R5-COP

Reconfigurable ROS-based Resilient Reasoning Robotic Cooperating Systems

Design of the monitoring infrastructure (final version)

BME

Project	R5-COP	**Grant agreement no.**	621447
Deliverable	D34.32	**Date**	31/07/2016
Contact Person	Istvan Majzik	**Organisation**	BME
E-Mail	majzik@mit.bme.hu	**Diss. Level**	PU

Document History			
Ver.	**Date**	**Changes**	**Author**
0.1	15/12/2015	Initial structure of the content	I. Majzik (BME)
0.2	05/01/2016	Integrating the overview of the monitoring infrastructure	I. Majzik (BME)
0.3	11/01/2016	Integrating the chapter on monitoring on the basis of temporal specification	I. Majzik, G. Horányi and other contributors (BME)
0.4	18/01/2016	Extending the document with introduction and conclusions.	I. Majzik (BME)
0.5	20/01/2016	Integrating the chapter on monitoring on the basis of statechart specifications	A. Vörös, T. Tóth, V. Molnár and other contributors (BME)
0.6	22/01/2016	Integrating the chapter in monitoring on the basis of scenario specifications	A. Vörös, T. Tóth, Z. Micskei and other contributors (BME)
0.65	25/01/2016	Modifications after discussions	All (BME)
0.7	25/01/2016	Document is ready for internal review	All (BME)
0.7	05/02/2016	Internal review by FAU	Matthias Meitner (FAU)
0.7	05/02/2016	Internal review by IMCS	Artis Gaujens, Janis Bicevskis (IMCS)
0.8	06/02/2016	Corrections after internal review	I. Majzik (BME)
1.0	09/02/2016	Ready for submission	I. Majzik (BME)
1.1	13/07/2016	D34.32: Adding new sections about the monitor interfaces and the use of tools	A. Vörös, I. Majzik and other contributors (BME)
1.2	19/07/2016	Corrections and additions integrated	A. Vörös, I. Majzik
1.3	31/07/2016	Corrections after the internal review by FAU and IMCS. Ready for submission.	Matthias Meitner (FAU), Artis Gaujens (IMCS), Istvan Majzik (BME)

Note: Filename should be

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

Fields are defined as follow

1. Deliverable number ***.***

2. Revision number:

draft version **v**

approved **a**

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

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

Content

List of Figures

List of tables

List of Acronyms

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

1 Introduction

1.1 Summary (abstract)

WP34 of R5-COP aims at supporting the off-line and on-line verification of the behaviour of R5-COP systems by elaborating methods and tools for incremental testing and runtime monitoring. Runtime monitoring focuses on checking the effects of runtime errors (due to random hardware faults, configuration faults, operator faults, faults in adaptation and self-healing), this way also supervising error handling and self-healing policies.

Deliverable D34.10 described the languages that are proposed for the specification of the properties to be monitored. In this deliverable the monitoring infrastructure is presented. This infrastructure consists of the tools that support the synthesis of the monitor components on the basis of the specified properties. According to the languages and formalisms used by the designers to specify these properties, three tool-chains are developed: the first is based on a state machine (statechart) specification, the second on scenario (sequence diagram) specification, while the third applies temporal properties.

The first version of this deliverable (D34.31) focused on the algorithms that are implemented by the tool-chains, namely the algorithms used for the internal processing of the input properties and the synthesis algorithms that result in low-level representations that are the basis for source code generation. The input languages were extended with the representation of time, context and configuration dependency, this way the processing of time, context and configuration related information is also described.

The final version of this deliverable (D34.32) includes additionally the description of the concrete interfaces of the monitor (Section 6), as well as demonstrates the use of the monitor synthesis tools that can generate monitor source code from high-level property specifications (Section 7).

The evaluation of the monitoring infrastructure will be provided in deliverable D34.50 (Assessment of on-line verification and incremental testing) in M36.

1.2 Purpose of document

This deliverable aims at the description of the monitor synthesis (i.e., the tool-chains and their internal algorithms that support the synthesis of monitor components) and the concrete interfacing of the generated monitor nodes. This is the result of *Task 34.3: Design of the monitoring infrastructure.* The algorithms designed in this task are responsible for processing the input properties (i.e., statechart, scenario, and temporal behaviour specifications extended with time, context and configuration dependency) and generating low-level representations for source code synthesis and the source code itself.

The tool-chains will be used and the generated monitor components will be applied in real demonstrators (motivating examples are also described in this deliverable). The application of monitoring will be evaluated in *Task 34.5* and reported in D34.50.

1.3 Partners involved

Partners and Contribution	
Short Name	**Contribution**
BME	Design of the monitoring infrastructure
FAU	Review of the document
IMCS	Review of the document

2 Overview of the monitoring infrastructure

In WP34, a monitoring infrastructure (method and tool support) is developed that allows automated construction of monitor components by the synthesis of their source code. These monitors perform online verification by observing the behaviour of the robot components (i.e., the trace of their states, events, actions, and the perceived context) to detect the hazardous situations and trigger a reaction (e.g., to stop the robot to maintain safety). The potential hazardous situations (e.g., the sequence of events and interactions among components) are specified using a high-level language: state machine diagram, sequence diagram, or temporal patterns. The tool generates the source code on the basis of this specification automatically. The novelty of the approach is that the monitoring infrastructure is suitable for local monitoring of robot control components that are characterized by context-aware, configuration-dependent,, real-time, event-driven behaviour.

The monitors are implemented as software components (ROS nodes) that can be interfaced with the observed components through the applied communication middleware (ROS topics) or using source code instrumentation.

In order to apply this kind of online verification, the following inputs are needed (Figure 1):

- The description of the monitored properties using behaviour specification, scenario specification, or temporal specification.
- Definition of the events or actions that shall be observed on interfaces or topics.

The output of the monitor synthesis is the source code of the core logic of the monitor component that performs the matching of events, context and configuration conditions and decides on the allowed behaviour. This core logic can be interfaced with the concrete implementation of the monitored component.

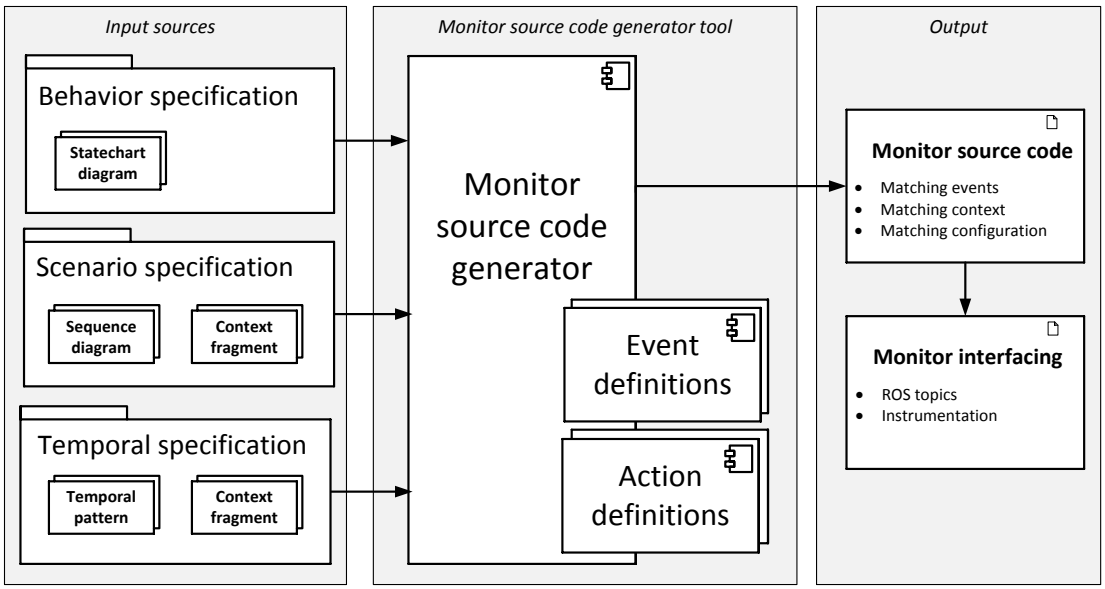

Figure 1. Support of runtime verification using the monitor source code generator tool

Note that the monitors are useful not only in runtime but also in the testing phase as part of the test oracle that decides whether the behaviour is acceptable during the execution of a given test suite.

The following subsections present the specification languages (subsection 2.1), the proposed solution for interfacing (subsection 2.2) and the overview of the monitor synthesis tool-chains (subsection 2.3).

2.1 Languages to express properties

In this section we summarize the use of specification languages that we introduced in deliverable D34.10 “Languages and formalisms for expressing properties for on-line and off-line verification”.

Languages are defined at two levels (Figure 2). Widely used engineering languages are adapted and extended to be used by developers to specify properties (behaviour, scenario, or temporal properties) to be monitored. These specifications are mapped to internal formal languages that can be used for monitor synthesis.

Here we summarize the main properties of the engineering languages, while the mapping to internal languages is discussed in the relevant sections later in this deliverable.

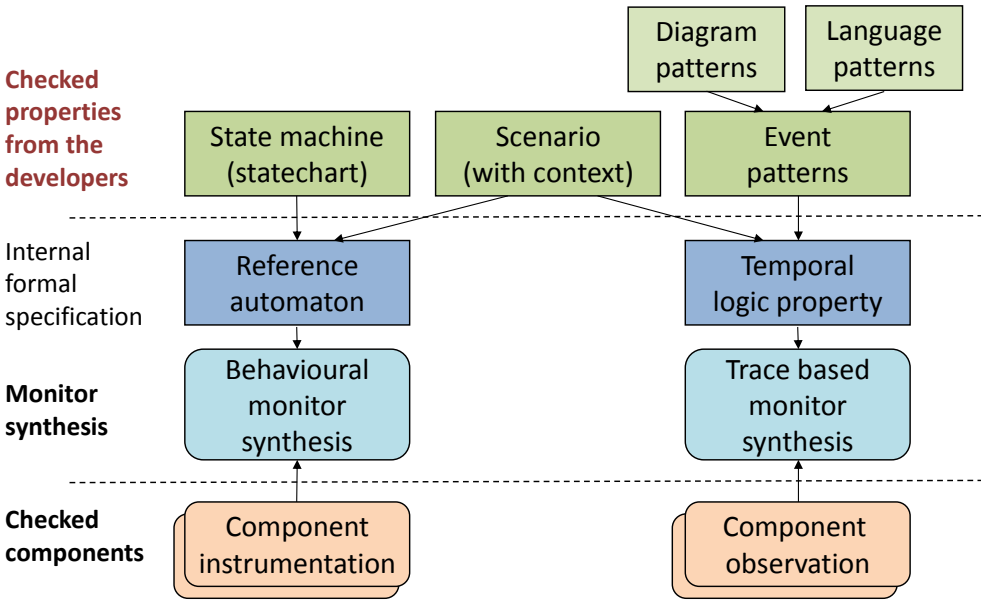

Figure 2. Languages used to describe properties to be monitored

2.1.1 High-level engineering languages

Developers may use the following languages to specify the properties to be monitored:

- Behaviour specification: In this case *UML 2 statechart diagrams* can be used. These can capture the allowed behaviour of the monitored system in form of sequences of events (input events as messages or signals; and output events as messages or actions). The runtime sequence of events (as observed by the monitor) has to match the ordering given in the statechart otherwise an error is detected. The following constructs can be used in the statechart:
  - State hierarchy and concurrent regions are allowed.
  - It is possible to use simple (integer and Boolean) variables as well as actions and guard conditions using these variables. Note that these variables are “specification variables”, i.e., these are used only to capture the behaviour (e.g., to limit the number of retrying an action) and do not represent observed variables of the monitored system.
  - Timeout can be specified using the “after” keyword and a time parameter: if no events are observed within this time period then the related action will occur (typically, a state change or action for error handling).

- Outputs can be specified using the “raise” keyword. The keyword “always” indicates an unconditional state transition.

For specifying the statechart diagrams, the open source Yakindu tool[1] is supported.

- Scenario specification: In this case *UML 2 sequence diagrams* can be used. These can express sequential, alternative or parallel composition of messages. They are suitable for describing for example command sequences. These sequence diagrams can be used to specify “if-then” style requirements. The first part of the diagram is a “condition” part (that shall be matched to trigger the checking) while the second part is an “assert” part (if it is not observed after the condition part then the monitor detects an error). This way it is possible to describe for example what outputs of the monitored component of a robot should be observed when receiving some input messages from others.
  - Everything before the “assert” box is the condition (trigger) part of the requirement. The assert box should only be checked, if the behaviour in the trigger part is observed.
  - We are only concerned with verifying a trace of messages, thus there is only one entity (lifeline) in the requirement. The exact sender of the messages or the exact recipients of the messages are not specified (as messages are observed in ROS topics).
  - The diagram captures the sequence of messages that shall be observed by the monitor; if the messages represented in the diagram are observed in other order then an error is detected. Other messages (that are not included in the diagram and thus not observed by the monitor) may occur in an interleaved way.
  - Content and configuration dependency can be used by referring to *context fragments* or *configuration fragments* in the scenario.

  For specifying the sequence diagrams, the open source Papyrus editor[2] is supported.
- Temporal specification: In this case *temporal property patterns* can be used to compose the property to be monitored.

  As there is no external tool that supports the composition of patterns using the specific temporal logic that we defined in D34.10, we implemented a tool for the Eclipse environment. This tool in described in Section 5.1.

