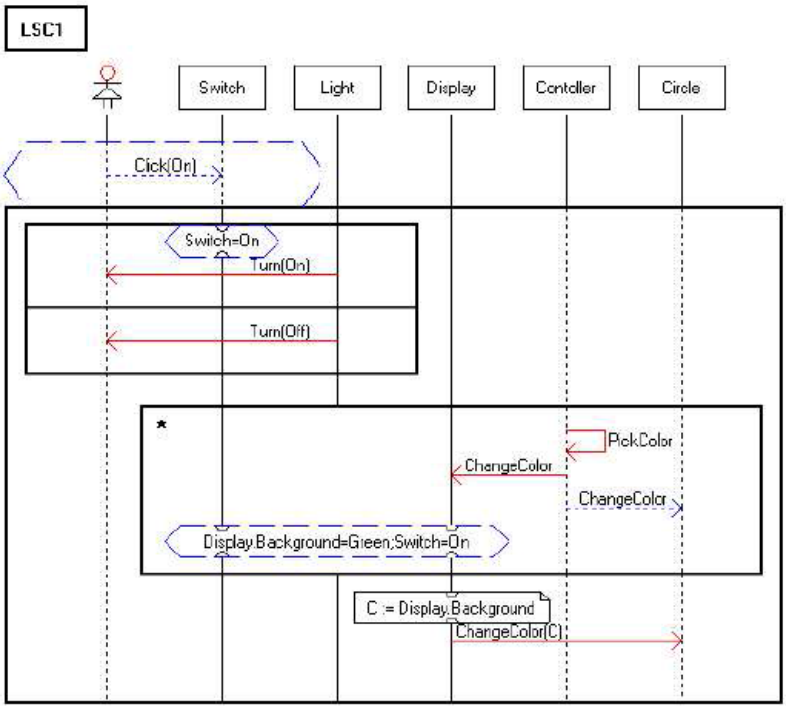


Figure 27. Example LSC [56]

5.2.2.8 Temporal OCL

Temporal OCL (TOCL) [60] is a generic model-based property language. TOCL is an extension of the Object Constraint Language (OCL), that “smoothly integrates” past and future temporal operators. The authors of [61] claim that even if TOCL is close to OCL, it is “still not well suited to many domain engineers”. An example TOCL expression can be seen in Figure 28.

```
context Program inv:
  self.mode = #initialization implies
    always self.mode = #initialization
    until (PhysicalUnit.allInstances->forAll(pu | pu.ready)
           or self.wlmdFailure)
```

Figure 28. Example TOCL specification [60]

5.2.2.9 Pattern-based Requirement Formalization

The idea of property specification patterns was first suggested by Dwyer et al [72]. The motivation was to free the user from building complex temporal logic expressions that needs deep knowledge and expertise. They proposed a specification pattern system which is a hierarchical system of simple patterns. These patterns generalize commonly occurring requirements without being too abstract. The paper [73] extends this work by assessing the method based on more than 500 real requirements, collected from literature, researchers, mailing lists and student projects. They found that 92% of these requirements were instances of their patterns. Their updated pattern collection is available online [74].

A similar work restricted to safety patterns can be read in [75]. It contains a hierarchical classification that can help the user to find the appropriate pattern. An example pattern definition (excerpt) can be seen in Figure 29.

Precedence

Intent

To describe a relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first. Also known as *Enables*.

Example Mappings

In these mappings S enables the occurrence of P.

CTL S precedes P:

Globally	$\neg E[\neg S\ \mathcal{U}(P \wedge \neg S))]$
Before R	$\neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R \wedge EF(R))]$
After Q	$\neg E[\neg Q\ \mathcal{U}(Q \wedge \neg E[\neg S\ \mathcal{U}(P \wedge \neg S)])]$
Between Q and R	$AG(Q \rightarrow \neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R \wedge EF(R))])$
After Q until R	$AG(Q \rightarrow \neg E[(\neg S \wedge \neg R)\ \mathcal{U}(P \wedge \neg S \wedge \neg R)])$

Figure 29. An example property pattern (Precedence) [75]

The work [76] applies the pattern-based requirement description to the domain of programmable logic controllers. Their main contributions are two new pattern groups (possibility and fairness), and a tool helping the users to produce the temporal logic expressions based on the patterns. Later a new pattern (liveness, that is the generalization of the possibility pattern) was proposed and applied in a real case study [77].

The work of Preusse et al. [78] follows a slightly different approach. The defined small "patterns" based on the Computational Tree Logic (CTL) that can be combined together freely. The result is a highly restricted English called *Safety-Oriented Technical Language* (SOTL). Although it is considered as a specification method, even they admit that it is not suitable for a complete specification of the behaviour, but to check critical cases.

The paper [79] presents many patterns formalised in CTL and ACTL (Action CTL – CTL on LTS). It can help to formulate new templates or to make questionnaires.

5.2.2.10 Restricted English

The paper [80] targets the automatic conversion of restricted subset of English sentences to CTL. To achieve that, they defined a CTL-to-English translation to get the set of possible sentences (transliteration). Then this language can be extended by synonyms and syntactic variants. Higher level languages are defined too, in order to find the proper trade-off between expressiveness and ease of processing.

Another attempt to use English as formal language is the *Attempto Controlled English* project[5] that does not specifically focus on temporal logics or on formal specification. The recent paper [81] shows a survey and classification on controlled natural languages.

5.2.2.11 Overall Evaluation of (Formal) Specification Languages

Sommerville's Software Engineering Book

In the book "Software engineering" by Ian Sommerville, a whole chapter is devoted to formal specification [82]. It starts with a historical overview about the motivation of formal specification. Later, it discusses the place of formal specification in the development process. The formal specification methods are classified in two dimensions: *algebraic* or *model-based* approaches and approaches to describe *sequential* or *concurrent* systems. The classified methods are Larch, OBJ (algebraic - sequential); Z, VDM, B (model-based - sequential); LOTOS (algebraic - concurrent); CSP, Petri nets (model-based - concurrent). Sommerville claims that formal verification is not widely used for four reasons:

[5] See the webpage of the project: http://attempto.ifi.uzh.ch/site/

- software engineering is successful even without formal methods;
- the time-to-market becomes more important than quality;
- the scope of formal methods is limited (e.g., it is difficult to specify user interfaces, extra-functional properties);
- the scalability (of formal specification and verification) is limited.

He emphasizes that besides the ability of formal verification, formal specification is helpful because it forces the specifier to make a detailed system analysis and it increases the understanding of the specification and the system.

Some examples are presented for the algebraic approach, stating that their applicability is limited (operations should not depend on the results of the previous operations) and they are increasingly difficult to understand as their size grows. Then examples in the Z notation are shown for model-based specification. Even in the presented small example, there are questionable and ambiguous parts. Furthermore, temporal behaviour is not modelled as it is possible but rather clumsy in Z.

The CESAR Project

The CESAR (Cost-efficient Methods and Processes for Safety Relevant Embedded Systems) was an ARTEMIS project (2009-2012) aimed to boost cost efficiency of embedded system development. In 2010, they published a survey report [11] on modelling languages and validation technologies for safety critical systems. They evaluate multiple technologies based on maturity, industrial adoption, and ongoing work. They assess state-of-the-art tools and languages in clusters such as “Requirement and scenario modelling” (e.g., UML Use Cases), “Property description languages” (e.g., OCL, temporal logics, Alloy), “Behaviour modelling” (e.g., UML State Machine, Activity Diagram). Although in a slightly inconsistent way, the document has a relatively big collection of relevant methods and tools.

Knight: Formal Methods in the Industry

The paper of Knight et al. [83] addresses the following question: why formal methods are not used more widely? It states that even if academia claims that formal methods could help to increase a better software quality, these methods are poorly accepted in the industry. The authors proposed an evaluation framework for formal methods to check their strengths and weaknesses from the point of industry. In this paper they focus on the assessment of specification methods, comparing Z, PVS and Statecharts. Their criteria include coverage of all aspects, integration in the development process, support for group development, support for evolution, usability (expressiveness). The specification is intended to serve as a means of communication, this way annotating the specification with explanations, rationale, or assumptions is important for both the use of the specification in later phases and for modifications of the specification. The authors state that the 3 checked languages and their tool support are not really satisfying the above criteria. They also present a small experiment where 3 specification languages were assessed by nuclear engineers (not formal methods experts) on a nuclear case study application. Their opinions were obtained by interviews after an informal presentation of the current specification notation. Although the role of specification was not understood by everyone for first, the nuclear engineers liked the concept of formal specification. Once they had experience with one or more of the formal specification notations, they said they would “never trust a natural language specification again”. They were impressed by the level of understanding of the system that was required to write the specifications and felt that with natural language specifications they could never be sure that the words were not just copied down with little understanding of the system. Z had a good welcome, but the mathematical notation was not convenient; PVS seemed too complex; and Statecharts had the best scores. Similar opinions were given by the interviewed computer engineers. The report [84] is a much more detailed version of the paper including the models and the questionnaires.

Comparison of Six Formal Methods

In [85] six formal methods were compared for specifying safety critical software. The applicability of the Integrated Approach (IA), the Software Cost Reduction (SCR), the Coloured Petri Nets (CPN), the Statecharts, the Z and the Prototype Verification System (PVS) methods were studied in the nuclear domain. A small example is presented in each studied formal language. Their final conclusion is the following: “We consider best the Statechart description of the system for external behavioural analysis and tabular notation for analysis of the internal conditions and calculations for algorithms".

Boeing Case Study

The paper [86] shows a case study from Boeing. They present lessons learned about statechart modelling and its tool support. They claim that statecharts is an intuitive representation, as it is close to the engineers’ intuition or how they describe the requirements in non-formal English specification. This simplicity provided a good learning curve for the developers. The statecharts even allowed the simplification of requirements in many cases. Also, the dynamic execution and visualization of statecharts helped the developers in the early validation. Furthermore, statechart models facilitate the clear communication. The paper concludes with the statement that using statecharts can even can “make the [specification] process an enjoyable one”.

5.2.3 New Challenges and Required Extensions

In our case, the new aspects of monitoring that are not addressed by the existing solutions in an integrated way can be identified as follows:

- *Monitoring of context-aware autonomous behaviour*: The properties to be monitored may include reference to the perceived context. Accordingly, the context (context fragment) is a concrete language element in the property languages, and the monitor shall perform continuous mapping between the perceived run-time context and the context fragments given in the property specification.
- *Monitoring of dynamic reconfiguration* (e.g., during error handling or due to context changes): The property to be monitored may include reference to the internal configuration of the robot. Accordingly, the configuration (configuration fragment) is a language element in the property languages, and the monitor shall be able to compare the runtime configuration of the robot with the specified configuration fragment.

  In the following, for the sake of simple language definitions, (*fragments of) the perceived context* and *(fragments of) the runtime configuration* are referred to commonly as *context (fragments)*. This way the internal configuration as “internal context” is handled in a similar way as the “external context”, as their semantics from the point of view of runtime monitoring is the same: fragments specified in the property language shall be matched with the detected runtime (internal and external) context.
- *Monitoring of timely context-aware behaviour and/or reconfiguration*: The monitored properties that refer to the perceived context and/or the internal configuration may also include timing information.

Regarding the use of specification languages, the main challenge is the proper integration of user-friendly (semi-formal) languages and formal (intermediate) languages. On the basis of the evaluation presented in Section 5.2.2.11, *statecharts* (a restricted variant of UML 2 State Machines) and *scenarios* (based on the LSC constructs) were selected as engineering languages, while *automata* and *temporal logic* as formalisms that have precise semantics supporting monitor source code generation. Extensions were defined to cover context-aware behaviour, dynamic reconfiguration, and time-dependent behaviour.

In this deliverable the corresponding languages are presented. The detailed design of the mapping among them and the monitor synthesis solutions will be described in deliverable D34.31/32 “Design of the monitoring infrastructure”.

5.3 Specifying Reference Automata

The expected behaviors of a checked component can be described in terms of an abstract reference automaton. This section describes a specification language for timed symbolic transition systems that is suitable to represent such automata.

In our framework the reference automata may be specified in two ways:

- It is possible that the reference automata are given directly by the developer.
- It is possible that the reference automata are derived from other higher level formalisms (like statechart models), this way they are used as an intermediate representation (as presented in Section 5.1).

To support the first option, we propose a powerful language to define the reference behaviour. The language consists of two parts:

- A constraint language is used to specify constraints that can be evaluated (e.g., local conditions in states).
- A system specification language that specifies (parametric) timed systems.

5.3.1 The Constraint Specification Language

This chapter introduces our general purpose constraint language that is suitable (in general) for defining satisfiability problems over complex data types.

The language consists of two sublanguages, the Type language and the Expression language, for which relating procedures like type checking and type inference are defined. The syntax and the notions of the language are inspired by the SAL language [87].

For the syntax of the language, the following building blocks are used:

$$\begin{aligned}
\textit{Digit} &:= \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\textit{Letter} &:= \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{Z} \\
\textit{Name} &:= (\textit{Letter} \mid \texttt{_})^{+}\ (\textit{Letter} \mid \textit{Digit} \mid \texttt{_})^{*} \\
\textit{Numeral} &:= (\textit{Digit})^{+}
\end{aligned}$$

The Constraint language has three components: type declarations, constant and function declarations, and constraints.

$$\begin{aligned}
\textit{Specification} &:= \textbf{specification}\ \textit{Name}\ \{\,(\,[\textit{Declaration}]_{,}^{+}\,)\,\}\ \{\,[\\
&\quad \textit{TypeDeclaration} \mid \\
&\quad \textit{ConstantDeclarartion} \mid \\
&\quad \textit{FunctionDeclaration} \mid \\
&\quad \textit{BasicConstraint} \\
&\quad]^{*}\,\} \\
\textit{TypeDeclaration} &:= \textbf{type}\ \textit{Declaration} \\
\textit{ConstantDeclarartion} &:= \textbf{const}\ \textit{Declaration} \\
&\quad \{\texttt{:=}\ \textit{Expression}\}
\end{aligned}$$

FunctionDeclaration := **function** *Name* ([*Declaration*], *) **:** *Type*
{**:=** *Expression*}

BasicConstraint := **constraint** *Expression*

5.3.1.1 Type Language

The Type language specifies the types available in the language. The types of the language include simple types (like integers), subranges, subtypes of a type constrained by an expression, and complex types like array or tuple types.

Type := *TypeDefinition* |
TypeReference

TypeDefinition := *BasicTypeDefinition* |
EnumerationTypeDefinition |
SubTypeDefinition |
SubRangeTypeDefinition |
FunctionTypeDefinition |
ArrayTypeDefinition |
TupleTypeDefinition |
RecordTypeDefinition |

TypeReference := *Name*

BasicTypeDefinition := **boolean** |
integer |
real |
natural

SubTypeDefinition := { *Declaration* | *Expression* }

SubRangeTypeDefinition := [(**-inf** | *Expression*) **to** (**inf** | *Expression*)]

ArrayTypeDefinition := **array** *Type* **of** *Type*

EnumerationTypeDefinition := **enum** { [*EnumerationLiteral*], + }

RecordTypeDefinition := **record** { [*Declaration*], + }

TupleTypeDefinition := **tuple** { [*Type*], + }

FunctionTypeDefinition := **function** ([*Declaration*], *) **returns** *Type*

5.3.1.2 Expression Language

The Expression language deals with operators of the language. The Expression language does not make a distinction between terms and formulae, as the later are interpreted as expressions of type Boolean. Operators include (simple and complex) literals, Boolean and temporal connectives, quantifiers, arithmetic operators and predicates, polymorphic equality, and operators to decompose complex types.

Expression := ... | (*Expression*)

BooleanLiteralExpression := *FalseExpression* | *TrueExpression*

FalseExpression := **false**

TrueExpression := **true**

IntegerLiteralExpression := *Numeral*

DecimalLiteralExpression := *Numeral* . *Numeral*

RationalLiteralExpression := *Numeral*%*Numeral*

EnumerationLiteralExpresion := ::*Name*

FunctionLiteralExpression := **lambda** ([*Declaration*],$^{+}$) := *Expression*

ArrayLiteralExpression := [*Declaration* | *Expression*]

TupleLiteralExpression := (#[*Expression*],$^{+}$#)

RecordLiteralExpression := {[*Name* := *Expression*],$^{+}$}

ReferenceExpression := *Name*

InExpression := *Expression* **in** *Type*

PrimedExpression := *Expression*′

FunctionAccessExpression := *Expression* ([*Expression*],*)

ArrayAccessExpression := *Expression* [*Expression*]

TupleAccessExpression := *Expression* ! *Numeral*

FieldAccessExpression := *Expression* . *Name*

UnaryMinusExpression := −*Expression*

UnaryPlusExpression := +*Expression*

AddExpression := *Expression* [+ *Expression*]$^{+}$

SubtractExpression := *Expression* − *Expression*

MultiplyExpression := *Expression* [* *Expression*]$^{+}$

DivideExpression := *Expression* / *Expression*

DivExpression := *Expression* **div** *Expression*

ModExpression := *Expression* **mod** *Expression*

GreaterExpression := *Expression* > *Expression*

GreaterEqualExpression := *Expression* >= *Expression*

LessExpression := *Expression* < *Expression*

LessEqualExpression := *Expression* <= *Expression*

EqualityExpression := *Expression* = *Expression*

InequalityExpression := *Expression* /= *Expression*

NextExpression := **X** *Expression*

FinallyExpression := **F** *Expression*

GloballyExpression := **G** *Expression*

UntilExpression := *Expression* **U** *Expression*

ReleaseExpression := *Expression* **R** *Expression*

TemporalForallExpression **A** *Expression*

TemporalExistsExpression **E** *Expression*

ForallExpression := **forall** (*[Declaration]*,$^{+}$) : *Expression*

ExistsExpression := **exists** (*[Declaration]*,$^{+}$) : *Expression*

NotExpression := **not** *Expression*

AndExpression := *Expression* [**and** *Expression*]$^{+}$

OrExpression := *Expression* [**or** *Expression*]$^{+}$

ImplyExpression := *Expression* **imply** *Expression*

EqualExpression := *Expression* **equal** *Expression*

IfThenElseExpression := **if** *Expression* **then** *Expression* **else** *Expression*

5.3.1.3 Example Specification

To demonstrate the capabilities of the constraint language, in the following as example the specification of a finite Abelian group is given.

```
specification FiniteAbelianGroup(n : natural) {
      type G : [ 1 to n ]

      const e : G
      function op(a : G, b : G) : G
      function inv(a : G) : G

      constraint forall (a : G, b : G, c : G) :
            op(a, op(b, c)) = op(op(a, b), c)
      constraint forall (a : G, b : G) :
            op(a, b) = op(b, a)
      constraint forall (a : G) : op(a, e) = a
      constraint forall (a : G) : op(a, inv(a)) = e
}
```

An (up to isomorphism unique) model for `FiniteAbelianGroup(2)` is induced by the following set of equalities:

```
e = 1
op = (lambda (a : G, b : G) := if a = b then 1 else 2)
inv = (lambda (a : G) := a)
```

5.3.2 The System Specification Language

In this section our language for specification of (parametric) timed systems is presented. Both the syntax and semantics of the language is based on the Constraint language presented in the previous section. Similarly to the Constraint language, it also is inspired by the intermediate language SAL. However, the language has some improvements to SAL in descriptive power:

- *Direct modeling support for timed behaviour*. The language presented below offers direct modeling support, like clock variables, state invariants and urgency constraints.
- *More flexible command structure*. In the language, different types of commands can be combined freely. Moreover, multiple definitions for the same variable in a composite command need not be collected in a single assignment manually.

Specification is given in the following form:

Specification := **specification** *Name* { ([*Declaration*]$_{,}^{+}$) } { [
TypeDeclaration | … |
TemplateDeclaration |
PropertyDeclaration
]* }

In order to express timing, the System language extends the set of basic types by the new type **clock**.

BasicTypeDefinition := **boolean** | … |
clock

5.3.2.1 System Language

The System language provides the notions for defining and composing systems.

System := *SystemDefinition* |
SystemReference |
AsynchronousCompositeSystem |
SynchronousCompositeSystem |
AsynchronousMultiSystem |
SynchronousMultiSystem

SystemDefinition := { [
VariableDeclaration |
DefinitionDeclaration |
SystemConstraint |
BehaviorDefinition
]* }

SystemReference := *Name* ([*Expression*]$_{,}^{*}$)

AsynchronousCompositeSystem := *System* [ ] *System*

SynchronousCompositeSystem := *System* || *System*

AsynchronousMultiSystem := **async** (*Declaration*) : *System*

SynchronousMultiSystem := **sync** (*Declaration*) : *System*

VariableDeclaration := *InputVariableDeclaration* |
OutputVariableDeclaration |
GlobalVariableDeclaration |
LocalVariableDeclaration

InputVariableDeclaration := **input var** *Declaration*

OutputVariableDeclaration := **output var** *Declaration*

GlobalVariableDeclaration := **global var** *Declaration*

LocalVariableDeclaration := **local var** *Declaration*

DefinitionDeclaration := **definition** *Declaration* := *Expression*

SystemConstraint := *UrgencyConstraint* |
InvariantConstraint

UrgencyConstraint := **urgent** *Expression*

InvariantConstraint := **invariant** *Expression*

BehaviorDefinition := *InitializationDefinition* |
TransitionDeifinition

InitalizationDefinition := **initialization** *SimpleCommand*

TransitionDefinition := **transition** *SimpleCommand*

5.3.2.2 Command Language

The Command language specifies the means for describing behavior of a system.

Command := *SimpleCommand* |
ElseCommand

SimpleCommand := *AsynchronousCompositeCommand* |
SynchronousCompositeCommand |
AsynchronousMultiCommand |
SynchronousMultiCommand |
GuardedCommand |
AssignmentCommand

AsynchronousCompositeCommand := **async** {
[*Command*]*
}

SynchronousCompositeCommand := {**sync**} {
[*SimpleCommand*]*
}

AsynchronousMultiCommand := **async** (*Declaration*) :
SimpleCommand

SynchronousMultiCommand := **sync** (*Declaration*) :
SimpleCommand

GuardedCommand := *Expression* --> *SimpleCommand*

AssignmentCommand := **let** *Expression*;

ElseCommand := **else** --> *SimpleCommand*

5.3.2.3 Example Specification

To demonstrate the capabilities of the system specification language, a solution (the Fischer protocol[6]) for a mutual exclusion problem is presented. Note that for the sake of completeness also a safety property of the protocol is included.

```
specification MutualExclusion(maxId : integer) {
      type Id : [1 to maxId]
      type Location : enum { sleeping, waiting, trying, critical }
      const a : natural
      const b : natural
      constraint a < b

      system Initializer := {
            global var lock : [0 to maxId]

            initialization let lock = 0;
      }

      system Fischer(id : Id) := {
            global var lock : [0 to maxId]
            local var location : Location
            local var c : clock
            invariant location = ::waiting imply c <= a

            initialization let location = ::sleeping;

            transition location = ::sleeping and lock = 0 --> {
                  let location' = ::waiting;
                  let c' = 0%1;
            }

            transition location = ::waiting and c <= a --> {
                  let location' = ::trying;
                  let lock' = id;
                  let c' = 0%1;
            }

            transition location = ::trying and
            c >= b and lock /= id --> {
                  let location' = ::sleeping;
            }
```

[6] This protocol is described among others in Stan Budkowski, Ana Cavalli, Elie Najm: Formal Description Techniques and Protocol Specification, Testing and Verification. Springer Science & Business Media, 1998.

```
            transition location = ::trying and
            c >= b and lock = id --> {
                  let location' = ::critical;
            }

            transition location = ::critical --> {
                  let location' = ::sleeping;
                  let lock' = 0;
            }
      }

      system FischerNetwork :=
            Initializer || async (i : Id) : Fischer(i)

      property safe : FischerNetwork models G(
            forall (i : Id) : (
                  location[i] = ::critical imply lock = i
            )
      )
}
```

5.3.2.4 Defining a Reference Automaton

Consider a component whose expected behavior is to keep speed low until a safety function is turned on. To express this requirement, a reference automaton and a simple invariant property is defined. The property is used for local checking of the automaton specification itself and means that the Boolean property “faulty” (when the location of the automaton is s2) is not true globally (as denoted by the G temporal operator) during the execution.

From such an automaton, the monitor code can be generated automatically.

```
specification Monitoring {

      system Monitor {
            input var speed_low : boolean
            input var safety_on : boolean
            local var loc : enum { s0, s1, s2 }
            definition faulty : boolean := loc = s2

            initialization async {
                  speed_low and not safety_on --> let loc = s0;
                  safety_on --> let loc = s1;
                  else --> let loc = s2;
            }

            transition async {
                  loc = s0 and speed_low and not safety_on -->
                        let loc’ = s0;

                  loc = s0 and safety_on --> let loc’ = s1;

                  loc = s0 and not speed_low and not safety_on -->
                        let loc’ = s2;
```

```
                loc = s1 --> let loc' = s1;

                loc = s2 --> let loc' = s2;
            }
        }

        property safe : Monitor models G not faulty
}
```

5.4 Specifying Temporal Properties

In this section another language is defined that is used to express requirements to be monitored. The goal of this language is to be able to easily and unambiguously express temporal properties for context-aware behaviour of dynamic systems. Besides the expressiveness of the language, the lightweight runtime checking of properties defined with the language shall be supported – even in environments with limited resources. For this high level demand the following features are required:

- Explicit context definitions: Context shall explicitly appear in the requirements.
- Timing: Time-dependent behaviour shall be expressed.
- Modality: Requirements shall define mandatory and optional behaviour.
- Requirement activation: Ordering between the requirements shall be supported.

In our framework this language may be used in two ways:

- The language is used as a property specification language and its expressions (the checked properties) are directly given by the developer.
- The language is used as an intermediate language, and its expressions are mapped from higher level graphical languages (as presented in Section 5.1).

In any case, the properties specified using this language form the input of the monitor synthesis tool. As textual languages are easier to be processed in an automated way, a textual concrete syntax is used.

Propositional Linear Temporal Logic (PLTL) [64] is extensively used for defining requirements, and particularly popular in runtime verification frameworks. PLTL expressions can be introduced as logic expressions that can be evaluated on a trace of steps, in which each step can be characterized by atomic propositions. Here atomic propositions are local characteristics of the step that may include all elements of a monitored execution trace that are relevant from the point of view of property monitoring: function call, function return, input or output signal, message received or sent, timer started or expired, state entered or left, context change, configuration change, predicate on the value of a variable etc. Later, we will call these atomic propositions in general as “events”, and the trace of steps is the “trace of events”.

Besides the usual Boolean language operators, PLTL has the following temporal operators:

- X: “Next” operator (X P means that the next step in the trace shall be characterized by the atomic proposition P).
- U: “Until” operator (P U Q means that a step characterized by the atomic proposition Q shall eventually occur, and until that occurrence all steps of the trace shall be characterized by P).
- G: “Globally” operator (G P means that each step in the trace shall be characterized by P).

- F: “Future” or “Eventually” operator (F P means that eventually a step shall occur in the trace that is characterized by P).
- W: “Weak until” operator (P W Q means that either there is no step in the trace characterized by Q, or a step characterized by the atomic proposition Q shall eventually occur and until that occurrence all steps of the trace shall be characterized by P).

In spite of there are several extensions of PLTL, for various purposes, there is no context-aware temporal logic that can be found in the literature. For this reason we defined a new extension of PLTL, the *Context-aware Timed Propositional Linear Temporal Logic* (CaTL).