2.1.2 Examples motivated by demonstrator use cases

In the following a reference statechart diagram is presented (Figure 3) which describes the allowed behaviour in case of the “Battery charging at the docking station” scenario from WP44 “Flexible reconfigurable mobile logistics robot” use case “R5COP-WP44-UC-1 Laundry handling in hospitals” (MIR MIR-100).

The runtime sequence of events observed by the monitor shall match this allowed behaviour otherwise the monitor detects an error. The observed events (e.g., derived from messages in ROS topics to which the monitor is subscribed) are the following:

- ShouldDock: The robot should dock for charging its battery.
- PathExist: Path is successfully planned to reach the docking station.
- NoWay: There is no path that can be planned to reach the docking station.

[1] The Yakindu tool can be downloaded from http://statecharts.org/

[2] The Papyrus tool can be downloaded from https://eclipse.org/papyrus/ (the latest 1.1.X version from Eclipse Mars is supported).

- Docked: Docking is successful.
- NotDocked: Docking is not successful.
- Charged: Charging was successful.
- ChargingFails: Charging is not successful.

The parameters of the statechart model are the following:

- PlanningTO: Timeout for performing the planning
- DockingTO: Timeout for docking
- ChargingTO: Timeout for successful charging
- MaxTryP: The planning can be retried at most MaxTryP times
- MaxTryD: The docking can be retried at most MaxTryD times

The specification variable:

- c: It is used to count the number of retries in case of planning and docking.

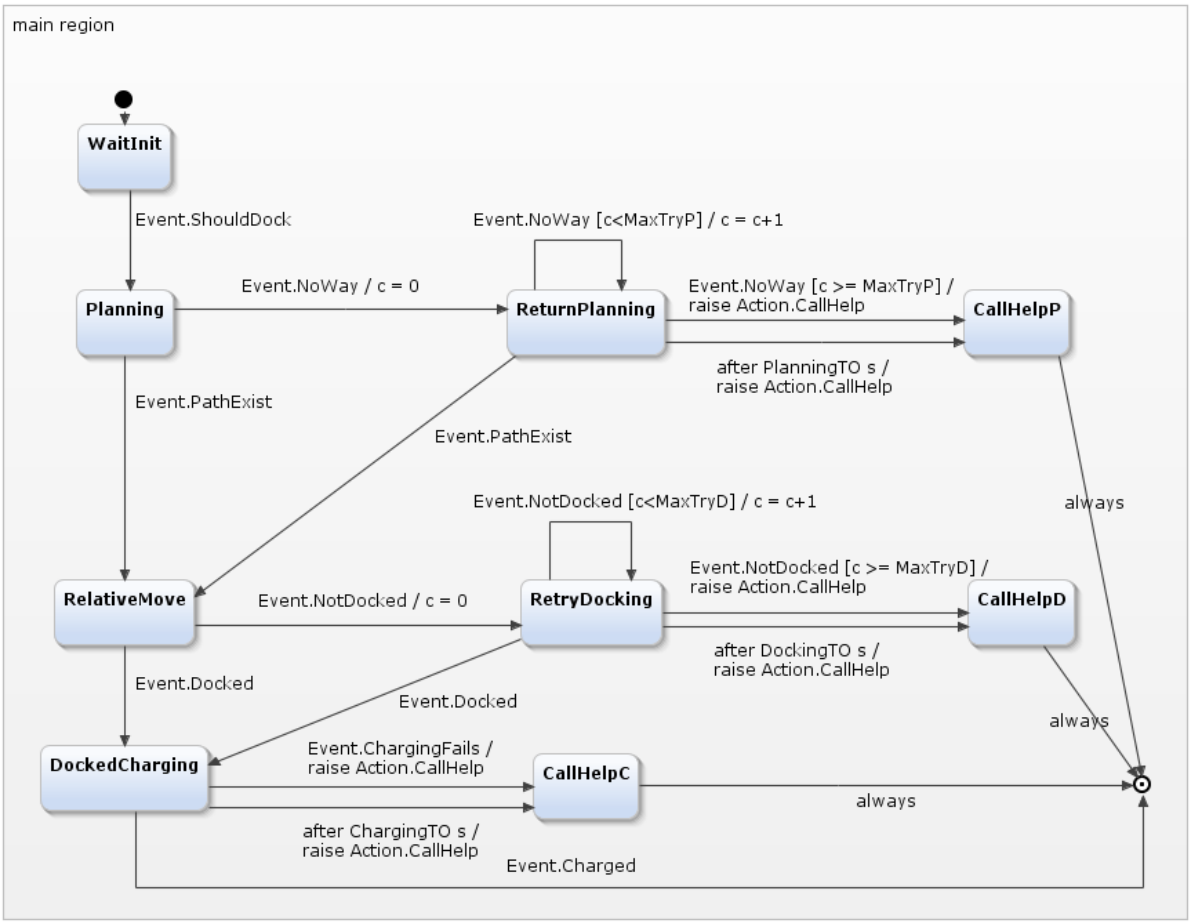

Figure 3. Reference statechart model for monitoring

In case of demonstrator WP42 “Professional service robot”, use case “R5COP-WP42-UC-1 Outdoor tele-robotic security” (PIAP Scout), rules (reference behaviour) to be monitored can be defined as scenarios or temporal patterns as follows:

- Sequence of commands: Validate if other commands are not sent prior to establishing a connection.

- Context dependency: In autonomous mode, if distance from obstacle in front is less than X, reduce speed to Y.
- Configuration and status dependency: Validate consistency of communicated commands with robot capabilities. When giving navigation orders, check if battery level is sufficient to proceed.
- Timing (time-out): Validate if the robot-side processing from receiving command to notifying of status change is shorter than T. Validate if frequency of sensor data (such as laser scanner) is higher than 1/T.
- Data change: Check if there is an unexpected jump in the robot position.

In the following we exemplify (in a generic way) the specification of command sequences and timing, as the most typical use case.

- Specifying a sequence: The following diagram (Figure 4) depicts a required sequence of commands. It can be expressed in natural language as follows: "*If the monitored component (SUT) receives a moveForward message, then it must start its engines by using the startEngine command and then its wheels by the startWheels command.*" The ordering of the messages is fixed in the requirement. Thus the above scenario says that the startEngine message has to be sent before the startWheels.

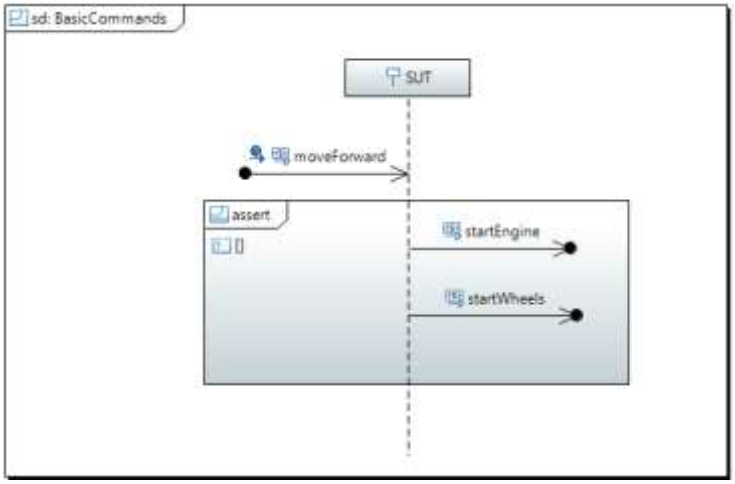

Figure 4. Specifying a sequence of commands

- Specifying alternatives: With the help of alt fragments, alternatives can be specified. This makes it possible to have more concise descriptions, as similar requirements do not have to be duplicated. An alt can have as many branches as needed. There is a variant of an alt fragment called opt, which has only one branch. The following diagram (Figure 5) depicts an alt scenario which can be expressed in natural language as: "*If there is a wallDetected message or a gapDetected message from the sensor, then the monitored component (SUT) shall give a stopEngine command.*"

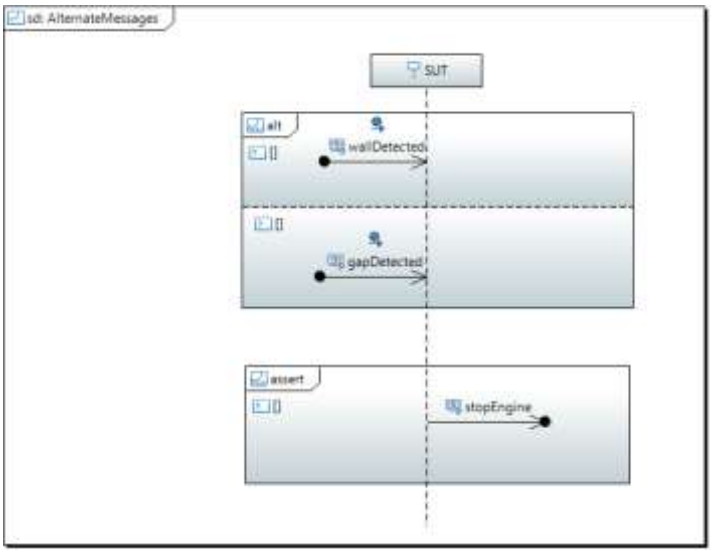

Figure 5. Specifying alternatives

- Parallel messages: If the order of some messages is not important, then a par fragment can be used. The following diagram (Figure 6) depicts a par scenario which can be expressed in natural language as: "*After a lookAround message is received then the extendManipulator and rotateCamera commands have to be sent, but the ordering of these two commands is not important.*"

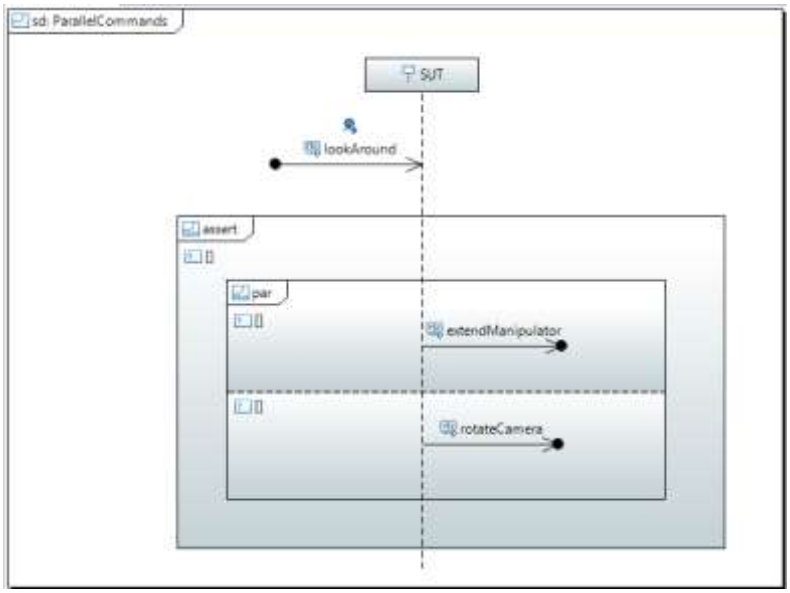

Figure 6. Specifying concurrent commands

- Using time constraints: In UML 2 sequence diagrams timing constraints can also be specified. The scenario below (Figure 7) presents how a duration constraint can be specified. It can be expressed in natural language as follows: "*If a stopRobot command is received, then in maximum 3 time units the stopWheels message and then the stopEngine message have to be sent.*"

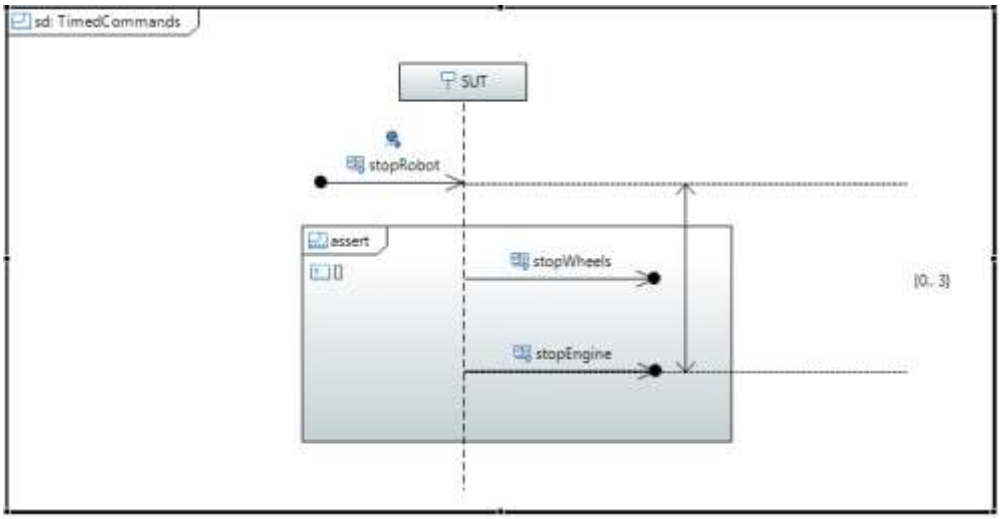

Figure 7. Specifying time constraints

2.2 Applications of the monitors

The goal of monitoring is to check that the behaviour of the checked component is conformant to the specified properties. Properties may specify not only the sequences of inputs and outputs of the checked component, but also context dependency, configuration dependency and time dependency. The handling of the related events is introduced in the following subsections.

2.2.1 Monitoring component behaviour

Basically, a monitor can observe the behaviour of the monitored component in two ways. Monitoring the *externally observable behaviour* (i.e., non-invasive monitoring) means that the monitor observes the timed sequence of inputs and outputs (events) on the interface of the component, together with context and configuration related information, and decides whether the runtime trace of these events is conformant with the specified properties (that define a set of allowed traces). Monitoring the *internal behaviour* (i.e., when the monitor is able to observe the variables and the control flow of the component) is possible if the monitor is instrumented to send to the monitor relevant information (events) that allows the checking of the related properties.

In ROS context, if the goal is monitoring the externally observable behaviour of ROS components, then the (trace of) events can be extracted from messages observable on ROS topics. Accordingly, the monitor components are interfaced as illustrated in Figure 8.

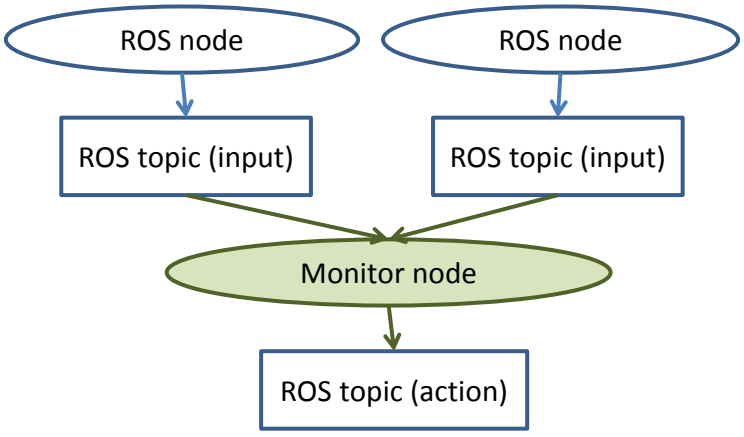

Figure 8. Interfacing the monitor node with ROS topics

Manually written functions are used to subscribe to ROS topics in which messages containing information about the occurrence of the events are published. These functions extract the events and then send these to the monitor. Similarly, manually written functions are used to generate an action when an error is detected by the monitor (Figure 9).

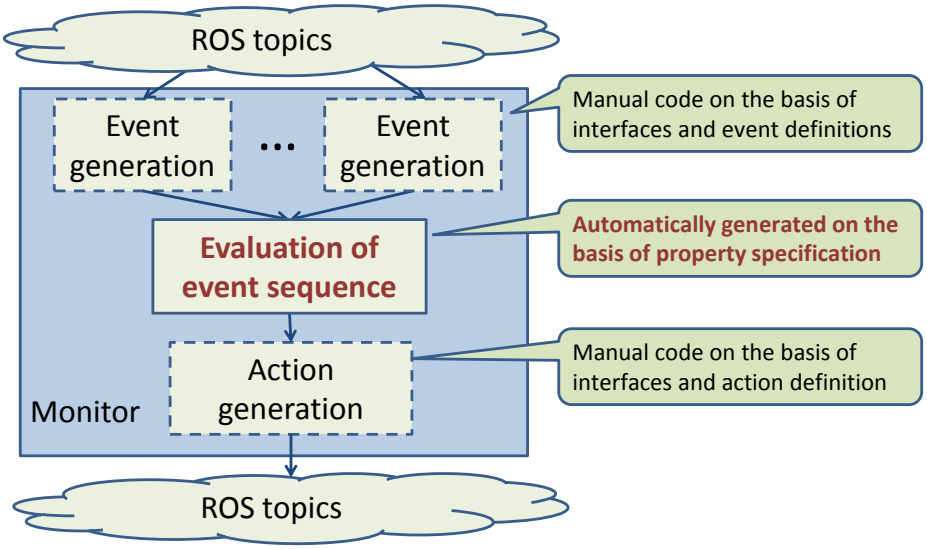

Figure 9. The internal functions of the monitor

In the simplest implementation, the events to be monitored are represented as elements of an enumeration. These are sent to the monitor by calling an evaluate*()* function with the event as a parameter, e.g. *evaluate(Docked)*.

It is exactly the *evaluate()* function that is generated automatically. It checks whether its input parameter (the current event) is an allowed successor of the previous event(s), i.e., the runtime sequence of events is allowed. Considering as example the statechart specification given in Figure 3, the Docked event is valid successor of PathExists and also of NotDocked if NotDocked occurred less than MaxTryD times and there was less than DockingTO seconds after the first NotDocked event (see in states RelativeMove and RetryDocking in the statechart).

In case of an event is not a valid successor of the previous one(s) then an invalid behaviour is detected and the *evaluate()* function calls an *errorAction()* function with the last valid event and the detected invalid event as parameters (e.g., the call *errorAction(NoWay, Docked)* is the case when detecting that Docked is not a valid successor of NoWay). This function is implemented manually and contains the functionality that is needed for handling the error (e.g., stopping the robot).

In case of a simple Turtlesim example[3], excerpts from the interface functions and the generated monitor function are illustrated in Figure 10.

The concrete interfaces of the generated monitor code are detailed in Section 6.

[3] http://wiki.ros.org/turtlesim

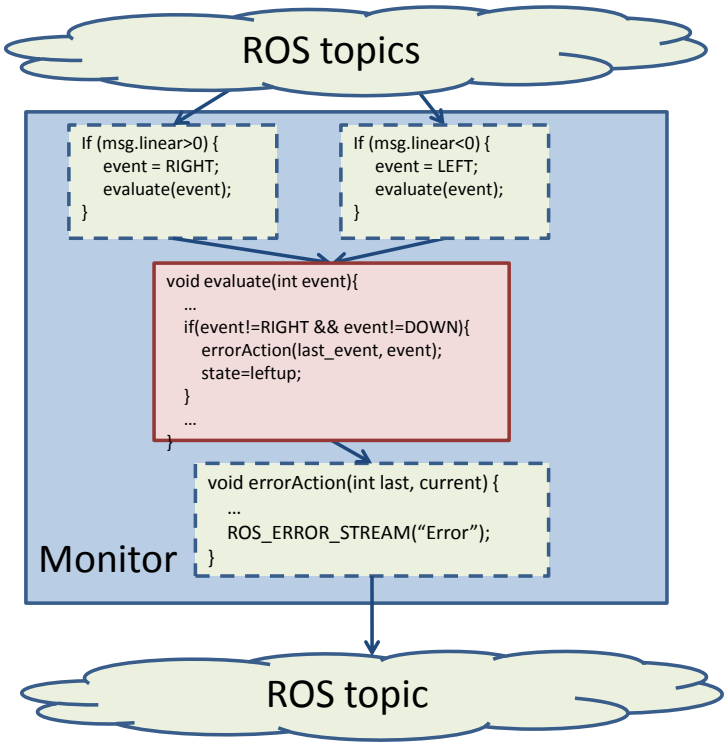

Figure 10. Illustration of the interface functions

2.2.2 Monitoring context dependent behaviour

As presented in D34.10, for specifying context-aware behaviour, the context is captured in the form of a *context model* that describes the environment of the checked system. The *static part* of the context model supports the representation of the environment objects, their attributes and relations, this way a scene of the environment (e.g., the furniture of a room with their colours, sizes and positions). The objects are modelled using a type hierarchy. The *dynamic part* of the context model includes the concepts of changes with regard to objects as well as their properties and relations. Changes are represented as *context events* (e.g., appears, disappears, moves) that have attributes and relations to the changed static objects and their relations (depending on the type of the context event).

The abstract syntax of the context model is defined in form of a *context metamodel*. Note that the type hierarchy of this metamodel can be systematically constructed on the basis of existing domain ontologies (e.g., RoboEarth, KnowRob). The metamodel is completed with *well-formedness constraints* (that define conceptual rules) and *semantics constraints* (that are application-specific preconditions or expectations).

An example context metamodel of a household robot is presented in Figure 11.

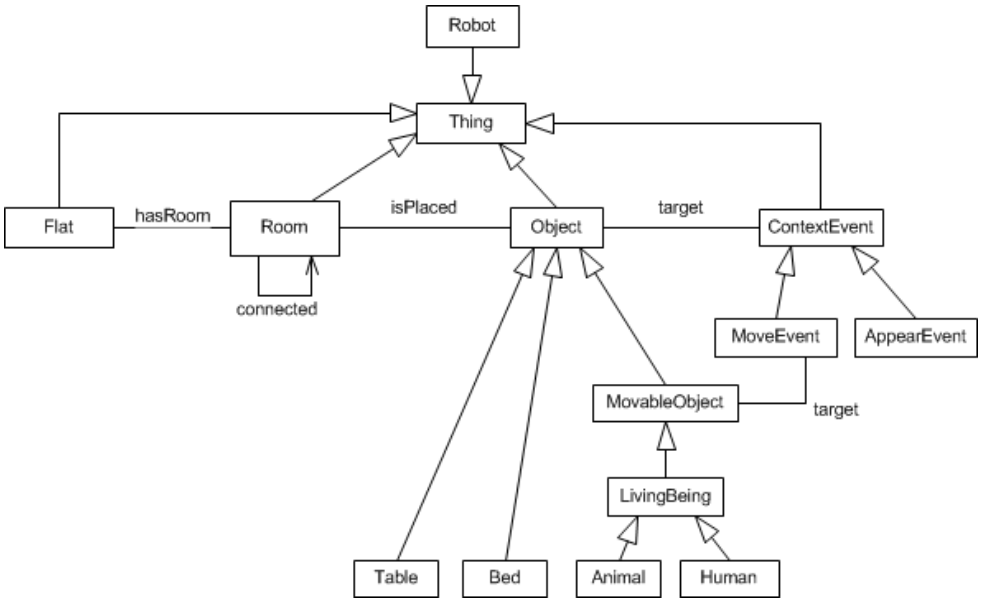

Figure 11. Parts of the context metamodel of a home environment

In the specified properties, the context dependency is formalized using so-called *context fragments*. Context fragments are partial instance models of the context metamodel that specify the relevant objects and relations from the point of view of the specified property (e.g., from the point of view of collision avoidance it is important that the robot is in close proximity to furniture, but it is not relevant what is the shape of the room and what is the colour of the wall).

The goal of context-oriented monitoring is to check that the observed behaviour of the internal (control) components of the checked system conforms to the property that specifies context-dependent behaviour. To be able to check this conformance, the monitor shall access the information that is perceived and represented (as a runtime model) about the environment. This way the monitor is able to detect that the context fragment given in the property occurs and then check that the observed behaviour of the checked component is conformant with the specified behaviour in that context.

To detect that the context fragment given in the property occurs the monitor shall perform a matching between the context fragment and the perceived context. Since the context fragments contain object instances with generic names while the perceived context contains real object instances, the matching involves a so-called *valuation* in which binding is established between the objects in the context fragments and the object instances of the matching type (according to the type hierarchy) in the perceived context that satisfy the relations and constraints. In case of a perceived context, multiple valuations may occur, e.g., when there are several object instances that match the type and fit the relations given in the context fragment.

Accordingly, each matching with a given valuation can be considered as an event (satisfaction of the context dependency) to be processed by the monitor. The precise handling of these events is detailed in the next sections where the monitoring algorithms are described. Here an overview of the related problems and the applied solutions is given. Note that these solutions resemble that of the off-line evaluation of recorded test sequences which is detailed in deliverable D4.2.2v2 of the R3-COP project [5].

The context matching problem can be formulated in an abstract way: Context dependency in a specified property is represented as a static context fragment, or a sequence of static context fragments (especially in case of dynamic events that are specified in the initial context fragment and can be mapped to interim context fragments by a pre-processing step). Each context fragment is represented as a graph, and context fragment sequences are represented as graph sequences (where each graph represents a context fragment). The

vertices of the graphs represent the objects, while the edges between the vertices represent relations between the objects of the context fragment.

These graphs have also two labelling functions, for the vertices and one for the edges, derived from the corresponding context fragments. The following labels are defined:

- Labels for vertices describe the type of the object, which is represented by that vertex (e.g., *Human* or *Furniture*).
- Labels for edges describe the relations between the represented objects (e.g., *isPlaced* or *nearBy*).

The graph sequences derived from the specified properties are called *requirement graph sequences*.

The sequence of context changes perceived by the checked system (and thus the monitor) is mapped to a graph sequence in a similar way. These graph sequences are called *context graph sequences*.

The matching of observed behaviour with the specified properties needs special considerations.

- *Matching all requirement graphs from each step of the observed trace*: To check potential violations of any requirement in each step of the observed trace, matching of each requirement shall be examined in each step of the observed trace (Figure 12). Moreover, in each step the requirements that were already partially matched in the previous steps, shall be checked for progress (continuation or failure of the matching).