Although no context-aware PLTL existed before, there are many examples for handling timing constraints within PLTL expressions. These can be used as a starting point, thus a short overview is presented in Section 5.4.1. Context modelling aspects are recapitulated in Section 5.4.2. After that, the syntax (Section 5.4.3) and the semantics (Section 5.4.4) of the CaTL formalism is defined. Finally, a few CaTL examples are presented in Section 5.4.5,

5.4.1 Overview of LTL’s Timing Extensions

The goal for extending LTL languages with timing support is clear: one should be able to define requirements regarding the timing of systems. The PLTL formalism is interpreted over models which retain only the temporal ordering of the states, losing the precise timing information. Therefore PLTL on its own cannot specify requirements, like “*An alarm must be raised, if the time difference between two successive states is more than 5 time units*”. To tackle this issue, various extensions can be found [65].

- The *Explicit Clock Temporal Logic* (XCTL) formulas [66] contain static timing variables and an explicit clock variable (referring to the current time). This logic allows capturing the different values of the global clock in the timing variables. The captured values can be then used in expressions. The previous example requirement for the alarm can be formalised as follows.

$$G(t_1 = c \rightarrow X(c > t_1 + 5 \rightarrow alarm))$$

- The *Metric Temporal Logic* (MTL) [67] formalism has a different approach. Instead of having timing variables, it has time bounded temporal operators. This makes it quite convenient to read expressions. Even the previous example is very easy to understand if expressed using MTL.

$$G(p \rightarrow F_{\geq 5} alarm)$$

- The *Timed Propositional Temporal Logic* (TPTL) language introduced in [68] utilizes the so-called *freeze quantification*. It means that each variable can be bound to the time of a particular state (similarly to XCTL). The freeze quantifier appears in the syntax as a dot: $x_i.\varphi$ means that the x_i variable is bound to the time of φ. The earlier example can be expressed with TPTL like this:

$$G(t_1.X(t_2.(t_2 > t_1 + 5 \rightarrow alarm)))$$

- The *Timeout based Extension of PLTL* (TLTL for short) [65] uses static and dynamic timing variables with the explicit global clock, which makes it flexible and expressive. Without going deeper into the syntax and semantics of the language, the same example looks as follows.

$$G((x = t_0) \rightarrow X((x > t_0 + 5) \rightarrow alarm))$$

  A more detailed explanation of this kind of temporal logics will follow in the next section where the CaTL formalism is introduced, since this approach was selected for the timing extension to be used in CaTL.

5.4.2 Context Modelling

Context is referenced in the CaTL formulas using context fragments. A context fragment is an instance model of the context metamodel (Section 3.1). It can be represented as a UML package with the name of the context fragment. The relevant part of UML's abstract and concrete syntax (the metaclasses *Package*, *InstanceSpecification* and *Slot*) is not modified.

As the context view consists of instances from the context metamodel, the creation of the context metamodel becomes a significant part of creating requirements. At first the context metamodel contains the classes, its properties and the links between them. Moreover the context metamodel shall define the well-formedness and semantic constraints. Well-formedness constraints define constraints that must be satisfied by any context model, otherwise conceptual rules or the laws of physics are violated. Semantic constraints are derived from the requirements of an application, this way these are only preconditions or expectations about the context that can be violated in particular cases (e.g., when the robustness of an autonomous system is exercised).

5.4.3 The Syntax of CaTL

The basic vocabulary of CaTL consist of a finite set P of propositions, a finite set T of static timing variables and a finite set C_M of static context variables.

Each $c_i \in C_M$ is an instance of M context metamodel ($c_i \propto M$). A context metamodel is defined as a 2-tuple *M = (N,R)*, where N represents the set of classes in the metamodel and R represents the relations (i.e., association or generalization) in the model. An $n_i \in N$ is a class, which has a set of properties. Each property has a name and a type (e.g., Boolean or string). One can create an E_M set of predefined contexts, where $e_i \propto M$ for all $e_i \in E_M$. The context variables and the predefined contexts contain instances of the classes (objects) from M. Each object has a unique identifier and an $n_i \in N$ class. The values of the properties can be defined by property constraints (defined later), which refer to the objects by the unique identifiers given in the variables. It is important, that if two context variables contain two objects with the same identifier, then that two objects must be equivalent.

In addition, one can use two dynamic variables: t, which represents the clock and e, which represents the context of the system. M must be defined in such a way, which ensures that e is always a valid instance of M ($e \propto M$).

A_f is the set of atomic formulas, which consists of propositions from P, atomic timing constraints, context constraints and property constraints.

- Propositions are labels, referring to properties of a system. Each proposition can be evaluated to true or false in each state of the system.

  Examples: *initialized*, *connected*

- The timing constraints are defined in the following form: $t \sim u$, where t is the dynamic clock variable, $\partial \sim \{<, >, =\}$, $u \in \{t_i + c, c\}$, $t_i \in T$ and $c \in N$.

  Examples: $t = t_0$, $t < t_0 + 5$

- The context constraints are defined in the following form: $x \approx y$, where $x \in E_M \cup V_M$ and $y \in C_M \cup \{e\}$. In this notation $\approx$ is a compatibility relation (meaning x is compatible with y) and V_M is a set of context definitions. Context definitions are instances of the M metamodel. A context definition can be one of the followings:

  - a static context variable ($c_i \in C_M$), or
  - a new context, created from a static context variable, with one of the following operators:

- *Node exclusion*: z - v, where z is a context definition and v is a present class instance of z,
- *Node addition:* z + w, where z is a context definition and w is an instance of the classes of M,
- *Connection exclusion*: z - - a, where z is a context definition and a is a present connection in z,
- *Connection addition*: z + + b(c, d), where z is a context definition and b is connection between c and d, compatible with M.

Examples: $e_0 \approx e$, e_0 - x $\approx$ e

- The property constraints are expressions over properties of an object. The following syntax is defined to unambiguously select a p property: context.object.p. The syntax of the property constraints is: p ~ v, where p is a property, v is a value, which has to be from the same type as the property, and ~ is a comparison operator, which can be evaluated to a Boolean value.

  Examples: e0.a.connected = true, e_1.b.speed < 10

For each atomic formula, Υ assigns the modality of that atomic formula (a so-called "temperature"): $\Upsilon : A_f \rightarrow$ {hot, cold}. An atomic formula with *hot temperature* is a mandatory, while *cold formulas* are optional. The notation of the modality is the following. If no additional notation is given, then the modality of the atomic formula is hot (mandatory). The cold (optional) modality of the a_f atomic formula is written like < a_f >.

A ϕ CaTL formula can be one of the followings:

- Atomic formula: $a_f \in A_f$
- Disjunction: $\phi \vee \phi$
- Negation: $\neg\phi$
- "Next" operator: X ϕ
- "Until" operator: ϕ_1 U ϕ_2

All static variables used in a formula are implicitly quantified with a universal quantifier. Additional operators can be defined with the previously defined ones as syntactical abbreviations. The most commonly used abbreviations are defined as follows:

- Conjunction: $a \wedge b = \neg (\neg a \vee \neg b)$
- Implication: $a \rightarrow b = \neg a \vee b$
- "Eventually" operator: F ϕ = true U ϕ, where "true" denotes the Boolean true value
- "Globally" operator: G$\phi = \neg$(F $\neg\phi$)
- "Weak until" operator: ϕ_1 W ϕ_2 = (G ϕ_1) $\vee$ (ϕ_1 U ϕ_2)

5.4.4 The Semantics of CaTL

Formally, the CaTL formulas are interpreted over finite traces of *Context-aware Kripke-structures* (CaKS). A CaKS is the extended version of the classical Kripke-structure, which is the mathematical abstraction of finite state-transition systems with labelled states. A CaKS for a P set of labels and an M context metamodel is defined as a 6-tuple: CaKS = (S, T, I, L, C, E), where

- S is a finite set of states.
- T $\subseteq$ S$\times$S is the state transition relation.

- $I \in S$ is the initial state of the system.
- L is a labelling function, which assigns labels to states $L: S \rightarrow 2^P$.
- C function assigns clock value to each state: $C: S \rightarrow N$, and the clock value assigned to the initial state is 0 ($C(I) = 0$) and if $(a, b) \in T$, then $C(b) >= C(a)$, so the time is not decreasing.
- E function assigns a context to each state: $E: S \rightarrow C$, where C is the set of context instances ($\forall c \in C: c \propto M$).

Thus a CaKS model is a finite state-transition system, where a context and a clock value are assigned to each state of the system.

A *finite trace* of a CaKS is sequence of states connected by the transition relation: $\Pi = (s_0, s_1, s_2, \ldots s_{n-1})$, where $s_i \in S$ ($i \in [0, n- 1]$), $n > 0$, $s_0 = I$ and $\forall_{0<i<n} : (s_{i-1}, s_i) \in T$. A Π^j *trace suffix* is defined by removing the first j steps ($j < n$) from the Π trace. By definition $\Pi^0=\Pi$.

The inductive definition of the semantics of a ϕ CaTL formula is given below. The notation $\Pi^i \models \phi$ means, that the ϕ formula is true on Π^i trace suffix. The $x|_{a=b}$ notation is used for substituting the a variable in x with the b value.

- $\Pi \models p$ if and only if $p \in L(s_0)$, where $p \in P$ is an atomic proposition.
- $\Pi \models c$ if and only if $c|_{t=C(s0)}$ is true, where c is a timing constraint.
- $\Pi \models d$ if and only if $d|_{e=E(s0)}$ is true, where d is a context constraint.
- $\Pi \models f$ if and only if $f|_{e=E(s0)}$ can be evaluated, and evaluated to true, where f is a property constraint. If f contains a property, which does not exist, then the constraint will be evaluated to false.
- $\Pi \models \neg\phi$ if and only if $\Pi \models \phi$ is not true
- $\Pi \models \phi_1 \vee \phi_2$ if and only if $\Pi \models \phi_1$ or $\Pi \models \phi_2$
- $\Pi \models X\ \phi$ if and only if length of Π is at least 2 ($n>1$) and $\Pi^1 \models \phi$
- $\Pi \models \phi_1\ U\ \phi_2$ if and only if $\exists 0 <= i < n: \Pi^i \models \phi_2$ and $\forall 0 <= j < i: \Pi^j \models \phi_1$

Lastly, a few terms concerning the contexts must be defined. An e_1 context is compatible with e_2 (denoted as $e_1 \approx e_2$) if, and only if, exists a bijective function between the two object sets e_1 and e_2, which assigns a compatible object to each object. Two objects are compatible, if and only if both have the same type and have the same relations to other objects. Therefore if the compatibility function assigns $o_2 \in e_2$ to $o_1 \in e_1$ and $o_4 \in e_2$ to $o_3 \in e_1$ and there is an edge between o_1 and o_3, then an edge must be present in e_2 between o_2 and o_4, with the same label as in e_1. Note that the context compatibility relation does not require the equality or compatibility of the properties of the objects, only the object types and relations are concerned.

The compatibility relation defines a bijective function, thus in $e_1 \approx e_2$ for all objects in e_1 (the O_{e1} notation will be used in the future), there must be an object from O_{e2} assigned, and if o', o'' $\in O_{e1}$ are two different objects then the assigned objects from O_{e2} must be different. It is also required to assign an object to all objects in O_{e1}, but objects in O_{e2} can remain without an assigned counterpart. An example for two compatible contexts can be found on Figure 30. It is possible that more than one assignment function between two contexts exist.

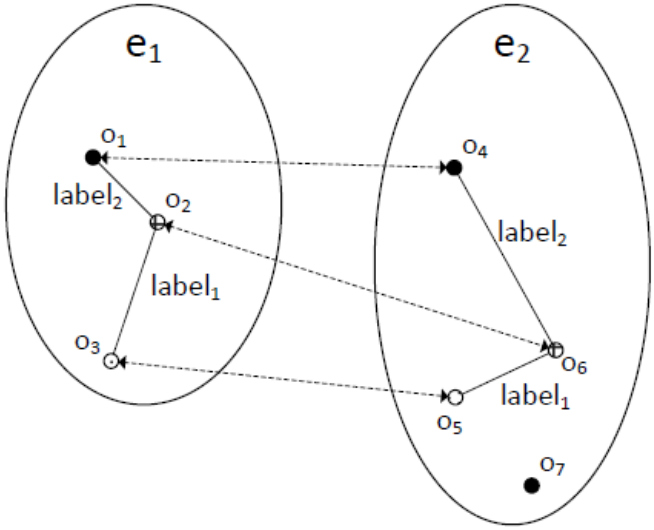


Figure 30. The compatibility relation between two contexts

The objects in the context variables have unique identifiers, meaning that if two objects with the same identifier appear in two contexts then these two objects are identical. This constraint results that the assignments between the objects are immutable. Each object has only one identifier, thus two different identifiers always mean two different objects.

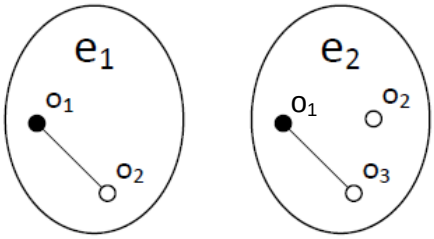


Figure 31. Context definition for illustrating the immutability of object assignments

Here follows an example to illustrate the effects of the previous constraints. Two contexts are defined on Figure 31. The $\phi = e_1 \approx e \wedge X(e_2 \approx e)$ requirement means that currently the o_1 object must be connected to o_2, but in the next state of the system the connection must be dropped and a new connection with a new object must be made, as o_1 is connected to o_3 in e_2 instead of o_2.

5.4.5 Examples for CaTL

After getting through syntax and semantics of CaTL, let us consider some easy to understand examples. First of all, by the definitions CaTL is an extension of PLTL, thus any valid PLTL formula is a valid CaTL formula. The following formulas are all valid CaTL formulas. The meaning for each formula is also given.

- $G(connected \rightarrow F(disconnected))$

  It is always true, that if the system is in the connected state, then it will eventually become disconnected.

- $G(connected \wedge t_0 = t \rightarrow F(disconnected \wedge t < t_0 + 5))$

  It is always true, that if the system is connected, then it will be disconnected in 5 seconds (it is assumed that the time unit is a second).

- $G(connected \wedge e_1 \approx e \wedge X(disconnected) \rightarrow F(e_2 \approx e))$

It is always true, that if the system is connected and in the e_1 context (from Figure 31) and will be disconnected in the next state, then eventually it will be in the e_2 context. It can be also phrased as follows: If the system is connected to an object and disconnects from it, then it will eventually be connected to another object.