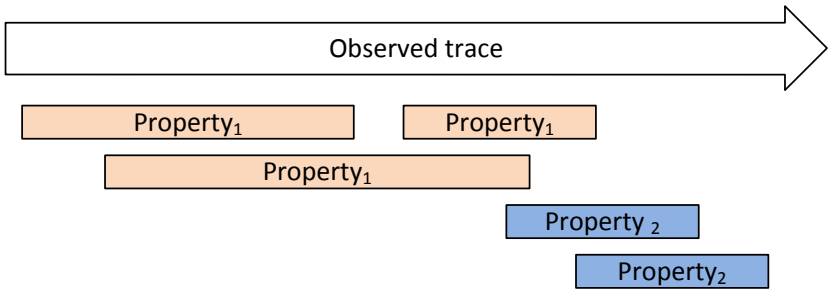

Figure 12. Matching of requirements (specified properties)

Accordingly, a context graph (representing an observed context in a given step) shall be matched to multiple requirement graphs. To solve this problem, we use a graph matching algorithm that is optimized for matching multiple graphs: the requirement graphs that are to be checked for matching a context graph in a given step are represented together in a so-called *decomposition structure*. In a decomposition structure the isomorph subgraphs (from multiple graphs) are represented only once, and this way the re-use of partial matching is supported. Re-use is efficient when the requirement graphs contain similar patterns, which is expected when the behaviour of a robot in a given environment (e.g., in a living room, where similar setup of objects appear in case of several requirements) is specified. In Figure 13 two requirement graphs (CF1 and CF2' on the left) and their decomposition structure (on the right) are illustrated. The dashed rectangles represent individual subgraphs stored in the decomposition structure, while the dotted lines represent how a complex subgraph is decomposed into simpler ones. For example, the graph representing CF1 is decomposed into one which contains only a Room vertex, and one with a Robot and LivingBeing vertices. This latter subgraph consisting of the vertices Robot and LivingBeing can be found in both requirements, but it is represented only once, thus its matching detected in the first requirement graph shall not be checked again when the second requirement graph is checked. The matching of graphs, as a core algorithm, is based

on the work of Messmer et al. [17]. On the top of this algorithm for matching individual graphs, we developed an algorithm that also handles valuations.

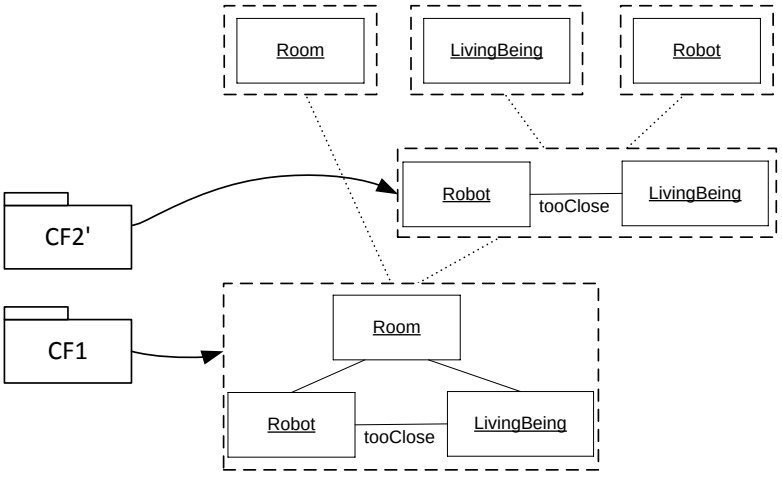

Figure 13. Requirement graphs and their decomposition structure

It may happen that the same requirement can be matched from multiple steps of an observed trace, even in an overlapped way (e.g., when the robot moves close to several objects). To solve this problem, several instances of the monitor (as observer automaton) are executed to check the matching. Each observer automaton has a loop transition in its initial state, this way the matching can be started at any step of the observed trace, as there will be a run of the automaton that skips any potential prefix (see in Figure 14: there is an active initial state and two potential next states). This also solves the problem of matching one requirement overlapping with itself (e.g., on Figure 12).

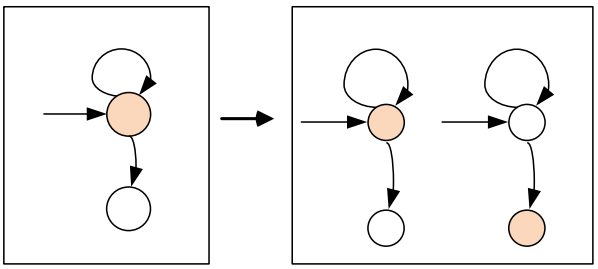

Figure 14 The nondeterministic semantics of the observer automaton

- *Handling the potential valuations*: If there are several potential matching to the context graph (i.e., with different valuations of graph elements), then an automaton instance is created for each possible valuation. To keep track of the potential valuations that may be applied at the same time, these are represented in a separate data structure linked to the decomposition structure. For example in Figure 15 the LivingBeing element in the requirement can be matched either to a Human or an Animal in the trace.

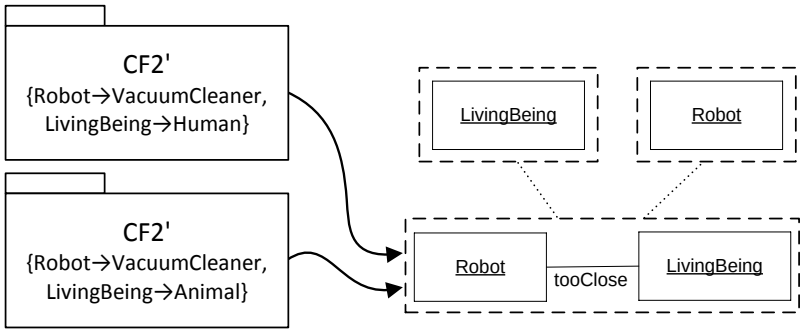

Figure 15 Handling valuations of the same graph structure

- *Matching abstract relations*: The mapping between the abstract relations and the concrete values (in the observed trace) shall be considered. To reconstruct the abstract relations, the monitor performs a pre-processing step on the observed trace which derives the valid and relevant abstract relations on the basis of the concrete values.
- *Matching the hierarchy of object types*: The hierarchy of the types of context objects shall also be considered: a subtype instance in the observed trace shall match its ancestor type in the requirement. To match the labels of vertices and edges (i.e., to provide valuations of graph elements), the so-called compatibility relation is introduced (instead of the direct equivalence of labels), that conforms to the type hierarchy defined in the context metamodel.
- *Handling dynamic changes*: There may be dynamic objects specified in the initial context fragment that appear/disappear with a given timing. Since the requirements can also contain a sequence of events, actions, and interim contexts that not necessarily include the precise timing of their occurrence (only their ordering), the relation between these and the occurrence of the dynamic changes is not known in advance. Therefore, the matching procedure shall insert these changes into the requirement graph sequence on-the-fly when the timing of the observed trace (relative to the start of matching) equals the timing property of a dynamic event.
- *Nondeterministic observer automaton*: A requirement may contain alternatives in the behaviour, this way one state in the observer automaton may have more successor states. The evaluation shall consider all possible runs simultaneously.

The decomposition based approach offers a significant increase in efficiency [17]: the search is faster than the classical Ullman's algorithm; in best case its expense is $O(IM)$ while in worst case it is $O(NI^M M^2)$, where N is the number of graphs, I is the number of vertices in the context graph, and M is the average size of the requirement graphs. In the best case the N graphs are the same, while in the worst case N completely different complete graphs are decomposed. Of course, the decomposition structure has to be constructed off-line that is characterized in worst case by $O(N^2 M^{M+3})$. Considering behavioural requirements of a robot operating in a given environment, the common parts in the requirement graphs are relatively frequent.

Another important characteristic of the performance of the monitoring is the number of observer automata that are executed simultaneously. In our setup the requirements (and thus the observer automata) are relatively small (i.e., they consist of a small number of states and transitions), but the observed traces are typically long. However, the number of simultaneous observers does not depend on the length of the observed trace, but depends on the structure of the observer automata (mainly on the alternative behaviours), and the number of specified dynamic events (that may interleave with the recorded events and actions).

2.2.3 Monitoring configuration dependent behaviour

The monitoring of configuration-dependent behaviour is handled in a similar way like monitoring context-dependent behaviour. To specify configuration dependency, a configuration metamodel and configuration fragments are used (e.g., to describe that processing of a command is relevant only in case of a configuration that contain the actuator instances that are capable of executing the command). The monitor shall access the runtime configuration of the checked system. To detect that the configuration fragment is satisfied in the current configuration, the monitor shall perform a matching with valuations. Accordingly, each matching with a given valuation can be considered as an event to be processed by the monitor.

In the following, to simplify the description, we do not distinguish context-related and configuration-related events, and uniformly consider these as context-related events.

2.2.4 Monitoring time dependent behaviour

Time dependency is captured in the requirements using time-related predicates in which dynamic clock variables (representing the passing of real time) are compared with constant deadlines or static time variables. The evaluation of a time-related predicate can be considered as a condition (the predicate is valid) or an event (when the time predicate becomes valid). The precise handling of these conditions or events is detailed in the next sections where the monitoring algorithms are described.

2.3 Tool-chains for the synthesis of monitors

Having the three different languages to specify the properties to be monitored, three tool-chains are defined for monitor synthesis.

- Monitor synthesis on the basis of behaviour specification (Figure 16): In this case we support the Yakindu Statechart Tools[4] (open source Eclipse-based editor) to construct the state machine model. It is extended with timeouts and content/configuration related events. The internal representation for monitor synthesis is an XText based language, to which we provide a systematic mapping from the statechart model. The monitor code (the event evaluation function) is generated by an Eclipse plugin. The details of the tool-chain are given in Section 3.

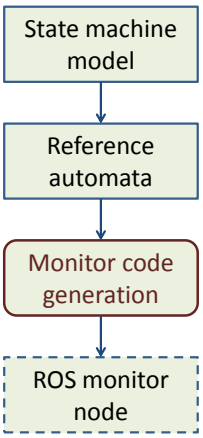

Figure 16. Monitor synthesis on the basis of behaviour specification

[4] The Yakindu tool can be downloaded from http://statecharts.org/

- Monitor synthesis on the basis of scenario specification (Figure 17). In this case we support the Papyrus tool[5] for UML2 Sequence Diagrams to construct the scenario model. It is extended with timing and content/configuration conditions using the conventions as described in deliverable D34.10. The internal representation for monitor synthesis is an XText based language, to which we provide a systematic mapping from the scenario model. The details of the tool-chain are described in Section 4.

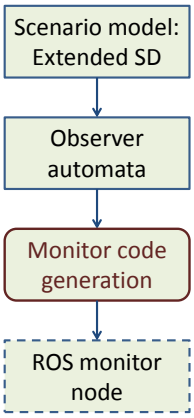

Figure 17. Monitor synthesis on the basis of scenario specification

- Monitor synthesis on the basis of temporal specification (Figure 18). In this case we provide a Sirius-based editor to compose temporal properties using patterns; the editor contains a built-in pattern library. The composed properties are exported in the (textual) format of the CaTL temporal logic that we defined in D34.10. The CaTL formulas are mapped to so-called evaluation blocks that support a tableau-based approach to evaluate temporal logic formula. The details of the tool-chain are given in Section 5.

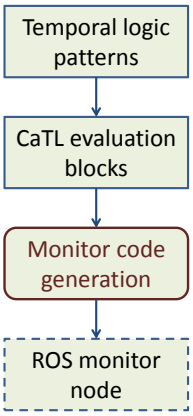

Figure 18. Monitor synthesis on the basis of temporal specification

These tool-chains aim at offering a flexible framework to specify properties according to the focus and level of completeness of the behaviour to be checked:

- Monitor synthesis on the basis of behaviour specification (UML2 statecharts extended with timeouts and context/configuration related events) is useful when the designer wants to specify *complete reference behaviour*. The monitor is responsible to detect and signal any behaviour that is different from this reference behaviour.

[5] The Papyrus tool can be downloaded from https://eclipse.org/papyrus/ (the latest 1.1.X version from Eclipse Mars is supported).

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

It is important to emphasize that the input languages are extensions of well-known UML2 diagrams (statechart and sequence diagrams) and typical property patterns, with which the designers may be familiar. This can reduce the time needed to learn and safely apply these tools.

3 Monitoring on the basis of behaviour specification

In this section we describe the process used for supporting the development of statechart based runtime monitors. The developers use the Yakindu Statechart Tools to define the properties to be monitored. From this modelling language an intermediate low-level representation is constructed (by the Intermediate Language Framework in Figure 19). This intermediate representation can be further developed and extended by a textual editor (provided in the framework). Moreover, it is also possible to specify or change context information in this phase. In the last step, a runtime monitor component is generated from the intermediate representation. This will be integrated with the system under monitoring.

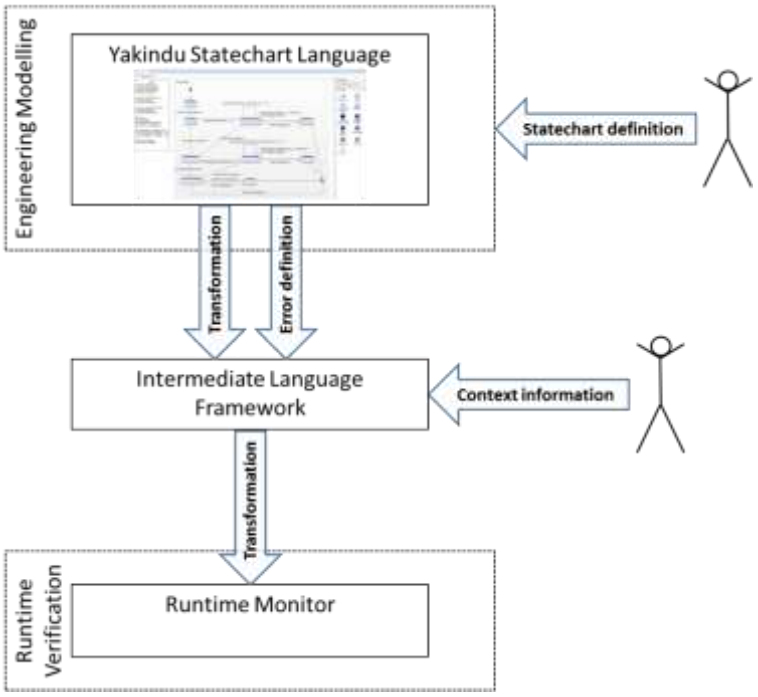

Figure 19. Overview of the monitor generator infrastructure

3.1 Statechart modelling

Yakindu Statechart Tools is widely used to support the development of state based models. It has an easy-to-use graphical interface. In addition, it is integrated into the Eclipse Modeling Framework (EMF) which provides portability and integration possibilities with other technologies and tools.

The statechart presented in Figure 3 was constructed using Yakindu. Variables were defined and complex guard expressions and actions were used to express the reference behaviour. We will use this example in the following.

3.2 Constructing the intermediate representation

In the following we overview the Intermediate Language Framework developed for the synthesis of runtime monitors. Textual editors support the modification and extension of the models in this step of the monitor synthesis process. This provides the ability to further extend the statechart models with additional information.

The intermediate representation (the so-called *intermediate statechart language*) is formalised by mapping it to the formal language of *transition systems*. This means that the semantics of the language is given formally (which is not common for high level engineering model-

ling languages). This also allows the formal verification of the reference behaviour specified for monitor synthesis.

3.2.1 General overview

The purpose of the intermediate statechart language is twofold. First, it enables the hierarchical specification of complex statechart monitors with concurrent, timed behaviour. Second, a syntactically restricted fragment (called *observer statecharts*) forms the basis of the monitor source code generation as presented in Figure 20. The transitions between the formalisms are defined and implemented as model transformations.

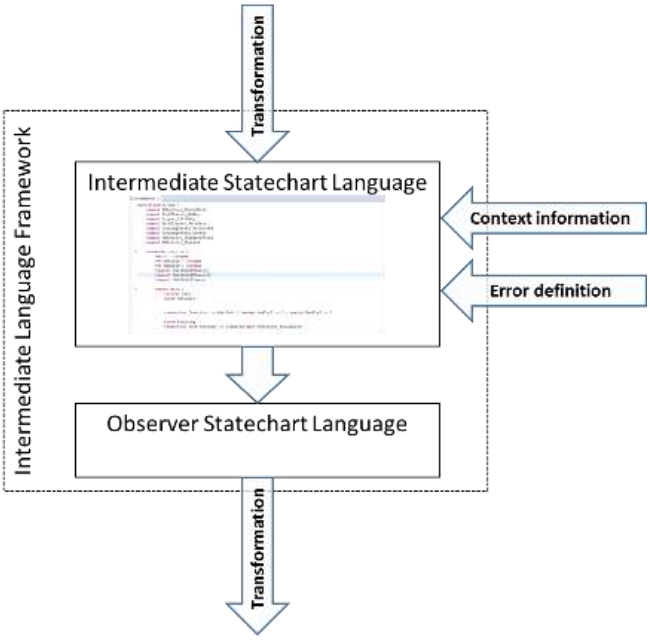

Figure 20. Overview of the intermediate language framework

In the following, the syntax and semantics of the intermediate statechart language are introduced briefly, and also the runtime monitoring infrastructure is overviewed.

3.2.2 Syntax of the language

The abstract and textual concrete syntax of the intermediate statechart language were designed to be simple and expressive. As it serves as an intermediate representation between the engineering models and the runtime monitors, it does not have a graphical syntax. In the following subsections, the abstract syntax is given in the form of EMF metamodels, while the textual syntax is defined by the syntax rules.

In general, the structure of the language conforms to visual statechart languages. The naming and other conventions are similar to programming languages like C or C++.

3.2.2.1 Extensions to the constraint language

The intermediate statechart language is defined as an extension of the constraint language described in D34.10. Special types for representing context information (context fragments) can be declared with the help of the Class keyword (reminding of the classes of context objects). This way the context events can be integrated into the monitor generation infrastructure. The structure of class declarations are given in Figure 21.

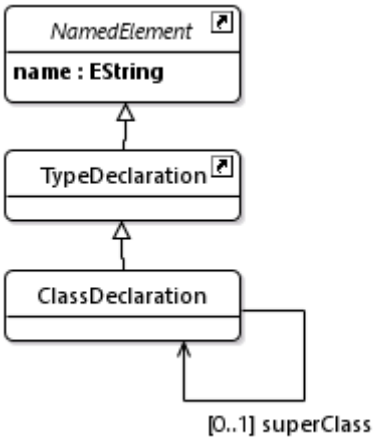

Figure 21. Class declarations

TypeDeclaration := … | *ClassDeclaration*

ClassDeclaration := **class** *Name* {**extends** *Name*}

3.2.2.2 Statechart specifications

Statecharts are defined as constraint specifications (see D34.10), thus they enable the declaration of types, constants and functions, and the definition of constraints over them. Moreover, in the scope of a specification, an arbitrary number of statecharts, signals and patterns can be declared.

Statecharts provide a proper means to describe an (active) component. As a single component can be divided further into smaller subcomponents, an enclosing specification is used for the description of the whole component and the various subcomponents that can interact with each other (Figure 22).

Patterns are used to declare the input interface of the received context information.

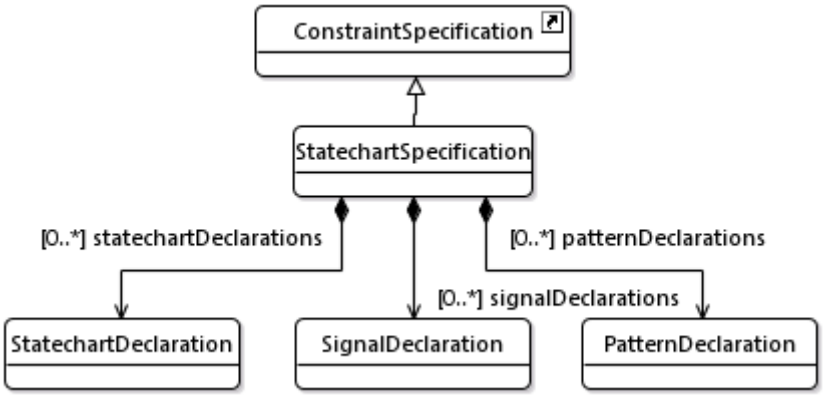

Figure 22. Statechart specifications

StatechartSpecification := **specification** *Name*{ ([*Declaration*],$^{+}$) } {[
TypeDeclaration | … |
StatechartDeclaration |
SignalDeclaration |
PatternDeclaration]* }

StatechartDeclaration := **statechart** *Name*{ ([*Declaration*],*) } := *Statechart*

SignalDeclaration := **signal** *Name*{ ([*Declaration*],*) }

PatternDeclaration := **pattern** *Name*{ ([*Declaration*],*) }

3.2.2.3 Statechart declarations

In order to support developers specifying the expected reference behaviours of the components properly, parameters can be used in the statechart instantiations: parameterized statecharts (Figure 23) can be created from existing templates by providing a value for each parameter. This is useful where components express similar behaviours (no redundant specifications have to be developed).

Behaviours of the statecharts are structured by regions. At the root of every statechart, a region encloses all of the states. A state can have an arbitrary number of inner regions and each region must contain at least one initial state to ensure the validity of the model.

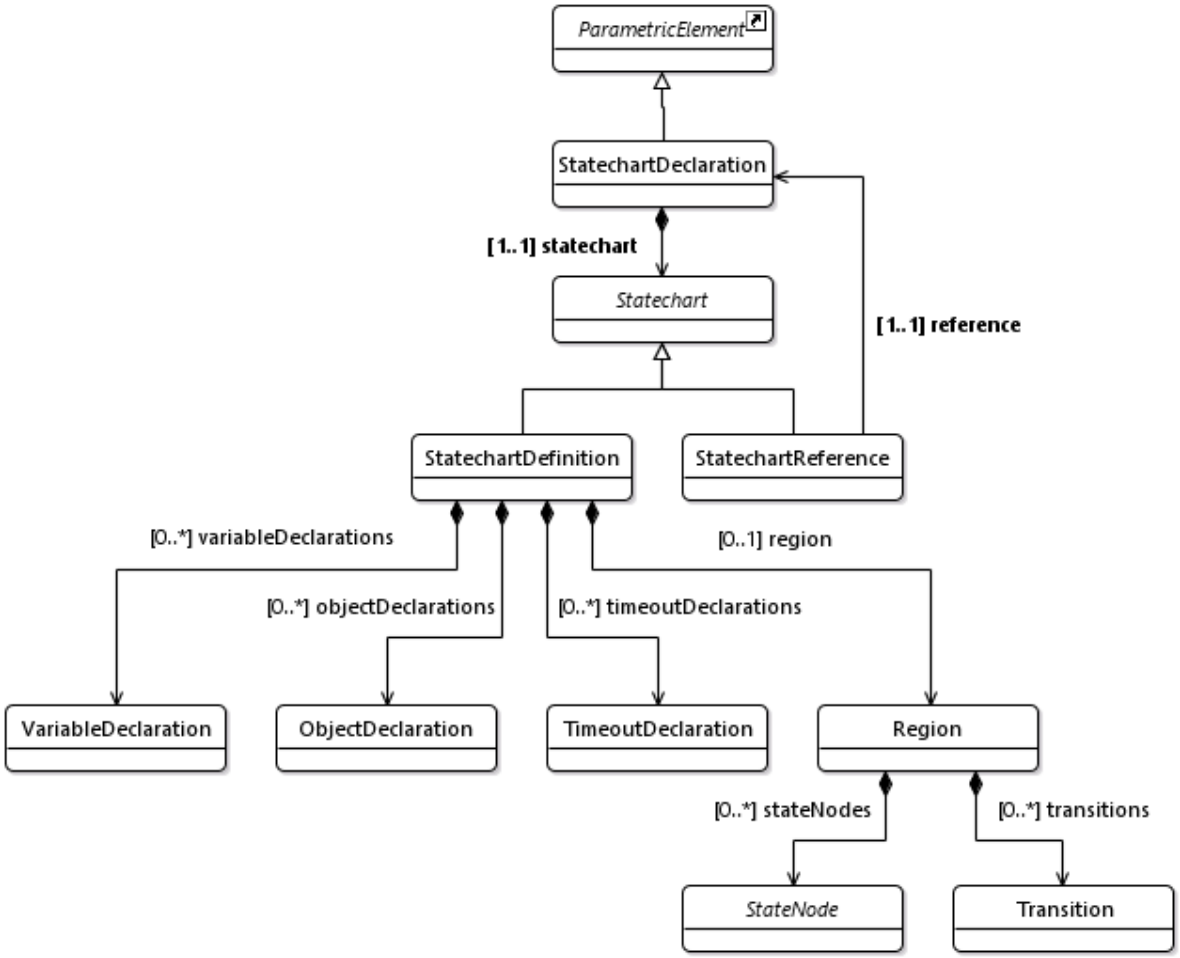

Figure 23. The structure of statechart definitions

Statechart := *StatechartDefinition* |
StatechartReference

StatechartDefinition := {
[
VariableDeclaration |

```
                    ObjectDeclaration |
                    TimeoutDeclaration
                ]*
                {Region}
            }
StatechartReference  := Name{ ([Expression],*) }
VariableDeclaration  := var Declaration {:= Expression}
ObjectDeclaration    := object Declaration
TimeoutDeclaration   := timeout Name
Region               := region Name {[
                        StateNode |
                        Transition
                    ]*}
```

3.2.2.4 States

Control flow is specified with the help of states and pseudostates (Figure 24). A state is always contained by at least one region. A pseudo state only serves to represent specific properties and relations like initial-, fork-, join-, or choice pseudostates.

States can either be atomic states or composite states. An atomic state is a state which does not contain inner regions. A state might contain entry and exit actions (Figure 25), which are executed when entering or exiting the state. Composite states contain one or more inner regions. Composite states help the developers to organize the definition of the behaviours according to some structure.

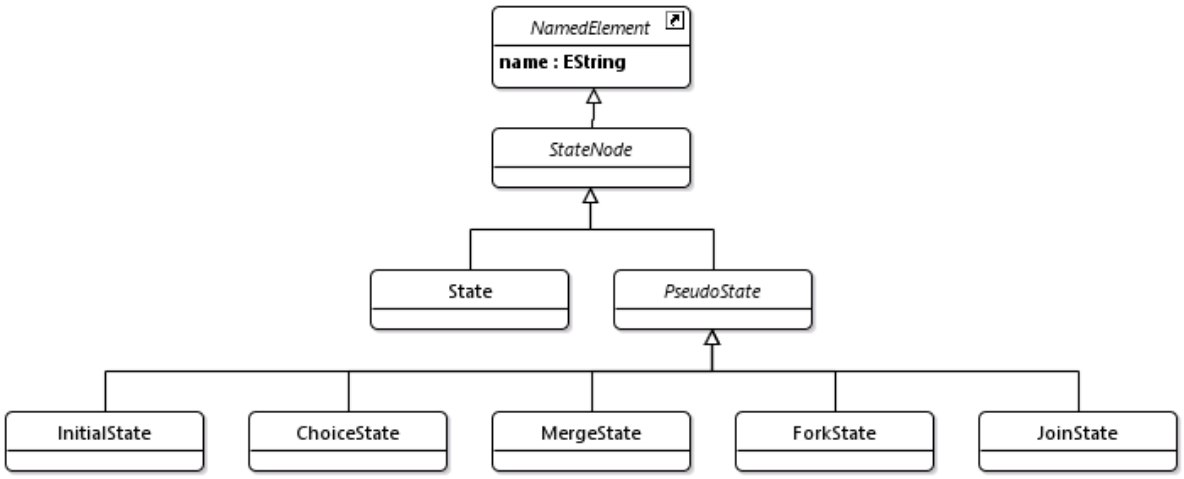

Figure 24. State nodes

A region must contain an initial state and activating a region means the activation of its initial state.