5.4.6 A Concrete Syntax for CaTL

To be able to represent CaTL formula in machine-readable textual format, we propose the following concrete syntax of the operators (Table 4).

Operator	Concrete textual syntax
$\neg$	not
$\wedge$	and
$\vee$	or
$\rightarrow$	implies
X	Next
U	Until
G	Globally
F	Eventually
W	Until*
$\approx$	Compatible

Table 4: Concrete textual syntax for PLTL operators

The other operators (=, <, >, +, ++, -, --) are used in their usual mathematical form.

5.5 Describing Code Contracts

Code contracts are used in the *design by contract* approach to specify the expected behaviour of a component or function, typically on its interface. According to B. Meyer, the key concept of design by contract in object oriented programming is “viewing the relationship between a class and its clients as a formal agreement, expressing each party's right and obligations” [88]. Code contracts can be checked both by static analysis and runtime verification. On the basis of the analysis of the existing approaches (Section 5.5.1) we define a language that will be used for describing the code contracts for on-line verification by executable code snippets (Section 5.5.2.

5.5.1 Existing Approaches

Code contracts were introduced into high level programming languages like C, C#, and Java. In the following we describe two approaches that demonstrate the basic concepts and typical language constructs that are used in languages describing the code contracts.

5.5.1.1 C# Code Contracts

Spec# being developed by Microsoft Research is a formal language for API contracts (influenced by JML, AsmL, and Eiffel). It extends C# with constructs for non-null types, preconditions, postconditions, and object invariants. Spec# comes with a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks and multi-threading. Spec# is a programming language providing support for the definition of pre- and postconditions.

In the example below the precondition requires the variable x to be greater or equal to zero, while the post condition ensures that the result of the function satisfies basic properties of a square root computing function.

```
requires 0 <= x;
ensures result*result <= x && x < (result+1)*(result+1);
```

While Spec# is a prototype with static analysis and offline verification techniques, code contracts in .Net Framework 4.5 provides an efficient language to do runtime analysis. Basic language elements are the following.

- *Preconditions* are expressed by using the method Contract.Requires. For example, the following example expresses that the parameter “y” must not be zero.

```
Contract.Requires(y != 0);
```

- *Postconditions* are contracts for the state of a method when it terminates. For example, the following example expresses that the parameter “y” must be less or equal zero upon termination.

```
Contract.Ensures(this.y <= 0);
```

- *Invariants* are conditions that should hold (evaluate to true) during the life cycle of the object. For example, the following invariant expresses that the parameter “y” must be less or equal to zero at every time point of the life cycle of the object (this).

```
Contract.Invariant(this.y <= 0);
```

A restriction regarding the called functions of the contracts is that they must be pure: they must not modify/update any pre-existing state. Pure methods are only allowed to modify objects that have been created after the entry into the pure method.

.Net code contracts can define special conditions called *exceptional postcondition* ensuring that some conditions hold when an exception is thrown.

5.5.1.2 Code Contracts in C

Microsoft Research developed a static verification tool Verifier for Concurrent C (VCC), which is an automated verifier [89]. VCC takes a C program, annotated with function specifications, data invariants, loop invariants, and ghost code, and tries to prove the correctness of these annotations. If it succeeds, VCC promises that the program meets its specifications. By using the language of VCC, the contracts can be transformed to running code, generating in this way assertions for runtime verification. Figure 32 presents the main components of this approach. As it turns out, aspect-oriented programming is used to integrate the checking of the code contracts with the original code.

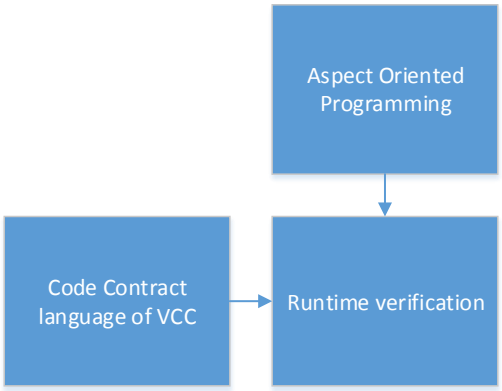


Figure 32. Overview of the components of the VCC on-line verification approach

In the following, the main concepts of the VCC code contract language are exemplified.

The following source code presents the definition of a *binary_search* function and its code contracts. The keyword *requires* defines the preconditions to be fulfilled by the context calling the method. In the example, a first order logic expression states that for all indices, the input buffer is ordered. The keyword *ensures* is used to define postconditions.

```
unsigned binary_search(int val, int *buf, unsigned len)
_(requires \forall unsigned i, j; i < j && j < len ==> buf[i] <= buf[j])       // buffer sorted
_(ensures \result != UINT_MAX ==> buf[\result] == val)                        // val found
_(ensures \result == UINT_MAX ==> \forall unsigned i; i < len ==> buf[i] != val) // val not found
```

The contract language of VCC also supports the definition of invariant properties of the system. For this purpose the *invariant* keyword is used. In the example below a loop invariant is defined to establish a relation between two variables.

```
low = 0; high = len;
while (low < high)
  _(invariant high <= len)
```

This invariant construct is difficult to verify on-line due to the fact that it has to be examined as many times as the loop is executed.

The contract language of VCC introduces the *assert* keyword to define simple guarantees during the program. In the following example an assertion is defined to establish a relation between two variables (similarly to the former example).

```
_(assert high <= len)
```

The advantage of the contract language of VCC is that it is formalised using the Z3 SMT solver.

5.5.2 The Language for Describing Code Contracts

On the basis of the language constructs demonstrated by the examples, we define below our language for describing code contracts. This is based on a subset of the contract language of VCC and includes the elements that are needed for specifying properties for on-line monitoring (and excludes elements that are relevant for correctness proofs but are not easy to check on-line).

The expressions of the language are similar to first order logic. Function parameters and return values can be referred in the expressions, using arithmetic and Boolean logic operators, together with quantification. The roles of expressions (precondition, postcondition, invariant or assertion) are given by predefined keywords. Besides some basic built-in types, the types of the C language can be used. As the expressions are intended to be used for specifying properties, only pure expressions are defined, without any side-effect.

The language is defined by the following set of rules:

Conditions:

Precondition: _(**`requires`** *expression*)

Postcondition: _(**`ensures`** *expression*)

Invariant: _(**`invariant`** *expression*)

Assertion: _(**`assert`** *expression*)

Expressions:

Quantifiers:

`\forall` *type variable*;

`\exists` *type variable*;

Operators:

/
*
%
+
-
<<
>>
<=
=>
<
>
==
!=
&
^
|
&&
||
<==
==>
<==>
\result

Types:

\bool
\natural
\integer
C primitive types

Variables

C variables

Note that C signed types can be cast to \integer and unsigned integral types can be cast to \natural in an obvious way. The comparison operators (<, >, <=, >=, ==, and !=) have the same precedence as in C. The logical operators have the following precedence: &&, ||, <==, ==>, <==>.

Note that contracts may be given in a separate module, this way the declaration of a component (referring to a software element of the configuration model) is necessary.

This contract language will be used to describe expected properties for on-line verification. The contracts (conditions) will be transformed to executable code which verifies the actual behaviour of the system in runtime. Synthesising verification code from first order logic formulae has a long standing history in the literature, so the code generation on the basis of this restricted set of code contracts is viable.

5.6 Describing Reference Behaviour using Statecharts

To enable convenient graphical modelling of reference automata (see in Section 5.3), we suggest the use of the statechart formalism in the form of a subset of the UML State Machine model elements, extended with guards and actions as specified by the reference automata language. The subset supports the following features:

- hierarchy (OR-states) and submachine states,
- concurrency (AND-states) and synchronization,
- communication via structured signal events,

- timing via timed events,
- inter-level transitions,
- variables and data manipulation.

A skeleton of the abstract syntax of the language is depicted in Figure 33. The concrete graphical syntax is the same as in case of UML 2 State Machine diagrams, this way existing tools can be used. Guards and actions (effects) are expressed using the Constraint language presented in Section 5.3.1.

The semantics of this statechart formalism is given by a structure-preserving mapping to the System specification language of the reference automaton (see in Section 5.3.2) on the basis of the formal semantics developed in [62]. Expressions used as guards and actions used as effects are handled by a straightforward mapping to the Constraint specification language (see in Section 5.3.1).

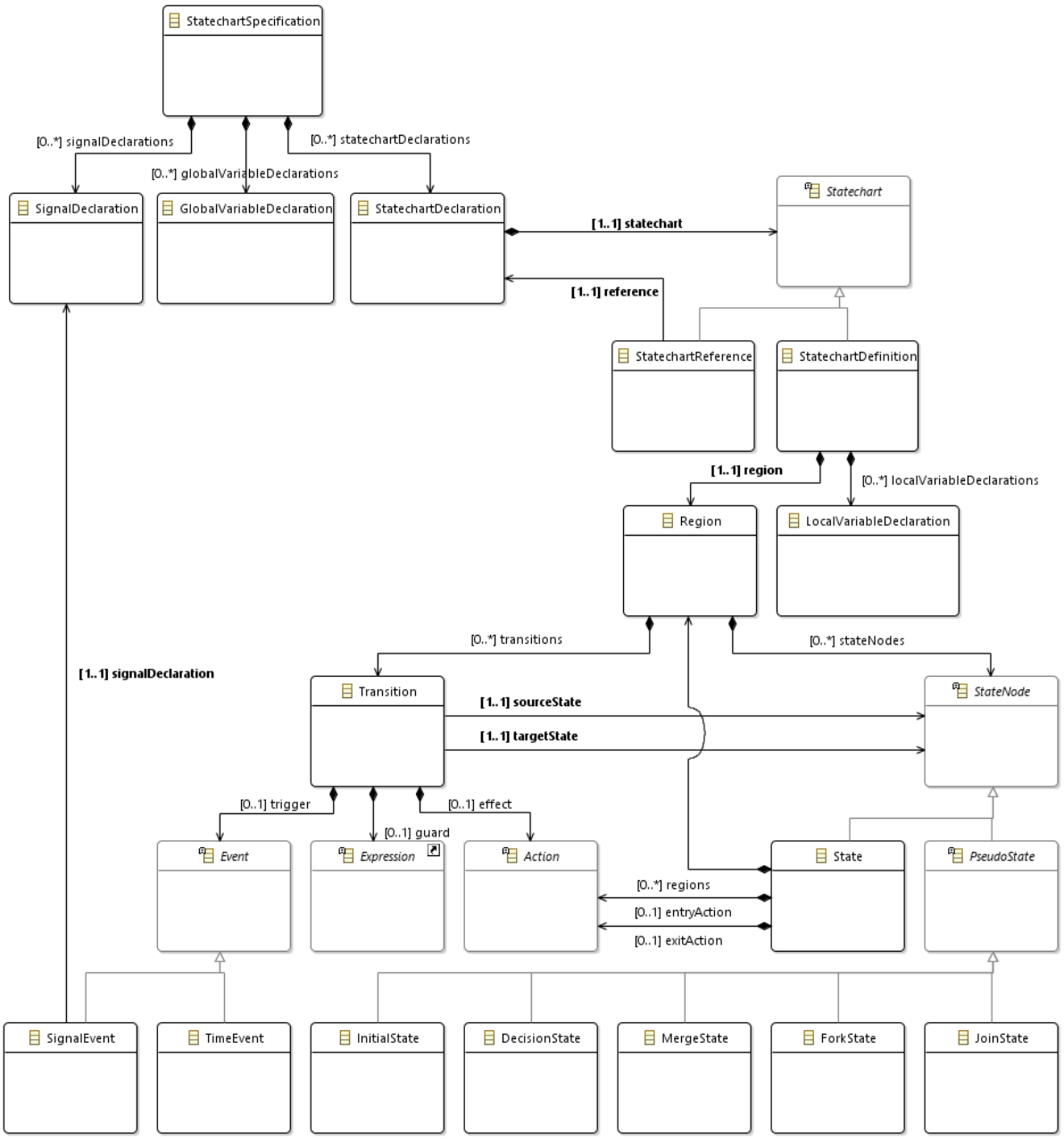


Figure 33. Abstract syntax (metamodel) of the statechart language

5.7 Describing Scenarios

Scenarios are used to specify high-level behavioural properties. According to the approach presented in Section 5.1, these are mapped to reference automata or temporal logic properties that form the basis of generating the source code of on-line monitors.

5.7.1 Syntax of the Language

The scenario formalism is summarized in Section 3.2. The language has two parts: message view and context view. The message view is a modified version of a Sequence Diagram, while the context view consists of zero or more context fragments expressed as Objects Diagrams. The message view contains (by convention) a trigger part (pre-chart) and an assert part (main chart).

To handle the trigger and assert part, a more general concept can be introduced which is inspired by the Live Sequence Chart (LSC) formalism [58]. A so-called *modality* can be added to the model elements (by introducing a new superclass for the InteractionFragment metaclass in the UML metamodel). The modality can be *hot* or *cold*, meaning mandatory or optional behaviour, respectively. This way of addition of modality to Sequence Diagrams is based on Modal Sequence Diagrams [18] and results a flexible way of defining preconditions of behaviour. If a cold InteractionFragment cannot be matched with the behaviour of the system, then the requirement is neither satisfied, nor violated: it is called inconclusive. The trigger part and the assert part can be translated to this extended form by giving cold modality to all fragments in the pre-chart and hot modality to the fragments in the main chart. Giving hot modality to all InteractionFragment results a classical Sequence Diagrams. In the following, we will use these modalities when the semantics of the language are described.

References to context fragments (used in initial, interim or final contexts) are expressed as StateInvariants placed on the Lifeline in the form of *{Context : ContextFragmentName}*.

Here we present 3 basic examples of the graphical requirement specification language.

- The first example (Figure 34) shows a simple Sequence Diagram with *a* and *b* components, where a condition is given (the *x* property of the *a* component shall be 5) and two messages are exchanged.