Choice states are pseudo states where the control flow branches according to specific conditions. Merge states collect the various branches together.

A fork state is a pseudo state that has a single incoming transition and an arbitrary number of outgoing transitions that are directed to different regions. The outgoing transitions cannot have triggers, guards, or actions. Fork states are used to define the simultaneous activation of multiple regions.

A join state is a pseudo state that has a single outgoing transition and multiple incoming transitions directed from different regions. The incoming transitions cannot have triggers, guards, or actions. Join states serve as a synchronization point in the control flow.

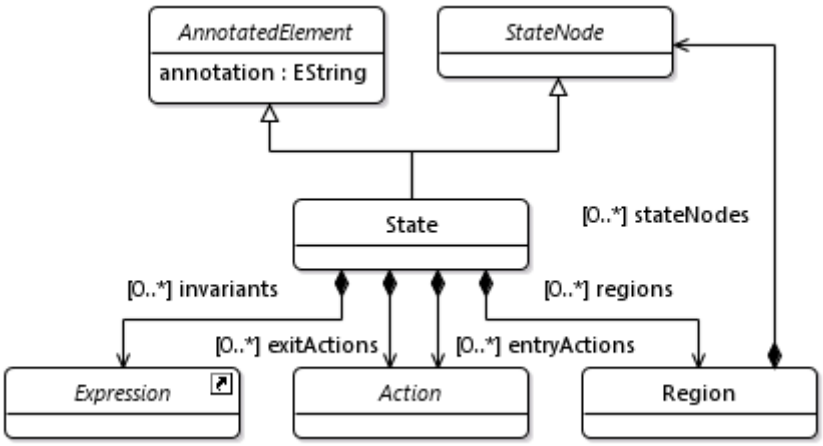

Figure 25. Structure of a state

StateNode := *State* |
InitialState |
ChoiceState |
MergeState |
ForkState |
JoinState

State := {*Annotation*} **state** *Name* {{
[*Invariant*]*
{*EntryActions*}
{*ExitActions*}
[*Region*]+
}}

Annotation := @*Name*

Invariant := **invariant** *Expression*

EntryActions := **entry** / [*Action*];+

ExitActions := **exit** / [*Action*];+

InitialState := **initial** *Name*

ChoiceState := **choice** *Name*

MergeState := **merge** *Name*

ForkState := **fork** *Name*

JoinState := **join** *Name*

3.2.2.5 Transitions

Transitions describe the possible state changes in the model. A transition is enabled if the source state is active, the guard expression is satisfied and an event trigger received. Firing a transition changes the active state of the region from the source state to the target state.

The guard condition of a transition is an expression that evaluates to a Boolean value. (The transition is enabled only if the guard condition evaluates to true.)

Actions are associated with transitions: these actions are executed when the transition fires.

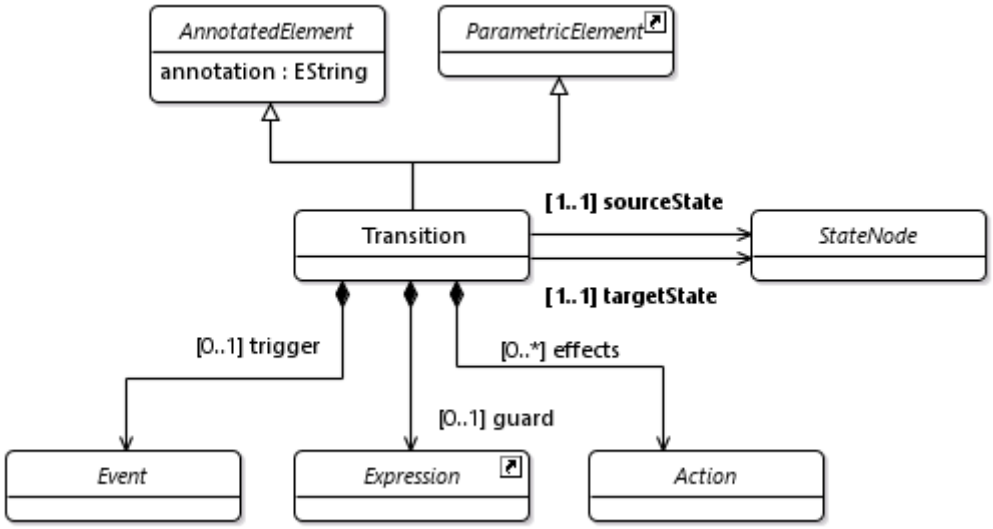

Figure 26. Structure of a transition

Transition := {*Annotation*} **transition**
Name{ ([*Declaration*], *) }
from *State* **to** *State*
{*Event*}
{ [*Expression*] }
{/ [*Action*], * }

3.2.2.6 Events

Various events (that provide input for the state machine through related interfaces) trigger a reaction of the statechart (e.g., the state changes). These events represent observations in the checked system (SignalEvent in Figure 27), refers to the occurrence of a context event (PatternEvent), or represent the expiration of a timer (TimeoutEvent).

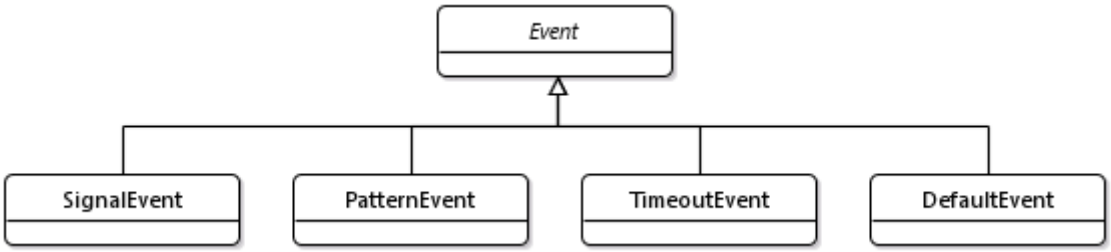

Figure 27. Hierarchy of events

Event := *SignalEvent* |
PatternEvent |
TimeoutEvent |
DefaultEvent

SignalEvent := **upon** *Name*{ ([*Expression*], *) }

PatternEvent := **case** *Name*{ (*[Expression]*,*) }

TimeoutEvent := **when** *Name*

DefaultEvent := **by default**

3.2.2.7 Actions

Actions are used to define reactions. Actions can change the values of the variables (AssignmentAction in Figure 28), send signals (SignalAction) or manipulate timers (SetTimeoutAction or DeactivateTimeoutAction).

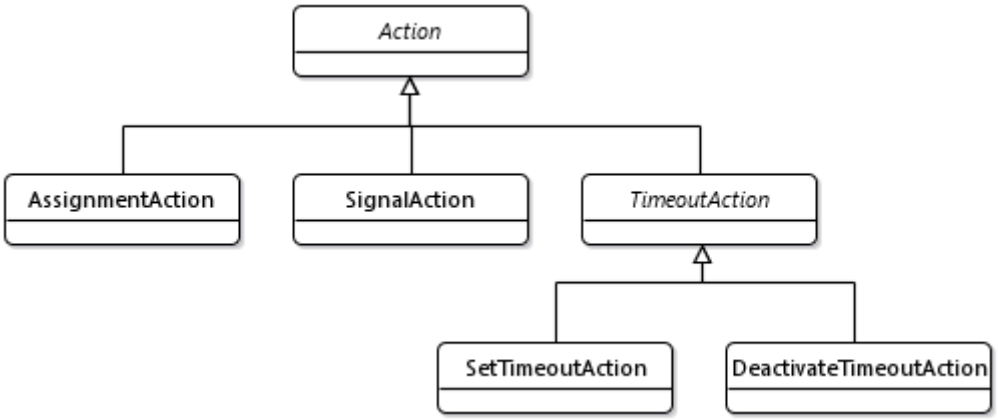

Figure 28. Hierarchy of actions

Action := *AssignmentAction* |
SignalAction |
SetTimeoutAction |
DeactivateTimeoutAction

AssignmentAction := **assign** *Expression* := *Expression*

SignalAction := **send** *Name*{ (*[Expression]*,*) }

SetTimeoutAction := **set** *Name* := *Expression*

DeactivateTimeoutAction := **deactivate** *Name*

3.2.3 Signals

Signals are used in the communication among specified (sub)components. These signals are declared directly in the specification. Signals can be used with an integer parameter (which can be either a constant or a variable).

3.2.4 Explicit error definition

A state or transition can be explicitly declared as erroneous using annotations defined in the syntax of states and transitions (see for example in Figure 26). This information is directly used by the monitor synthesis algorithm (when an erroneous state is reached or an erroneous transition is fired then the monitor shall output an error action).

When no explicit error definition is present in the model, then those behaviours are considered as erroneous that are not defined explicitly by the statechart model.

3.2.5 Well-formedness rules

Well-formedness rules have to be satisfied in order to specify a valid and executable monitor. The following rules are checked in the specification (in addition to those which are explicitly represented in the metamodel):

- Each region shall have exactly one initial state.
- Each region shall have at least one (non-pseudo) state.
- The reaction to the input events has to be deterministic:
  - There aren't two or more outgoing transitions triggered by the same event.
  - There is no transition without trigger ("always" is considered as a specific trigger that enables the transition).
- Entry and Exit states shall not have input or output transition.
- All states have to be reachable.
- There shall not be livelocks in the specification.

To illustrate the definition of well-formedness rules, in the following example we present the rule which validates that all transitions have trigger. Note that the validation can be implemented at the level of the Yakindu Statechart Tool and also at the level of the intermediate language. Currently the rule is defined for the Yakindu model, however it is straightforward to adapt it to the intermediate language.

```
/* Pattern for triggered transitions. */
pattern transitionWithTrigger(transition : Transition) {
	Transition.trigger(transition, _trigger);
}

/* Pattern that returns the left or right operand (or the operand if there
is only one) of the initial expression. */
pattern recursiveExpressions(expression : Expression, nextExpression :
Expression) {
	LogicalRelationExpression.leftOperand(expression, nextExpression);
} or {
	LogicalRelationExpression.rightOperand(expression, nextExpression);
} or {
	LogicalAndExpression.leftOperand(expression, nextExpression);
} or {
	LogicalAndExpression.rightOperand(expression, nextExpression);
} or {
	LogicalOrExpression.leftOperand(expression, nextExpression);
} or {
	LogicalOrExpression.rightOperand(expression, nextExpression);
} or {
	LogicalNotExpression.operand(expression, nextExpression);
} or {
	ParenthesizedExpression.expression(expression, nextExpression);
}

/* Returns triggered transitions, and those transitions that shouldn't be
triggered. */
pattern transitionsWithTriggerOrDefault(transition: Transition) {
	Transition.trigger(transition, trigger);
	ReactionTrigger.triggers(trigger, _aTrigger);
} or {
	Transition.trigger(transition, defaultTrigger);
	DefaultTrigger(defaultTrigger);
} or {
	Entry.outgoingTransitions(_entry, transition);
} or {
```

```
		Choice.outgoingTransitions(_choice, transition);
} or {
		Exit.parentRegion.composite(_exit, parentState);
		State.outgoingTransitions(parentState, transition);
		neg find transitionWithTrigger(transition);
} or {
		Transition.trigger(transition, trigger);
		ReactionTrigger.guard(trigger, guard);
		Guard.expression(guard, expression);
		find recursiveExpressions+(expression, eventValue);
		EventValueReferenceExpression(eventValue);
}

@Constraint(targetEditorId =
"org.yakindu.sct.ui.editor.editor.StatechartDiagramEditor",
		message = "Missing trigger. Transition is never taken. Consider
'always' instead.",
		severity = "error",
		location = transition
)

/* Returns transitions without a trigger. */
pattern transitionsWithoutTrigger(transition : Transition) {
		Transition(transition);
		neg find transitionsWithTriggerOrDefault(transition);
}
```

3.2.6 An example

As an illustration of the usage of the formerly introduced statechart concepts, we show an example. Figure 29 depicts a sequence chart with a context fragment. The occurrence of the context fragment (i.e., matching with the observed context) triggers a sequence of interactions. The statechart that specifies a reference behaviour belonging to this sequence is presented below using the textual syntax. Here four states (initial, start, matched. detected) and an error state (error) are declared together with the transitions that process the observed events and the context events (ContextFragment2 with concrete valuations).

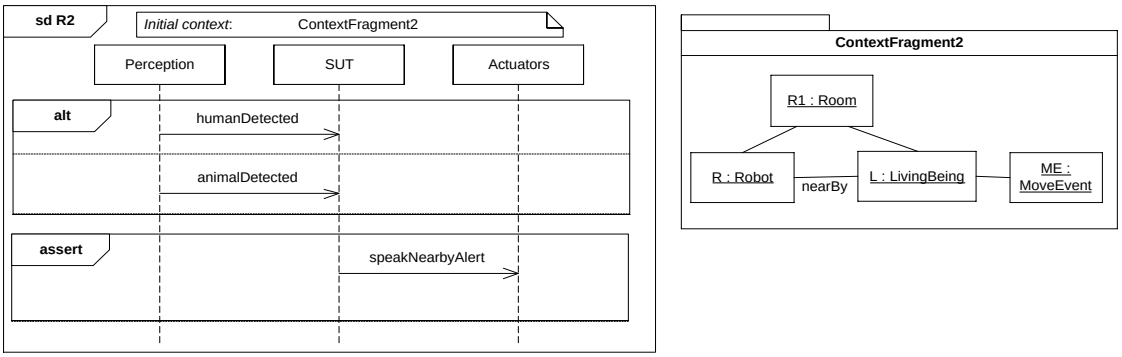

Figure 29. Example scenario model R2: Alerting a living being

```
specification AlertLivingBeing {

		class Room
		class Robot
		class LivingBeing
		class MoveEvent
```

```
	pattern ContextFragment2(
		r1 : Room,
		r : Robot,
		l : LivingBeing,
		me : MoveEvent
	)

	signal humanDetected
	signal animalDetected
	signal speakNearbyAlert

	statechart Monitor := {
		// ...
	}
}
```

```
statechart Monitor := {
	object r1 : Room
	object r  : Robot
	object l : LivingBeing
	object me : MoveEvent

	region main {
		initial init
		state start
		state matched
		state detected
		@ERROR state error

		transition from init to start
		transition from start to matched
			case ContextFragment2(r1, r, l, me)
		transition from start to start by default
		transition from matched to detected upon humanDetected
		transition from matched to detected upon animalDetected
		transition from matched to start by default
		transition from detected to start upon speakNearbyAlert
		transition from detected to error by default
		transition from error to error by default
	}
}
```

3.2.7 Mapping to the observer statechart

The intermediate statechart language provides a precise representation of the high level UML2 statechart models. This is transformed into a syntactically restricted fragment, the so-called observer statechart by the elimination of the complex constructs that cannot be implemented in monitor source code directly. Namely, the hierarchical structure of the intermediate statechart is flattened, representing the state hierarchy and parallel regions by the corresponding state configurations (consisting of the combinations of basic states). Note

that in many cases, developers specify statecharts without hierarchy and parallelism. In these cases it is possible to construct observer statecharts directly from the engineering models. This is the situation in the statechart example in Figure 3.

3.3 Monitor source code generation

The runtime monitoring component is automatically generated from the observer statechart. The main purpose of the monitoring is to detect violations of the specification. Specifying error states/transitions can explicitly express errors. Another option is the implicit declaration of erroneous behaviours: those behaviours are declared erroneous which are not explicitly allowed by the statechart definition. The monitor generation algorithm will generate different monitors according to the error declaration.

In the following the basic ideas behind the generation of monitor source code are presented. Here one variant of monitors is considered, namely, the generation of extensible, object oriented C++ code. In addition, as the interfacing of the monitors is done manually, it was also our goal to provide a simple interface for the monitors.

The source code is generated by processing and representing the various elements of an observer statechart. Basically, the generation procedure consists of two steps depicted on Figure 30. The first step is the construction of the *monitor skeleton* and the generation of the various data structures. In the next step, the *monitor logic* is built according to the statechart specification. Then the output is produced which is a runtime monitor ready to be placed in its environment.

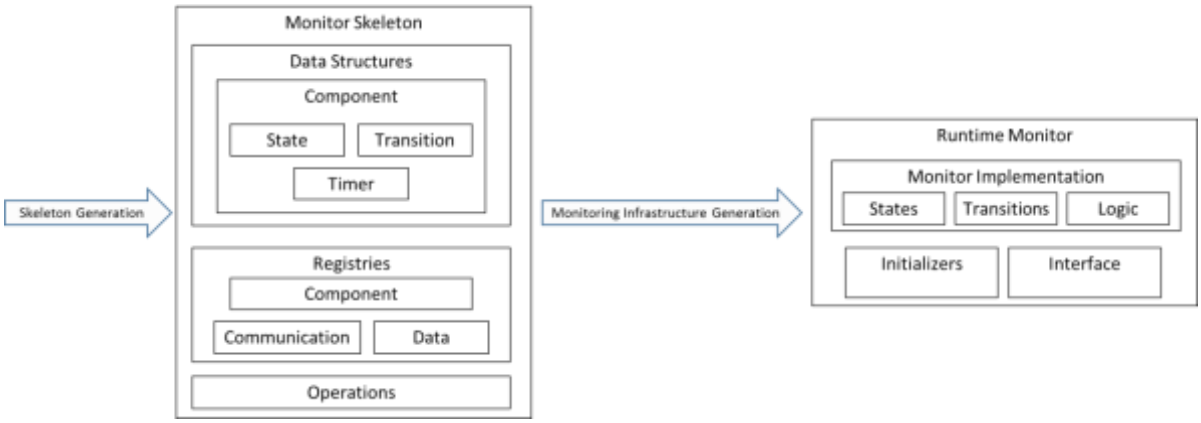

Figure 30. Steps of the monitor generation

In the following we summarize the tasks of the first step. In the intermediate language, specification is used to encapsulate the behavioural specification of the system under monitoring so this serves as a starting point in the monitor generation. In the monitor skeleton, StatechartRegistry encapsulates and manages the generated *monitoring components*. A monitoring component is an individual monitor generated from a single statechart. All the necessary data structures as those for representing the states, transitions and timers are generated in the first phase. Besides the manager object of the subcomponents (StatechartRegistry), the registry components for the handling of communication (EventRegistry) and data (VariableRegister) are constructed according to the variable and event specifications.

The generated monitors can be executed in multiple threads in an environment where the fast processing of events is required. In multi-threaded monitoring, registries ensure the thread-safe behaviour. In addition, each monitor component has its function for calculating their enabled transitions and maintaining the list of active states.

Various data structures and functions are generated to store the relevant parts of the states, transitions and variables. Pseudo states of various types will not be stored directly in the monitor, instead, the transformation adjusts the related transitions according to the semantics of the corresponding pseudostates.

The basic data structures of the states are given on the following code snippet:

```
class State {
    std::vector<Transition*> transitions;
    bool initial;
public:
    std::string name;
    Statechart* parentStatechart;
    State(Statechart* parentStatechart, std::string name =
"unnamed", bool initial = false):parentStatechart(parentStatechart),
name(name), initial(initial) {}
    ~State() = default;

    virtual void EnterState() {}
    virtual void ExitState() {}
};
```

Transitions are explicitly represented in the generated data structures: they are objects connecting two states. In the first phase only the operations and the basic data structures are generated, all the logic related part is generated in the second phase of the transformation. Various operations support the monitoring of state changes.

The basic data structures of the transitions are given in the following code snippet:

```
class Transition {
public:
    Transition(State* from, State* to, bool onEvent = false) :
from(from), to(to), onEvent(onEvent) {}
    State* from;
    State* to;
    std::vector<State*>* stateList = nullptr;
    bool onEvent;
    virtual bool Enabled() {
        return true;
    }
    virtual void Action() {
        return;
    }
};
```

The handling of guards is generated in the second phase. Transitions without actions or guards are instances of the generic Transition class. Transitions with complex guard conditions or actions are inherited classes and function overloading is used to redefine the operations.

The variable types of the statechart language are transformed into plain data types of C++.

The handling of communication is managed by the EventRegistry: it handles communication queues and the related activities. An additional task is to manage synchronization and provide interfaces.

The basic data structures and operations of the EventRegistry are given in the following code snippet:

```
class EventRegistry {
    ~EventRegistry() = default;
    EventRegistry() = default;
```

```
    static std::vector<Event> eventQue;
    static std::vector<std::pair<Event, TimeStamp>>futureEventQue;
public:
    static std::map<Signal, std::vector<Transition*>> mapping;
    static EventRegistry* GetInstance() {
        static EventRegistry instance;
        return &instance;
    }
    static void CheckFutureEvent() {
    auto currentMoment = TimeStamp::getCurrentTimeStamp();
    std::lock_guard<std::mutex> lg{eventQueMutex};
    for(auto i = futureEventQue.begin(); i < futureEventQue.end();
++i) {
         if((*i).second <= currentMoment) {
            eventQue.push_back((*i).first);
            i = futureEventQue.erase(i);
            }
        }
    }
    static void EventArrived(std::string eventName, int value =
noParameter, int timeout = 0) {
        if(timeout == 0) {
            std::lock_guard<std::mutex> lg{eventQueMutex};
            eventQue.push_back(Signal(eventName, value));
        } else {
         std::lock_guard<std::mutex> lg{futureEventQueMutex};
         futureEventQue.push_back(std::make_pair(Event(eventName,
value),TimeStamp::getFutureTimeStamp(timeout)));
        }
    }
    static std::vector<Event> GetCopyOfEventQue() {
        std::lock_guard<std::mutex> lg{eventQueMutex};
        std::vector<Event> copyOfEventQue = eventQue;
        eventQue.clear();
        return copyOfEventQue;
    }
};
```

After the creation of the data structures, the second phase of the monitor generation extends it with the features required to perform monitoring.

In the second phase, we start by instantiating the states. Simple transitions are ones that have one source and one target state specified in this phase. Transitions from and to choice states are handled by unfolding them to simple transitions and by transforming the guard conditions according to the decomposition rules. Special transformation rules (handling state configurations) are also defined for fork and join pseudo states when generating the logic of the monitor.

Events and communication are handled by the EventRegistry, which has to be filled with the data from the model at this phase of monitor generation.

4 Monitoring on the basis of scenario specification

In R3-COP, a transformation algorithm was developed to generate test oracles from scenario specifications for the purpose of evaluating test traces. In this work this basic transformation is adapted and extended for online monitor synthesis purposes. In the first part of the section, the syntax and semantics of the scenario language is overviewed and the procedure of mapping it to observe automata is detailed. In the second part of the section this approach is extended to handle timing-related requirements. The observer automata are represented using an intermediate observer statechart language that provides a proper way to represent the timed monitor specifications.

4.1 Mapping from scenario models to automata

This section overviews the language used to specify properties and the method for transforming these properties to automata models. This will serve as the basis of our further developments.

The main purpose of the language is to express properties on an observed trace. The monitor that checks this trace categorizes the steps of the trace with respect to a given scenario as passed, failed or inconclusive. A trace in this context consists of the following elements:

- events representing observed properties of the monitored component (like events extracted from its input or output messages),
- events that represent changes in the context.

The operational semantics of the scenario language is defined by building one global finite automaton for the whole scenario. This automaton can be used as an observer automaton to categorize steps of an observed trace. The proposed semantics is based on [19], which in turn was inspired by the semantics defined for LSC [20].

In typical online monitoring setting, when the control component of a robot is monitored, the perception components send messages to the control component and in response the control component sends messages to the actuators. In other words, input and output events are carried by messages. If the interactions of the different components are represented by a message sequence chart then as relevant part only the lifeline of the monitored component shall be considered in order to extract the ordered sequence of events to be observed and evaluated by the automata. This concept is illustrated in Figure 31 (the monitored component is denoted by SUT).

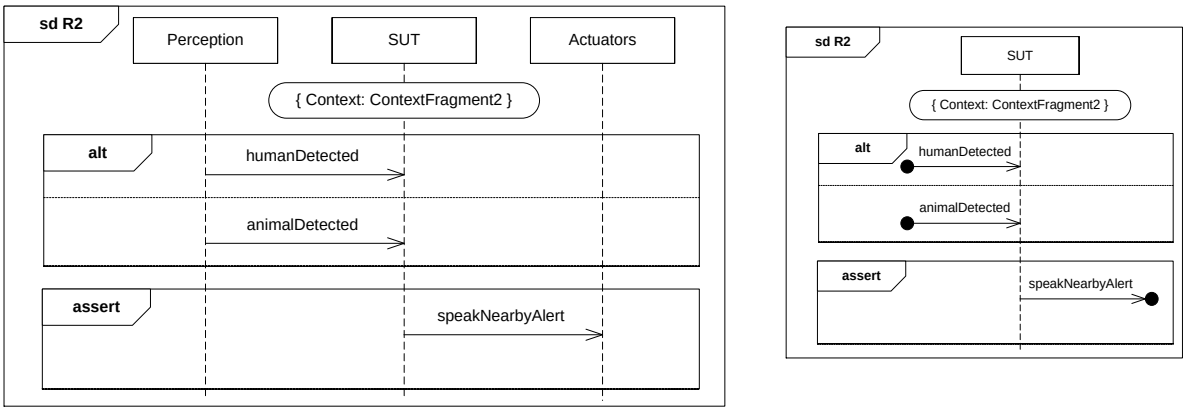

Figure 31. A complete sequence diagram and its relevant part

In Figure 31, as only the lifeline of the monitored component is retained, messages are replaced with standard UML *lost and found messages*. Because only one Lifeline remains in the scenario, several challenges found in other semantics for scenario languages (e.g., finding global synchronization points, non-local choices, etc.) are not to be solved here.