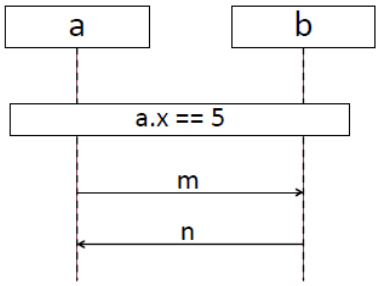


Figure 34. Example of the graphical requirement specification language

- The second example (Figure 35) shows a requirement with a reference to the *C1* context fragment which is defined by an object diagram.

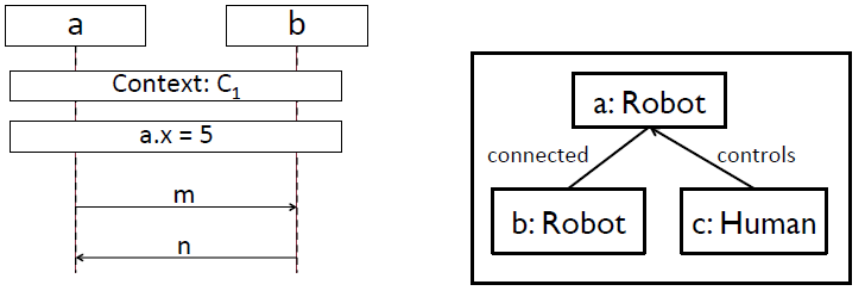


Figure 35. Example requirement with a referenced context fragment: Event view (left) and Context view (right)

- The third example (Figure 36) depicts the use of cold conditions (pre-chart), meaning that if the *x* property of *a* is not 5, then this requirement shall not be checked.

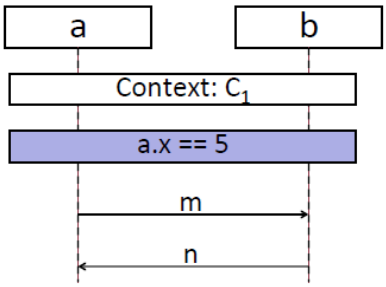


Figure 36. Example of the usage of cold conditions (condition in blue)

The semantics of the scenario language is defined by giving a translation algorithm first to observer automata, then to temporal logic formalism.

5.7.2 Semantics

To define the semantics, an observer automaton is created for each lifeline in the scenario. The role of the automaton is to observe the behaviour of the monitored system and accept the specified behaviour. Note that even though a scenario may include multiple components (lifelines), the semantics is defined only for a single component that is monitored. Accordingly, in this section a recursive *unwinding algorithm* is presented, which takes one lifeline (with the associated events, constraints and other fragments) and translates it to an automaton. The algorithm covers all considered LSC fragments (alt, opt, loop, break, neg) and not only the ones that are used for specifying properties for on-line verification (see in Section 3.2).

The observer automata can be represented using the language introduced for describing reference automata (Section 5.3), this way directly forming the input for the generation of the source code of monitors.

5.7.2.1 Translation of component-level requirements

The unwinding algorithm generates an automaton, which will observe the behaviour of the component and will accept only those runs, which satisfy the requirement. A decision over a requirement can be one of the following:

- *Accept*: The requirement is satisfied, all hot conditions are met and the automaton observed the expected behaviour.
- *Reject*: The requirement is violated.
- *Inconclusive*: The automaton did not observe the preconditions of the requirement (cold conditions), therefore the requirement was not violated, but was not checked either.

An automaton is defined with the following tuple:

$$A_M = (\Sigma, Q, q_0, F_a. q_{inconclusive}, T, Var_E)$$

where

- Σ is the set of transition labels with predicates (it can use variables from the Var_E sets),
- Q is the set of states.
- q_0 is the initial state of the automaton.
- $F_a \subseteq Q$ is the set of accepting states, where if the automaton stops, the requirement will be accepted.
- $q_{inconclusive} \in Q$ is a state, where if the automaton stops, then the requirement was neither accepted nor rejected (the matched run was inconclusive).
- $T \subseteq Q \times Q$ is the set of transitions.
- Var_E is a set of context variables, extracted from the graphical requirement. It contains all context variables which appear on the lifeline. All context variables are instances of M.
- M is a context metamodel, which is the common metamodel for all contexts defined in the requirement.

5.7.2.2 The Unwinding Algorithm

To define the unwinding algorithm some terms are expected to be known from the metamodel of the UML Sequence Diagram. A quick informal overview of the most important terms can be found below.

- *Interaction*: A list of interaction fragments with lifelines and messages. A requirement is one interaction.
- *Lifeline*: A line, which connects all events and constrains concerning one component.
- *Interaction fragment*: Anything which can appear on a lifeline. An InteractionFragment can cover one or more lifelines.
- *Combined fragment*: InteractionFragment, which has multiple nested interaction fragments (e.g., alt or loop).

The top level translation algorithm (Figure 37) creates an automaton from an interaction fragment by executing the methods defined later in this section.

The translation first initializes the automaton by creating the first (start) and the final states and initializing some structures. In the initialization, a transition is created between the initial state and the final state. This transition will be deleted later by the other methods. Afterwards it executes the unwinding algorithm (by calling the Unwind method, which is given as Algorithm 2 in Figure 38). After it is done, it calls the Postprocess method, which adds loop transitions to the automaton (Algorithm 8 in Figure 44).

The Unwinding algorithm has four inputs:

- an *f* interaction fragment with an *l* lifeline and an *F* list of interaction fragments,
- an *A* automaton to populate with new states and transitions,
- a *p* state in A, which is a temporal state for the interaction to unwind, and
- a *P* map which store the assignments between fragments and states.

Algorithm 1: Translate

input : i interaction fragment

output : $\mathcal{A}$ automaton

1 $\mathcal{A} \leftarrow$ empty automaton
2 $q_{start} \leftarrow$ new place
3 $q_{final} \leftarrow$ new place
4 add $\{q_{start}, q_{final}\}$ to $\mathcal{A}.Q$
5 $\mathcal{A}.q_0 \leftarrow q_{start}$
6 $\mathcal{A}.q_{inconclusive} \leftarrow$ new place
7 add $(q_{start}, \emptyset, q_{final})$ to $\mathcal{A}.T$
8 add q_{final} to $\mathcal{A}.F_a$
9 $\mathcal{P} \leftarrow$ empty map
10 Unwind(i, $\mathcal{A}$, q_{start}, $\mathcal{P}$)
11 Postprocess($\mathcal{A}$, $\mathcal{P}$)
12 return $\mathcal{A}$

Figure 37. The translation algorithm

The algorithm will recursively unwind the provided interaction and replace the temporal p state. As Figure 38 shows, the combined fragments are handled by the algorithm as well. The algorithm will create the Q set of states recursively by maintaining the U set of not yet unwound interaction fragments assigned to a temporal state in Q. Therefore U contains *(f, q)* pairs, where f is the fragment, which is unwound (it is either an interaction or a combined fragment) and q is the temporarily created state, which represents the f fragment at this moment.

The *handleCombined(k, q, A, P)* algorithm is a large switch-case structure, which executes one of five methods depending on the type of the *k* interaction fragment (see Algorithms 3 to 7, presented in Figure 39 to Figure 43).

After the unwinding algorithm is executed additional postprocessing steps are necessary, which add loop edges to the generated observer automaton. It is necessary, as if a message does not appear on the requirement at all, but is observed, it should be ignored. If the developer explicitly wants to denote that no message of a given kind was observed, then she/he should use the neg fragment. The postprocessing is done with the steps shown in Algorithm 8 (Figure 44).

Algorithm 2: Unwinding

input : f interaction fragment, $\mathcal{A}$ partial automaton, p temporal state, $\mathcal{P}$ fragment-state map
output : an accepting state

```
1  U ← empty set
2  q_currentTerminal ← p
3  t ← get outgoing transition from p in A.T
4  remove t from A.T
5  foreach k_i interaction fragment ∈ f ordered by location do
6      create q
7      add q to Q
8      store q for k_i in P
9      if k_i is cold then
10         add q_currentTerminal to A.F_i
11     if k is an Interaction or a Combined Fragment then
12         x ← (k, q)
13         add x to U
14     if k is a State Invariant then
15         l ← contextLabel(k)
16     else if k is a Message Occurrence then
17         l ← messageLabel(k)
18     else
19         l ← ∅
20     add (q_currentTerminal, l, q) to A.T
21     q_currentTerminal ← q
22 add (q_currentTerminal, label of t, target of t) to A.T
23 foreach (k, q) ∈ U do
24     handleCombined(k, q, A, P)
25 return q_currentTerminal
```

Figure 38. The unwinding algorithm

Algorithm 3: Alt fragment

input : k alt combined fragment, p temporal state, $\mathcal{A}$ partial automaton, $\mathcal{P}$ state–fragment map

```
1  t ← get outgoing transition from p in A.T
2  remove t from A.T
3  foreach o operand ∈ k do
4      i ← interaction nesting all interaction fragments from o
5      create q state
6      add q to A.Q
7      add (p, guard(o), q) to A.T
8      add (q, label of t, target of t) to A.T
9      Unwind(i, A, q)
```

Figure 39. Handling an *alt* fragment

Algorithm 4: Opt fragment

input : k opt combined fragment, p temporal state, $\mathcal{A}$ partial automaton, $\mathcal{P}$ state–fragment map

1 $t \leftarrow$ get outgoing transition from p in $\mathcal{A}.T$
2 remove t from $\mathcal{A}.T$
3 $f \leftarrow$ all interaction fragments from k
4 change all fragments $\in f$ to cold
5 $i \leftarrow$ interaction nesting all interaction fragments from f
6 add (q, label of t, target of t) to $\mathcal{A}.T$
7 Unwind(i, $\mathcal{A}$, p)

Figure 40. Handling an *opt* fragment

Algorithm 5: Loop fragment

input : k loop combined fragment, p temporal state, $\mathcal{A}$ partial automaton, $\mathcal{P}$ state–fragment map

1 $t \leftarrow$ get outgoing transition from p in $\mathcal{A}.T$
2 remove t from $\mathcal{A}.T$
3 $i \leftarrow$ interaction nesting all interaction fragments from k
4 add (p, $guard(k)$, p) to $\mathcal{A}.T$
5 $n \leftarrow$ succ(q, $\mathcal{A}.T$)
6 add (p, $negatedGuard(k)$, n) to $\mathcal{A}.T$
7 Unwind(i, $\mathcal{A}$, p)

Figure 41. Handling a *loop* fragment

Algorithm 6: Break fragment

input : k break combined fragment, p temporal state, $\mathcal{A}$ partial automaton, $\mathcal{M}$ state–fragment map

1 $t \leftarrow$ get outgoing transition from p in $\mathcal{A}.T$
2 remove t from $\mathcal{A}.T$
3 $i \leftarrow$ interaction nesting all interaction fragments from k
4 $q \leftarrow$ Unwind(i, $\mathcal{A}$, q)
5 add q to $\mathcal{A}.Q$

Figure 42. Handling a *break* fragment

Algorithm 7: Neg fragment

input : k neg combined fragment, p temporal state, $\mathcal{A}$ partial automaton, $\mathcal{M}$ state–fragment map

1 $t \leftarrow$ get outgoing transition from p in $\mathcal{A}.T$
2 remove t from $\mathcal{A}.T$
3 $f \leftarrow$ all interaction fragments from k
4 change all fragments except the last $\in f$ to cold
5 $i \leftarrow$ interaction nesting all interaction fragments from f
6 Unwind(i, $\mathcal{A}$, p)

Figure 43. Handling a *neg* fragment

Algorithm 8: Postprocessing

input : $\mathcal{A}$ automaton, $\mathcal{M}$ state–fragment map

```
1  appearing ← appearing message types in A
2  foreach q state ∈ {A.Q \ A.F_a} do
3      create m empty list of messages
4      add appearing to m
5      parent ← pair of q in M
6      while parent is nested do
7          if parent is a consider fragment then
8              add considered messages of parent to m
9          else if parent is an ignore fragment then
10             remove ignored messages of parent from m
11         parent ← parent fragment of parent
12     x ← message types appearing on outgoing transitions from q
13     remove x message types from m
14     guard ← message sent or received with a type ∉ m
15     add (q, guard, q) to A.T
16     if pair of q in M is cold then
17         add (q, negated(m.guard), A.q_inconclusive)
```

Figure 44. The postprocessing algorithm

5.7.2.3 Demonstrating the Mapping to Observer Automaton

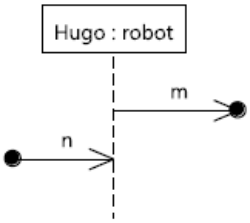
Figure 45 shows three example scenarios (requirements) defined with the scenario language (for the sake of simplicity, only the lifeline of a single component is presented). Both the informal meaning of these scenarios and the precise semantics with observer automata are given below.

- The (a) scenario describes that an m message should be sent and an n message should be received.
- The (b) scenario shows that either the robot is in ready state and sends an m message or not ready but receives an n message.
- The (c) scenario describes the following behaviour: if the robot receives a start message, there are two alternative cases. If it is ready, it should receive a doMagic message and then it should be in the c context. The second option is that the robot is not ready, therefore it sends a shutdown message. If the second case happens, this does not require being in the c context after sending the shutdown message (due to the break fragment).

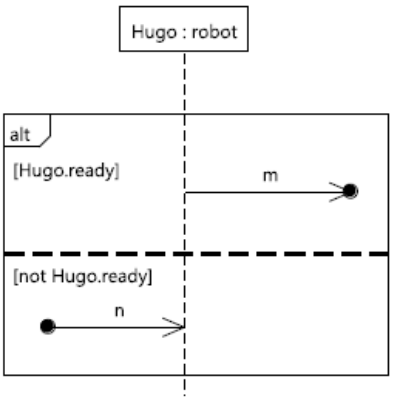
The result of the translation algorithm is presented in Figure 46. The visualization of the automata in the figure uses the following conventions:

- circles are states,
- directed edges between circles are transitions,
- circle with q_0 is the initial state,
- blue circle shows the inconclusively accepting state,
- green circles are accepting states,

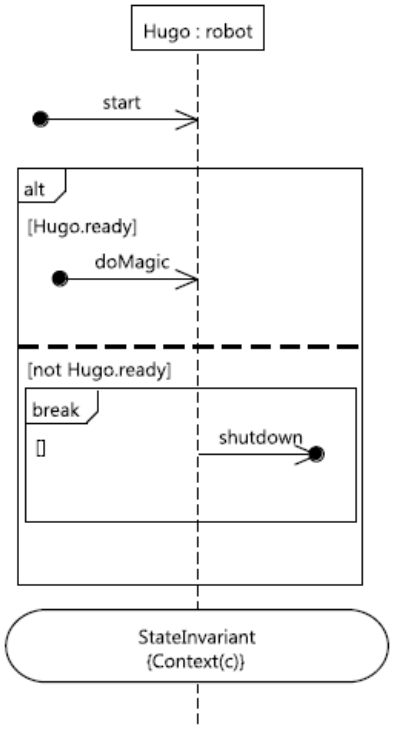
- message sending is denoted with postfix !, while receiving is with postfix ?,
- guard conditions are between [and],
- context change asserts are denoted with context(c) text, where c is the new context.



(a) The first scenario

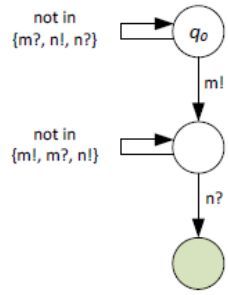


(b) The second scenario

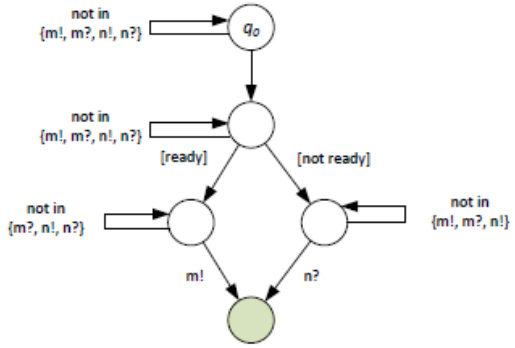


(c) The third scenario

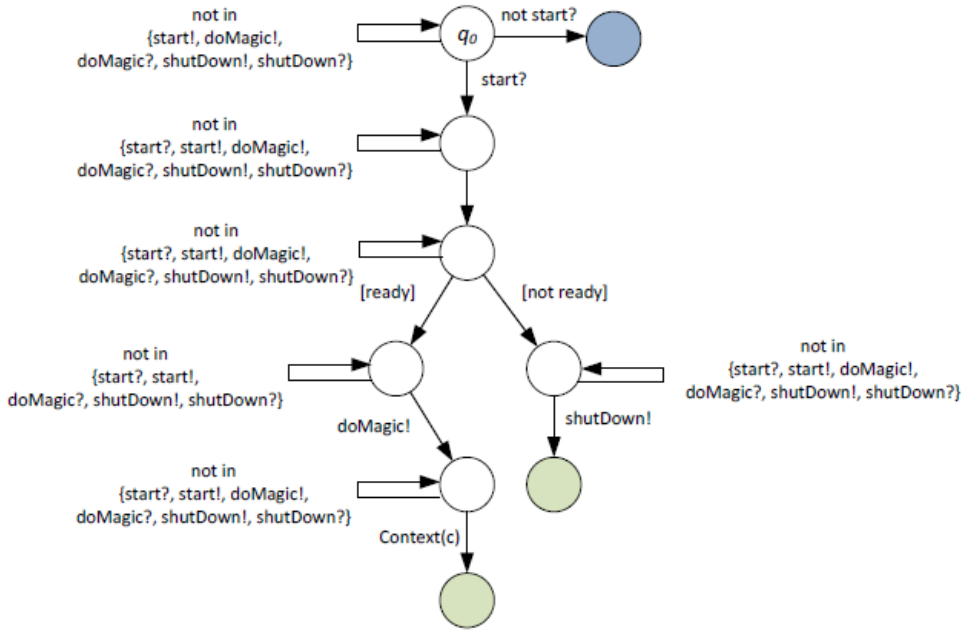
Figure 45. Example requirement scenarios



(a) Observer automaton for the first scenario



(b) Observer automaton for the second scenario



(c) Observer automata for the third scenario

Figure 46. Observer automata belonging to scenarios in Figure 45

5.7.3 Mapping to CaTL

This subsection gives a translation mechanism from the graphical scenario language to the temporal logic CaTL (Section 5.4). This way on-line verification of a component (considering the interaction fragments on its lifeline) can be provided by a monitor component that is synthesized on the basis of the CaTL description.

The translation is defined indirectly, on the basis of the observer automaton which was defined as the semantics of the graphical scenario.

To describe the translation, let us first introduce a function *succState(s):* $Q \rightarrow 2^Q$ that gives the succeeding state for a given state *s*, where Q is the set of states. This function will not only return all succeeding states but it will take into account how can those states be reached from s.

If no outgoing edges exist for the given state, then the function will return *true* if the state is accepting and *false* if it is not accepting. The inconclusively accepting state will return an inconclusive result which is denoted by <*true*>.

$$succState(s_1) = \begin{cases} true & \text{if } s_1 \in F_a \\ < true > & \text{if } s_1 \in \mathcal{A}.q_{inconclusive} \\ false & \text{if } s_1 \text{has no outgoing edges and } s_1 \notin F_a \\ \bigvee\limits_{\forall t = e(s_1)} succEdge(t) & \text{else (where } e(s) \text{ is the outgoing edges from } s_1) \end{cases}$$

The *succEdge(t)* function will define how an edge t is translated to CaTL expressions>

$$succEdge(t) = X\ (label(t) \wedge succState(t.target))$$

The *label(t)* function translates the guard conditions, message checking and context assertions to the CaTL-valid form, therefore the *label(t)* function takes a label of an edge and returns an atomic formula. For example, if the t input label has *a = 5* as guard condition and requires the component to be in the e_0 context (where c is a context variable), the label(t) function will return $a = 5 \wedge c \approx e_0$.

After these functions are defined, it is fairly simple to translate an A automaton to a CaTL formula.

$$\phi = succState(A, q_0)$$

Here follows three examples of the translation. The three graphical scenarios are shown in Figure 45 while the translated observer automata are found in Figure 46 (for the ease of reading, the loop edges generated by the postprocessing algorithm are omitted). Figure 47 presents the CaTL expressions belonging to the main lifeline (robot component) of these three scenarios.

$$\phi'_{(a)} = \mathcal{X}(m! \wedge \mathcal{X}(n? \wedge true))$$

(a) CaTL expression belonging to the first scenario

$$\phi'_{(b)} = \mathcal{X}(true \wedge ((\mathcal{X}(robot.ready \wedge \mathcal{X}(m! \wedge true))) \vee (\mathcal{X}(notrobot.ready \wedge \mathcal{X}(n? \wedge true)))))$$

(b) CaTL expression belonging to the second scenario

$$\phi'_{(c)} = \mathcal{X}((\text{not } start? \wedge < true >) \vee (start? \wedge \mathcal{X}(true \wedge (\mathcal{X}(robot.ready \wedge doMagic! \wedge \mathcal{X}(c \rightsquigarrow e \wedge true))) \vee (\text{not } robot.ready \wedge \mathcal{X}(shutDown! \wedge true)))))$$

(c) CaTL belonging to the third scenario

Figure 47. CaTL expressions belonging to scenarios in Figure 45

5.8 Describing Event Patterns

As formal specification languages like temporal logics or formal automata are often considered too low-level for the developers, a possible approach is the definition of easy-to-use requirement patterns. They combine the precise textual description with a graphical and/or formal representation (in a similar way like design patterns in OO architecture design).

In the following we

- identify the main patterns that are supported by our approach (as reported in [73], over 90% of the practical properties that were investigated could be expressed using these simple patterns),
- give the CaTL temporal logic based representation of these patterns (this is used to construct the complete CaTL based representation of the properties for monitor source code generation),
- propose an abstract syntax for a graphical pattern language (the concrete syntax can be elaborated in agreement with the domain specific tools of the demonstrators).

5.8.1 The Pattern Library

In the following basic patterns are identified, giving the natural language representation together with the temporal logic formalization. In the description “events” mean all input or output occurrences (i.e., elements of a monitored execution trace) that are relevant from the point of view of property monitoring: function call, function return, input or output signal, message received or sent, timer started or expired, state entered or left, predicate on a variable, context change, configuration change etc.

The property patterns are divided into two groups: occurrence patterns and ordering patterns (see below). The scopes of the patterns in an execution trace are illustrated in Figure 48.

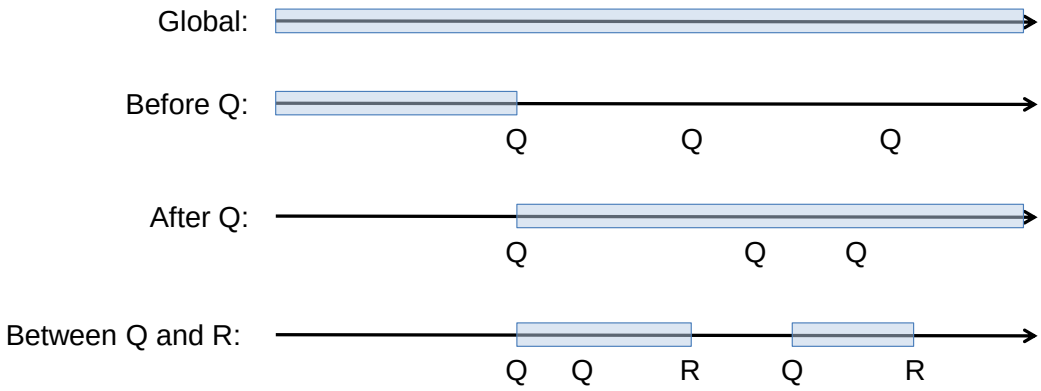


Figure 48. Scope of a pattern in a trace w.r.t. events Q and R

- *Occurrence patterns* describe the occurrence of a given event during execution. The following basic patterns are in this group:
  - *Universality* (also known as Always or Henceforth): It describes a (portion of) execution which contains only steps that are characterized with event P.

Property with scope	**Formalized property in CaTL**
Event P occurs in each step of the execution.	Globally P
Event P occurs in each step of the execution before event Q.	Eventually Q implies (P Until Q)
Event P occurs in each step of the execution after event Q.	Globally (Q implies Globally P)
Event P occurs in each step of the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (P Until R))

- *Absence* (also known as Never): It describes a (portion of) execution in which a certain event P does not occur.

Event P does not occur in the execution globally.	Globally (not P)
Event P does not occur in the execution before event Q.	Eventually Q implies (not P Until Q)
Event P does not occur in the execution after event Q.	Globally (Q implies Globally (not P))
Event P does not occur in the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (not P Until R))

- *Existence* (also known as Eventually)[7]: It describes a (portion of) execution that contains event P.

Event P occurs in the execution.	Eventually (P)
Event P occurs in the execution before event Q.	not Q Until* (P and not Q)
Event P occurs in the execution after event Q.	Globally (not Q) or Eventually (Q and Eventually P))
Event P occurs in the execution between events Q and R.	Globally (((Q and not R) and (Eventually R)) implies (not R Until* (P and not R)))

- *Bounded existence*: It describes a (portion of) execution in which an event occurs at most a specified number of times. Here the most typical case is considered when the specified number of times is 2 (where "2 times" means "twice").

Event P occurs at most 2 times in the execution.	(not P Until* (P Until* (not P Until* (P Until* Globally not P))))
Event P occurs at most 2 times in the execution before event Q.	Eventually Q implies ((not P and not Q) Until (Q or ((P and not Q) Until (Q or ((not P and not Q) Until (Q or ((P and not Q) Until (Q or (not P Until Q)))))))))
Event P occurs in the execution after event Q.	Eventually Q implies (not Q Until (Q and (not P Until* (P Until* (not P Until* (P Until* Globally not P))))))
Event P occurs in the execution between events Q and R.	Globally ((Q and Eventually R) implies ((not P and not R) Until (R or ((P and not R) Until (R or ((not P and not R) Until (R or ((P and not R) Until (R or (not P Until R)))))))))))

[7] In the pattern library, in order to formalize the often used natural language constructs, in some cases patterns and their negation are also considered (e.g., Absence and Existence).

- *Ordering patterns* describe the relative order in which multiple events occur during execution. The following basic patterns are in this group:
    - *Precedence*: It describes a pair of events where the occurrence of the first event is a necessary pre-condition for an occurrence of the second event (i.e., the occurrence of the second event is enabled by an occurrence of the first event). Note that a Precedence pattern allows causes to occur without subsequent effects.

Event S precedes P in the execution.	Eventually P implies (not P Until* S)
Event S precedes P in the execution before event Q.[8]	Eventually Q implies (not P Until (S or Q))
Event S precedes P in the execution after event Q.	Globally not Q or Eventually (Q and (not P Until* S))
Event S precedes P in the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (not P Until (S or R)))

    - *Response*: It describes a pair of events where an occurrence of the first event must be followed by, or happen together with an occurrence of the second event (i.e., there is a cause-effect relationship between the first and the second event). Also known as Follows or Leads-to. Note that a Response pattern allows effects to occur without causes (this way Precedence and Response patterns are not equivalent, response is just a “converse” of Precedence).

Event S responds to P in the execution.	Globally (P implies Eventually S)
Event S responds to P in the execution before event Q.	Eventually Q implies (P implies (not Q Until (S and not Q))) Until Q
Event S responds to P in the execution after event Q.	Globally (Q implies Globally (P implies Eventually S))
Event S responds to P in the execution between events Q and R.	Globally ((Q and not R and Eventually R) implies (P implies (not R Until (S and not R))) Until R)

    - *Chain precedence*: Chain patterns in general describe requirements related to combinations of event relationships. In case of chain precedence, a precedence relationship is described, consisting of (sequences of) individual events. First a 2 cause – 1 effect chain precedence relationship pattern is presented.

[8] Note that here „before event Q”, „after event Q”, and „between events Q and R” are scopes of the properties as defined at the beginning of this section

Events S followed by T precede P in the execution.[9]	Eventually P implies (not P Until (S and not P and Next (not P Until T)))
Events S followed by T precede P in the execution before event Q.	Eventually Q implies (not P Until (Q or (S and not P and Next (not P Until T))))
Events S followed by T precede P in the execution after event Q.	(Globally not Q) or (not Q Until (Q and Eventually P implies (not P Until (S and not P and Next (not P Until T))))
Events S followed by T precede P in the execution between events Q and R.	Globally ((Q and Eventually R) implies (not P Until (R or (S and not P and Next (not P Until T)))))

Second, a 1 cause – 2 effects chain precedence relationship pattern is presented.

Event P precedes S followed by T in the execution.	(Eventually (S and Next Eventually T)) implies ((not S) Until P))
Event P precedes S followed by T in the execution before event Q.	Eventually Q implies ((S and (not Q) and Next (not Q Until (T and not Q))) Until (Q or P))
Event P precedes S followed by T in the execution after event Q.	(Globally not Q) or ((not Q) Until (Q and ((Eventually (S and Next Eventually T)) implies ((not S) Until P)))
Event P precedes S followed by T in the execution between events Q and R.	Globally ((Q and Eventually R) implies ((not(S and (not R) and Next (not R Until (T and not R)))) Until (R or P)))

- *Chain response*: It describes a response relationship, consisting of (sequences of) individual events. First, a 2 stimuli – 1 response chain response relationship pattern is presented.

Event P responds to (S followed by T) in the execution.	(Eventually (S and Next Eventually T)) implies ((not S) Until P))
Event P responds to (S followed by T) in the execution before event Q.	Eventually Q implies ((not (S and (not Q) and Next (not Q Until (T and not Q)))) Until (Q or P))
Event P responds to (S followed by T) in the execution after event Q.	(Globally not Q) or ((not Q) Until (Q and ((Eventually (S and Next Eventually T)) implies ((not S) Until P)))
Event P responds to (S followed by T) in the execution between events Q and R.	Globally ((Q and Eventually R) implies ((not (S and (not R) and Next (not R Until (T and not R)))) Until (R or P)))

[9] In other words, it is not allowed that P occurs before S followed by T.

Second, a 1 stimulus - 2 responses chain is presented:

Events S followed by T respond to P in the execution.	Globally (P implies Eventually (S and Next Eventually T))
Events S followed by T respond to P in the execution before event Q.	Eventually Q implies (P implies (not Q Until (S and not Q and Next (not Q Until T)))) Until Q
Events S followed by T respond to P in the execution after event Q.	Globally (Q implies Globally (P implies (S and Next Eventually T)))
Events S followed by T respond to P in the execution between events Q and R.	Globally ((Q and Eventually R) implies (P implies (not R Until (S and not R and Next (not R Until T)))) Until R)

As it turns out, the CaTL expressions provide a precise description (that is needed for monitor source code generation), but their interpretation is difficult without some experience with temporal logics. The natural language description helps the designer to select the corresponding formalized property and understand its formalization. However, the natural language description is often less precise, for example, in case of the property “Event P occurs in each step of the execution before event Q”, the existence of Q is necessary for the satisfaction of this property, but this is not evident from the natural language description (one may consider that the property is satisfied when Q never occurs only a sequence of P).

An example of an application of a pattern is the following:

- Application specific property: For the Control component of a remotely controlled surveillance robot, receiving a StopCommand message from the Remote Operator guarantees that the StopAction signal will sent to the Motor component of the Robot.
- Pattern: Response with global scope: “Event S responds to P in the execution.”
- Instantiation of the pattern: S is the StopAction (signal), P is the StopCommand (message).
- CaTL formalization of the property: Globally (StopCommand implies Eventually StopAction)

The above listed patterns focused on the most frequently used temporal properties. These can be extended with timing (e.g., to capture the time between stimulus and response) using the timing extensions of the CaTL language.

5.8.2 Abstract Syntax for a Graphical Pattern Language

To describe and re-use patterns, we also propose a language (with its abstract syntax) that is inspired by [71]. It allows the developer to describe properties over the system or its components by using a combination of quantifiers, temporal patterns, and structural patterns on the domain model(s). Accordingly, the language consists of four parts.

- *Quantification of the formula* (Figure 49). Here *forAll* or *exists* quantifiers can be used together with the corresponding structural patterns (see below). Accordingly, the property must be satisfied for all, or for one (depending on the quantifier) matches of the structural pattern. Quantification patterns can be nested, or can contain a temporal pattern.

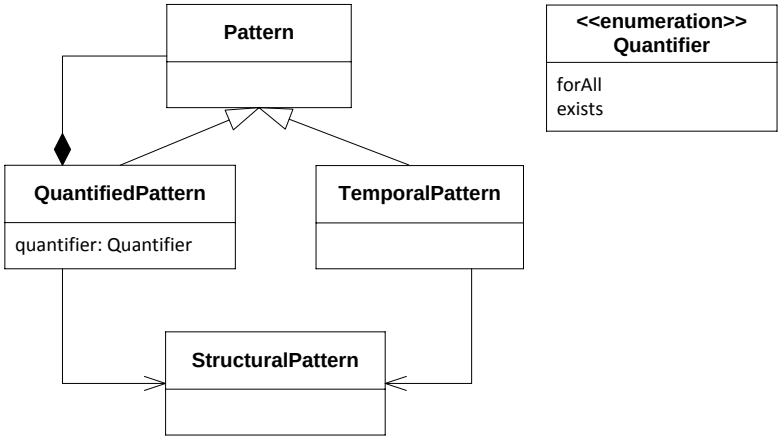


Figure 49. The quantification of the formula

- *Temporal patterns* (Figure 50). The temporal patterns consists of the typical occurrence patterns (absence, universality, existence, bounded existence) and ordering patterns (response, precedence, chain response, chain precedence) together with a scope (globally, before, after, between). As presented in Figure 49, these temporal patterns refer to structural patterns.

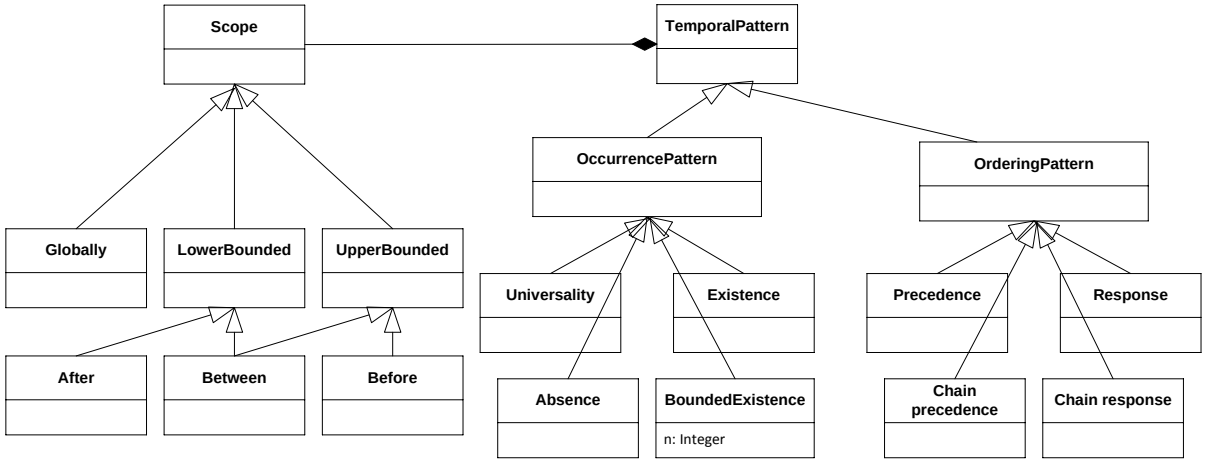


Figure 50. The temporal patterns

- *Structural patterns* (Figure 51) can be used in quantification or in a temporal pattern. A structural pattern means a query on a model (by a pattern matching algorithm). In quantification it returns all bound variables in found matches, while in case of a temporal pattern it returns true if at least one match is found or false when no match is found. The patterns presented in Figure 51 can be extended with additional language constructs if needed (the presented set of patterns is sufficient to represent the majority of practical properties).

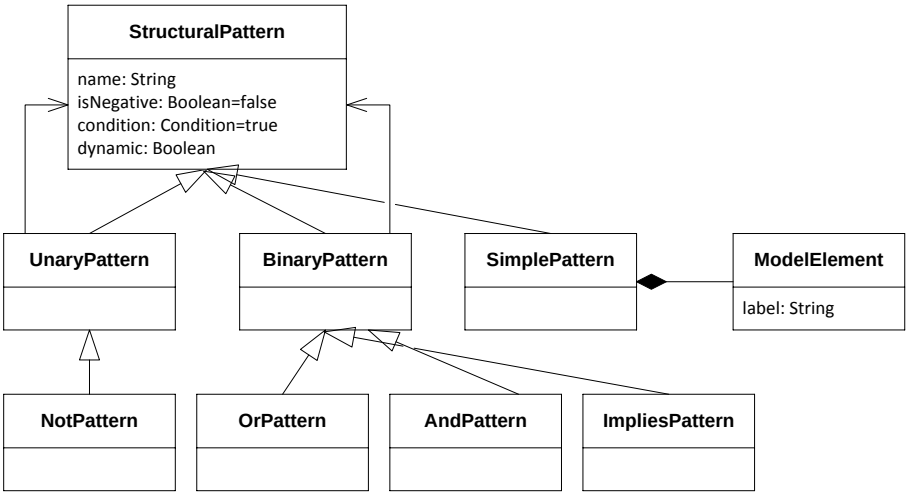


Figure 51. The structural pattern

- *Pattern elements* (Figure 52). A generic model element (ModelElement) serves as the superclass for pattern elements that are specific to the metamodel of the domain modelling language. In Figure 52 the root metamodel of statecharts is included together with specific model elements for events and actions. A straightforward extension is the inclusion of context fragments and configuration fragments as pattern elements, this way the context and configuration metamodel elements should be inserted. All classes are subclasses of ModelElement and have a label (for binding variables) and a condition.

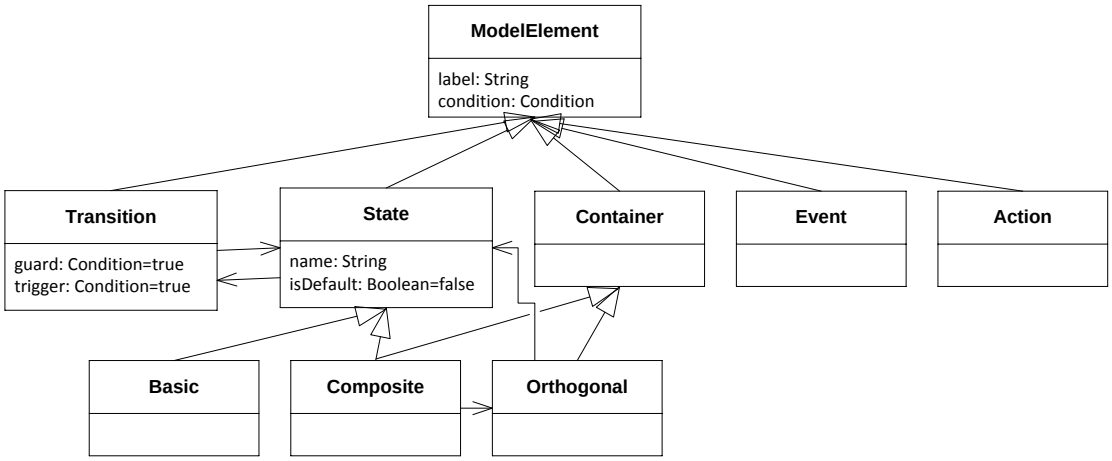


Figure 52. The pattern elements

The concrete syntax of this property language depends mainly on the concrete syntax of the domain model (context and configuration models) and the software artefacts (in this latter case the use of UML 2 model elements is a natural solution). In case of the quantifiers, temporal and structural operators, graphical as well as natural language representation can be used.

6 Conclusions

This deliverable aimed at the selection and definition of description languages that can be used for (1) capturing the properties to be checked by on-line verification and (2) describing the relation of components, properties and test cases for incremental testing. These languages allow the formalization of capabilities and restrictions, safety rules, function contracts, temporal or trace-based reference behaviour, as well as test coverage with respect to components and specified properties.

The main contributions are the following:

- Languages were selected to describe contexts (environment), scenarios (safety requirements), and configurations (hierarchical setup of skills, software components and hardware components). These are needed both for incremental testing and on-line verification as the target systems are context-aware and reconfigurable systems with safety-relevant behaviour.
- 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.
- An approach was presented to specify properties for on-line verification (monitoring) using graphical engineering languages that are automatically mapped to low-level formal languages. The engineering languages are easy to use and understand, while the low-level formal languages are precise and simple, being suitable for automated processing and source code generation for the monitor components.
    - On the basis of an analysis of the literature of runtime verification and property description approaches, the statecharts language and the scenario language were selected as graphical engineering languages.
    - A new temporal logic variant, the Context-aware Timed Propositional Linear Temporal Logic (CaTL) was defined as a low-level formal language to capture context-dependent timed properties that can be checked by trace-based monitoring.
    - A powerful constraint and system specification language was defined as a low-level formal language to capture reference behaviour (reference automaton) that can be checked by state-based monitoring.

These languages form the basis of incremental test selection and monitoring. In case of incremental testing, the description of the relation of test cases, system components and properties is used by the *methods and tools for selecting, adapting and extending test cases from existing test suites* in an incremental way, in order to check the changed components or properties. In case of runtime monitoring, the description of the properties is used by the *methods and tools for monitor synthesis*, i.e., an automated construction of software monitors to check the specified system properties. These developments will be the topic of later tasks and deliverables (D34.20 “Incremental testing of behaviour”, and D34.31/32 “Design of the monitoring infrastructure”). The use of these languages will be evaluated in D34.50 “Assessment of the on-line verification and incremental testing” at the end of the project.

7 References

[1] Yoo, S. and Harman, M.: Regression testing minimization, selection and prioritization: a survey. In: Software Testing, Verification and Reliability 22.2 (2012), pp. 67–120. doi: 10.1002/stvr.430.

[2] Engström, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques, Information and Software Technology, Volume 52, Issue 1, January 2010, Pages 14-30, doi: 10.1016/j.infsof.2009.07.001.

[3] Institute of Electrical and Electronics Engineers.: Systems and software engineering – Vocabulary. Standard 24765:2010. 2010, pp.1–418. doi: 10.1109/IEEESTD.2010.5733835.

[4] Rothermel, G. and Harrold, M.J.: Analyzing Regression Test Selection Techniques, IEEE Trans. Software Eng., vol. 22, no. 8, pp. 529-551, Aug. 1996.

[5] Rothermel, G., Untch, R. H., Chu, C., Harrold, M.J.: Prioritizing Test Cases For Regression Testing, IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929-948, October, 2001.

[6] Tsai, J., Fang, K., Chen, H., and Bi, Y.: A noninterference monitoring and replay mechanism for real-time software testing and debugging. IEEE Transactions on Software Engineering, 16(8):897–916, 1990.

[7] Chodrow, S., Jahanian, F., Donner, M.: Runtime monitoring of real-time systems. In IEEE Real-Time Systems Symposium, pages 74–83, 1991.

[8] Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In Tools and Algorithms for Construction and Analysis of Systems, pages 342–356, 2002.

[9] Arafat, O., Bauer, A., Leucker, M., Schallhart, C.: Runtime verification revisited. Technical Report TUM-I05, Technical University of Munich, October 2005.

[10] Sankar, S., Mandal, M.: Concurrent runtime monitoring of formally specified programs. IEEE Computer, 26(3):32–41, 1993.

[11] Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In RTSS'08: Proceedings of the 29th IEEE Real-Time System Symposium, pages 481–491, 2008.

[12] Bhargavan, K., Chandra, S., McCann, P., Gunter, C.A.: What packets may come: Automata for network monitoring. SIGPLAN Notices, 35(3):209–219, 2001.

[13] Bhargavan, K., Gunter, C. A.: Requirement for a practical network event recognition language. Electronic Notes in Theoretical Computer Science, 70(4):1–20, 2002.

[14] Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In 11th Euromicro Conference on Real-Time Systems, pages 114–122, 1999.

[15] Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In International Conference on Parallel and Distributed Processing Techniques and Applications, pages 279–287, 1999.

[16] Bhargavan, K., Gunter, C. A., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal analysis of network simulations. IEEE Transactions on Software Engineering, 28(2):129–145, 2002.

[17] Kim, M., Lee, I., Sammapun, U., Shin, J., Sokolsky, O.: Monitoring, checking, and steering of real-time systems. Proceedings of 2nd International Conference on Runtime Verification, Electronic Notes in Theoretical Computer Science, 70(4), 2002.

[18] Sokolsky, O., Sammapun, U., Lee, I., and Kim, J.: Run-time checking of dynamic properties. Proceedings of 5th International Conference on Runtime Verification, Electronic Notes in Theoretical Computer Science, 144(4):91–108, 2005.

[19] Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In ICSE'04: Proceedings of 6th International Conference on Software Engineering, pages 418–427, 2004.

[20] Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In Proceedings of the 2006 Australian Software Engineering Conference (ASWEC), Sydney, Australia, April 2006. IEEE Computer Society.

[21] Meredith, P., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. Software Tools for Technology Transfer, Special Section on Runtime Verification (2011)

[22] Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04). LNCS, vol. 2937, pp. 44–57, January 2004

[23] Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for runtime monitoring: From Eagle to RuleR. In: Proceedings of the 7th Workshop on Runtime Verification (RV'07). LNCS, vol. 4839, pp. 111–125, March 2007

[24] Havelund, K., Rosu, G.: Monitoring Java programs with JavaPathExplorer. In: Proceedings of the 1st Workshop on Runtime Verification. Electronic Notes in Theoretical Computer Science, vol. 55, Elsevier Publishing (2001)

[25] Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan,M.: Java-MaC: a run-time assurance approach for Java programs. Formal Methods Syst. Des. 24(2), 129–155 (2004)

[26] Bodden, E.: A lightweight LTL runtime verification tool for Java. In: 9th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), pp. 306–307, October 2004

[27] Havelund, K.. Runtime verification of C programs. In Proc. of TestCom/FATES, Lecture Notes in Computer Science 5047, Springer-Verlag, 2008

[28] Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs (1988)

[29] Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04). LNCS, vol. 3362, March 2004

[30] Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10), pp. 2103–2110 (2010)

[31] Barringer, H., Havelund, K.: TraceContract: a Scala DSL for trace analysis. In: Proceedings of 17th International Symposium on Formal Methods(FM'11). LNCS, vol. 6664, June 2011

[32] Burdy, L., Cheon, Y., Cok, D., Ernst,M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. 7(3), 212–232 (2005)

[33] Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass—Java with assertions. In: Proceedings of the 1st Workshop on Runtime Verification (RV'01), July 2001

[34] Bodden, E.: A lightweight LTL runtime verification tool for Java. In: 9th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), pp. 306–307, October 2004

[35] D'Amorim, M., Havelund, K.: Event-based runtime verification of Java programs. In: Workshop on Dynamic Program Analysis (WODA'05). ACM Sigsoft Software Engineering Notes, vol. 30, pp. 1–7 (2005)

[36] Allan, C., Avgustinov, P., Kuzins, S., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J., Christensen, A.S., Hendren, L., Lhoták, O.: Adding trace matching with free variables to AspectJ. In: Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05), pp. 345–364, October 2005

[37] Colombo, C., Pace, G., Abela, P.: Compensation-aware runtime monitoring. In: Proceedings of the 1st International Conference on Runtime Verification (RV'10). LNCS, vol. 6418, pp. 214–228, November 2010

[38] Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K., Smolka, S.A., Stoller, S.D., Zadok, E.: Aspect-oriented instrumentation with GCC. In: Proceedings of the 1st International Conference on Runtime Verification. LNCS, vol. 6418, pp. 405–420, November 2010

[39] Bodden E., Hendren, L.: The Clara framework for hybrid typestate analysis. Software Tools for Technology Transfer, Special Section on Runtime Verification, in this volume (2011)

[40] Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8(3):231{274, 1987.

[41] ISO/IEC 19505-2:2012 information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 2: Superstructure, 2012. http://www.omg.org/spec/UML/ISO/19505-2/PDF

[42] Latella, D., Majzik, I, and Massink, M.: Towards a formal operational semantics of UML statechart diagrams. In Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), pages 465{481. Kluwer, B.V., 1999.

[43] Plat, N and Larsen, P. G.: An overview of the ISO/VDM-SL standard. ACM SIGPLAN Notices, 27(8):76{82, 1992.

[44] Almeida, J. B., Frade, M. J., Pinto, J. S. and de Sousa, S, M.: An overview of formal methods tools and techniques. In Rigorous Software Development, Undergraduate Topics in Computer Science, pp 15-44. Springer London, 2011.

[45] Larsen, P. G., Lausdahl, K., Fitzgerald, J. and Wolff, S.: VDM-10 Language Manual, Ouverture Technical Report Series, TR-001, Ouverture, 2011.

[46] ISO/IEC 13568:2002 standard, 2002.

[47] Amalio, N., Polack, F. and Stepney, S.: UML + Z: UML augmented with Z. In Marc Frappier and Henri Habrias, editors, Software Specification Methods - An Overview Using a Case Study, new edition. Hermes Science Publishing, 2006.

[48] Thompson, J. M., Heimdahl, M. P. E. and Miller, S.P.: Specification-based prototyping for embedded systems. SIGSOFT Softw. Eng. Notes, 24(6):163-179, 1999.

[49] Heimdahl, M. P. E., Whalen, M. W.: Reduction and slicing of hierarchical state machines. In Mehdi Jazayeri and Helmut Schauer, editors, Software Engineering { ESEC/FSE'97, volume 1301 of Lecture Notes in Computer Science, pp 450-467. Springer, 1997.

[50] Choi, Y. and Heimdahl, M. P. E.: Model checking RSML^{-e} requirements. In Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering (HASE), pages 109-118. IEEE Computer Society, 2002.

[51] Leveson, N. G., Heimdahl, M. P. E. and Reese, J. D.: Designing specification languages for process control systems: Lessons learned and steps to the future. In Oscar Nierstrasz and Michel Lemoine, editors, Software Engineering ESEC/FSE 99, volume 1687 of Lecture Notes in Computer Science, pages 127-146. Springer, 1999.

[52] Braberman, V., Kicillof, N. and Olivero, A.: A scenario matching approach to the description and model checking of real-time properties. IEEE Transactions on Software Engineering, 31(12):1028-1041, 2005.

[53] Dillon, L. K., Kutty, G., Moser, L. E., Melliar-Smith, P. M., and Ramakrishna, Y. S.: A graphical interval logic for specifying concurrent systems. ACM Trans. Softw. Eng. Methodol., 3(2):131-165, 1994.

[54] Foster, H., Maschner, E. and Wolfsthal, Y.: IEEE 1850 PSL: The next generation. In Proceedings of Design and Verification Conference and Exhibition (DVCON), 2005.

[55] IEEE 1850-2010 - IEEE standard for Property Specification Language (PSL), 2010.

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

[57] ITU-T. ITU-T Z.120 recommendation: Formal description techniques (FDT) - Message sequence chart (MSC), 2011.

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

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

[60] Ziemann, P. and Gogolla, M.: OCL extended with temporal logic. In Manfred Broy and Alexandre V. Zamulin, editors, Perspectives of System Informatics, volume 2890 of Lecture Notes in Computer Science, pp 351-357. Springer, 2003.

[61] Meyers, B., Wimmer, M., Vangheluwe, H. and Denil, J.: Towards domain-specific property languages: The ProMoBox approach. In Proceedings of the 2013 ACM Workshop on Domain-specific Modeling, DSM '13, pages 39-44. ACM, 2013.

[62] Pintér, G.: Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems. PhD thesis, Budapest University of Technology and Economics, 2007.

[63] Mills, C.: Using Design by Contract in C. O'Reilly ONLamp.com, October 28, 2004. http://www.onlamp.com/lpt/a/5288

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

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

[66] Harel, E., Lichtenstein, O., and Pnueli, A.: Explicit Clock Temporal Logic. Logic in Computer Science, 1990.

[67] Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems, 2(4):255–299, 1990.

[68] Alur, R., and Henzinger, T. A.: A Really Temporal Logic. Journal of the ACM (JACM), (July), 1994.

[69] 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.

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

[71] Meyers, B., Wimmer, M., Vangheluwe, H., and Denil, J.: Towards Domain-Specific Property Languages: The ProMoBox Approach. In Proc. International Dependency and Structure Modelling Conference (DSM 13), Indianapolis, USA, pp 39-44, 2013.

[72] 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.

[73] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C.: Patterns in Property Specifications for Finite-State Verification. In Proc. International Conference on Software Engineering (ICSE 1999), pp 411-420, 1999.

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

[75] Bitsch, F. Safety Patterns - The Key to Formal Specification of safety requirements. In Proceedings of the 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP), pp 176-189. Springer-Verlag, 2001.

[76] Campos, J. C., Machado, J., and Seabra, E.: Property Patterns for the Formal Verification of Automated Production Systems. In Proceedings of the 17th IFAC World Congress, pp 5107-5112. IFAC, 2008.

[77] Campos, J. C., Machado, J.: Specification Patterns System for Discrete Event Systems Analysis. International Journal of Advanced Robotic Systems, 10(315), 2013.

[78] Preusse, S., and Hanisch, H.-M.: Specification of technical plant behavior with a safety-oriented technical language. In Proceedings of the 7th IEEE International Conference on Industrial Informatics (INDIN), pp 632-637. IEEE, 2009.

[79] Meolic, R., Kapus, T., and Brezocnik, Z.: CTL and ACTL patterns. In Proceedings of the International Conference on Trends in Communications (EUROCON), 2001.

[80] Holt, A., Klein, E.: A Semantically-derived Subset of English for Hardware Verification. In Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics (ACL), pp 451-456. Association for Computational Linguistics, 1999.

[81] Kuhn, T.: A Survey and Classification of Controlled Natural Languages. Computational Linguistics, 40(1):121-170, 2014.

[82] Sommerville, I.: Formal specification. In Software Engineering 9, chapter 27. Pearson Education, 2011.

[83] Knight, J. C., DeJong, C. L., Gibble, M. S., and Nakano, L. G.: Why are formal methods not used more widely? In C. Michael Holloway and Hayhurst Kelly J, editors, Fourth NASA Langley Formal Methods Workshop (LFM), pages 1-12, 1997.

[84] DeJong, C. L., Gibble, M. S., Knight, J. C., and Nakano, L. G..: Formal specification: A systematic evaluation. Computer Science Report CS-97-09, University of Virginia, 1997. http://www.cs.virginia.edu/~techrep/CS-97-09.ps.Z

[85] Sohn, S., and Seong, P. H.: A comparative study of formal methods for safety critical software in nuclear power plant. Journal of the Korean Nuclear Society, 32:537-548, 2000.

[86] Nobe, C. R., and Warner, W.E.: Lessons learned from a trial application of requirements modeling using statecharts. In Proceedings of the Second International Conference on Requirements Engineering, pp 86-93, 1996.

[87] Symbolic Analysis Laboratory. http://sal.csl.sri.com/ (accessed on January 6, 2015)

[88] Bertrand, M.: Object-Oriented Software Construction (2nd edition). Prentice-Hall, New York, 1997.

[89] Dahlweid, M., Moskal, M., Santen, T., Tobies, S., and Schulte, W.: VCC: Contract-based Modular Verification of Concurrent C. In 31st International Conference on Software Engineering, ICSE 2009, IEEE Computer Society, 2008.

[90] Harel, D. and Naamad, A.: The STATEMATE Semantics of Statecharts. In: ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 4, October 1996, pp 293-333, 1996.