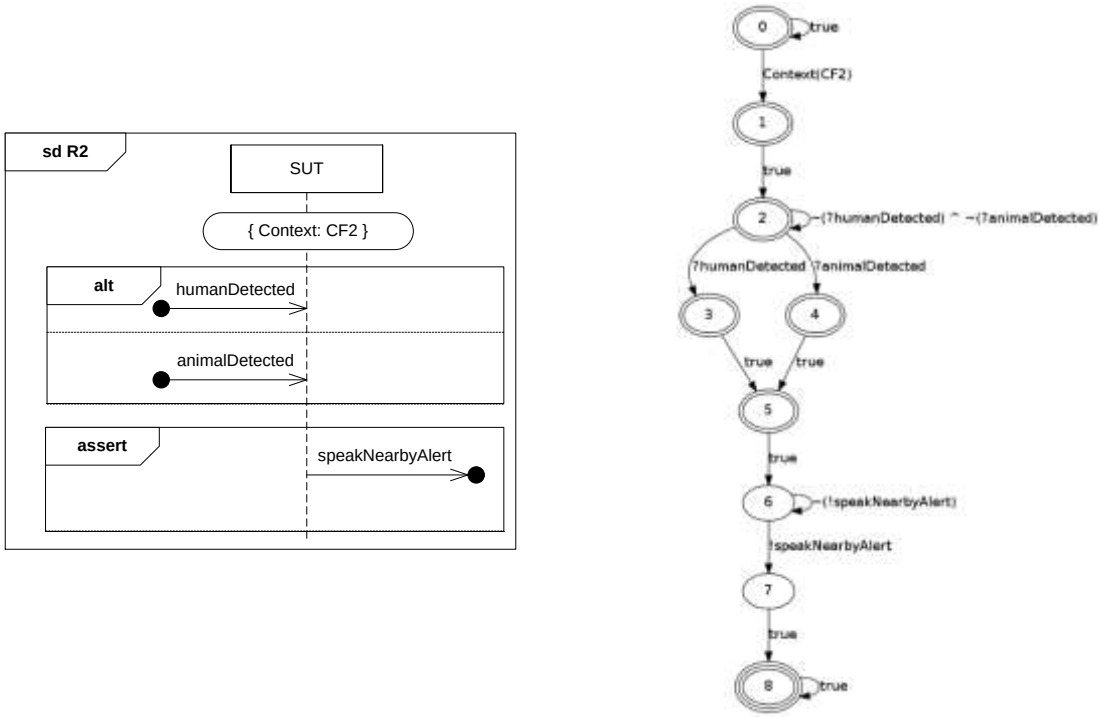

Figure 32. Example for generating the automaton

Figure 32 illustrates how an (observer) automaton can be generated for a given scenario. Each state of the automaton represents that a set of abstract events on the Lifeline has already been processed. Transitions are labelled with possible events; Context(**ContextName**) denotes a change in the context of the system (where a star as a name denotes a wildcard); while the symbol ? represents the receiving of a message (input event) and ! represents the sending of a message (output action). On the figures the character ~ is used for negation.

The automaton can be interpreted in the following way. State 0 is the initial state. When the system changes to a context, which is compatible (matching) with the one named *CF2*, then the automaton proceeds to State 1. The transition from State 1 to State 2 represents entering the *alt* CombinedFragment. As entering and exiting CombinedFragments are “silent” events, i.e., they are not real events, more precisely OccurrenceSpecifications in the original UML semantics, they are labelled with *true*, meaning that the automaton can advance to the next state without any external event in the trace.

As the requirement scenarios express only a partial behaviour, other messages can interleave with the ones depicted. This is represented by self-loops. For example, the self-loop on State 2 states that the monitored component can receive or send any message except receiving humanDetected or animalDetected. State 3 and State 4 represent different operands of an *alt* fragment, thus they are mutually exclusive. State 6 and State 7 belong to the *assert* fragment, while State 8 denotes that everything in the *assert* has also been observed in the actual trace.

One more thing is needed: to categorize steps, a verdict should be assigned based on where the automaton stays after processing a step. As there are three different possible verdicts, three types of states are distinguished:

- *trivial accept* (denoted with double circles): the state is still inside the trigger part of the scenario, error is not detected yet;
- *reject* (denoted with single circles): the state is inside the assert part, and when the trace is finished here an error is detected;
- *stringent accept* (denoted with triple circles): the state has successfully reached the end of the assert part.

After processing a trace, if the automaton stops in a trivial accept state, the verdict is inconclusive, while a reject state means a failed, a stringent accept implies a passed verdict.

As it can be seen from this example, the crucial part of the semantics is to define the relations between the abstract events, and identify in which fragment, and in which operand of the scenario language a given event is contained (this can be complicated if nested fragments are used). Thus, the construction of the automaton consists of the following two steps:

1. *Pre-processing the scenario*: the elements and orderings are identified by defining important locations, and searching for locations that happen at the same time.
2. *Unwinding to create the automaton*: this step creates an automaton from the pre-processed scenario by gradually unwinding it.

The rest of the section details the pre-processing and unwinding steps, which result in an automaton representing the scenario.

Sequence diagrams provide a rich set of features to define scenarios. Beside the basic elements, the following combined fragments are supported by our transformation:

- alt, opt fragment: alternative and optional fragments defining choices in the possible behaviours.
- loop fragment: parts of the scenarios which can be executed multiple times
- assert fragment: definition of the postcondition, the precondition is implicitly stated before the assertion
- par fragment: parallel interleaving behaviour

We also applied a syntactic restriction, so there must be an atom in every combined fragment. This restriction is used to avoid ambiguous monitor specifications which can be misleading.

4.1.1 Pre-processing the scenario

In the pre-processing step a so-called *unwinding structure* is constructed, which defines the important events in the scenario and their relations. Using the terminology of semantics defined for LSC, these events are called *atoms*, and every atom is assigned a position, i.e., a unique identifier that indicates the exact place of the atom.

*Definition 1 (Atom)***:** The following elements in a scenario are atoms: Lifeline heads and ends, MessageOccurenceSpecifications, StateInvariants (including context changes), entering or exiting a CombinedFragment and InteractionConstraints (guards in operands).

In some of the classic semantics for LSC, the position is simply an integer number. However, in UML SD the visual position of an atom does not indicate its ordering (e.g., for two events in separate operands of a *par* fragment, the one that is “lower” does not necessarily happen later). For this reason, a path expression is assigned as position.

Definition 2 (Position): The position has the form *[path]id*, where *path* is a string identifying in which CombinedFragment the atom is, and *id* is an integer giving the order of the atom compared to the other atoms inside that fragment. The string *path* is empty if the atom is in

the main fragment of the diagram, otherwise it is in the form *p.opr(op)*, where *p* is the position of the CombinedFragment the atom is in, *opr* is name of the operator of the fragment, and *op* is the number of the operand the atom is in.

Figure 33 presents an example to these concepts. The Lifeline's head, the two context changes, entering and exiting the *assert* fragment and finally the Lifeline's end are in the main fragment of the diagram, they are not contained in any CombinedFragment, thus they are assigned simply increasing integers as positions. This ordering reflects for example that the two context changes should happen before entering the assert fragment. The other atoms are inside the *assert* fragment, thus they inherit the path of the *assert* in their position, i.e., *4.assert(1)* signifies that the assert is in the main fragment of the scenario. The two message sending atoms are in different operands of the same *par* fragment, thus their path expression only differs in the operand number. With the help of the position expressions, the conflict and causality relations between the atoms can be defined.

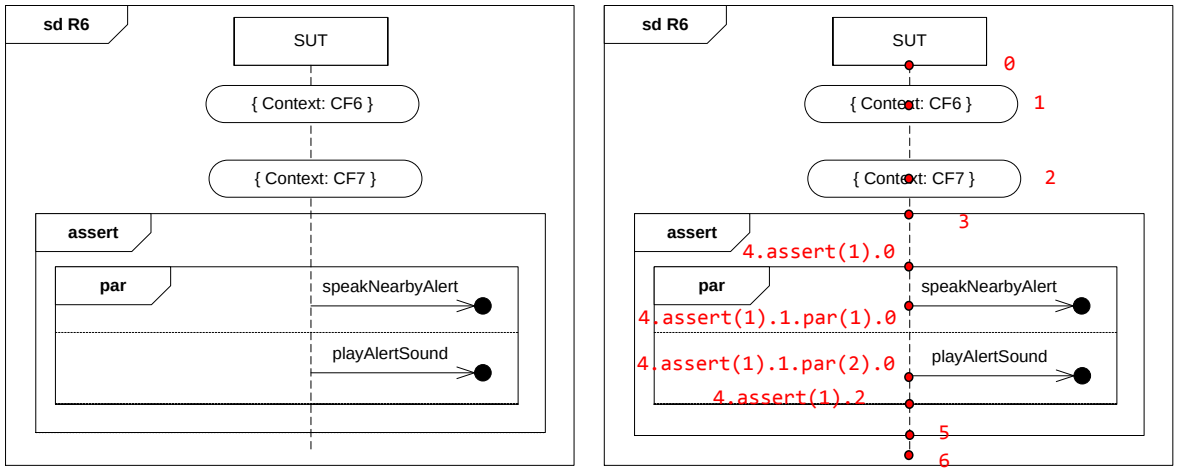

Figure 33. Assigning positions to atoms

Causality, denoted by $\prec$, defines a partial order between atoms. Intuitively, if two atoms are contained by the same fragment (either by the main fragment or by the same operand of a CombinedFragment), then their visual positions imply causality also.

Definition 3 (Causality): Let *a* and *b* be two atoms with their position in the form $p_1.i.p_2$ and $p_1.j.p_3$, where if p_1 is the empty string, then the $p_1.$ prefix, if p_2 or p_3 is the empty string, then the $.p_2$ or $.p_3$ postfix is removed respectively. Causality is defined then as

$a \prec b$ *iff* $j > i$

Using this definition in the example from Figure 33 results in orderings for example, that the context change to CF7 (position *2*) should happen before exiting the assert (position *4.assert(1).2***)** or that the two message sending atoms are not causally related.

Definition 4 (Predecessors): The predecessors function calculates the immediate predecessor(s) of an *a* atom:

$$Predecessors(a) \equiv \{ a' \in Atoms \mid a' \prec a \wedge !(\exists a'' \in Atoms : a' \prec a'' \prec a) \}$$

Note that, for example on Figure 33, the end of the *par* fragment (i.e., the atom with position *4.assert(1).2)* has two predecessors, the two message sending atoms inside the operands of the *par* fragment.

One more relation is needed to describe precisely the relationship between atoms, and that is conflict. By definition of the semantics, atoms from two different operands of the same *alt* fragment should not be observed in the same trace; this information is captured in the following definition.

Definition 5 (Conflict): Let *a* and *b* be two atoms with their position in the form $p_1.alt(i).p_2$ and $p_1.alt(j).p_3$, then conflict is defined as
$a \# b \text{ iff } i \neq j$

With the help of these definitions, an automaton representing the scenario can be constructed.

4.1.2 Unwinding to create the automaton

The unwinding step is much simpler in our case than in [19]. On one hand, there is only one Lifeline in the requirements, thus calculating the possible global cuts (coherent set of atoms from each Lifeline that represent a borderline of already processed events) is not necessary. Only *local cuts* need to be determined, and these cuts will form the states of the automaton. On the other hand, in [19] the construction of a symbolic automaton was needed, because the sending and receiving events of the same message have to be matched using message identifiers. This is also not necessary in our case, as the identity of the other communicating party is not relevant, thus either the sending or the receiving of a message is retained in the automaton.

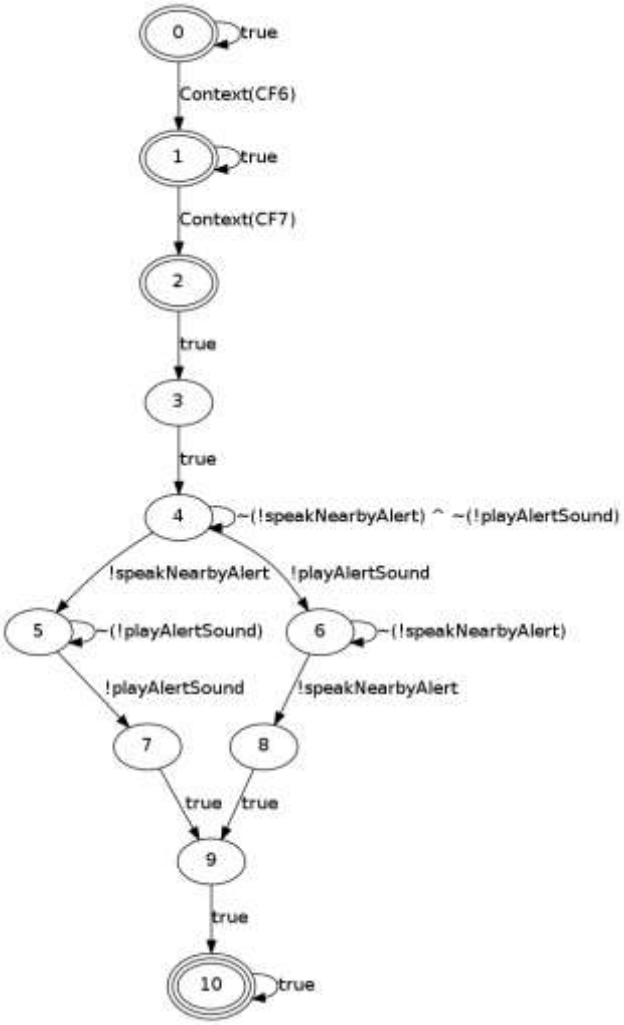

Figure 34. Automaton generated for the scenario on Figure 33

Accordingly, the automaton is constructed using the following steps.

- The states of the automaton correspond to the cuts of the scenario. The initial cut consists of only the monitored component Lifeline's head.

- New cuts are obtained by adding an atom to the previous cut. The Predecessors and Conflict relations are used to search for new, possible cuts: an atom can be added to a cut if all of its non-conflicting predecessors are already in the current cut. When a new cut is found, a corresponding new state and a transition going to the new state from the state representing the current cut are added to the automaton.
- The type of a state (trivial accept, reject, stringent accept) is set according to whether the unwinding has entered or exited the final assert or not.
- Transition labels are assigned depending on the processed atom.
    - If the atom represents sending the *m* message, the label will be *!m*, for receiving the *m* message, the label will be *?m*.
    - For a change to the *CF* context, the label will be *Context(CF).*
    - If the atom is entering or exiting a CombinedFragment, the label of the transition will be 'true'. These internal transitions with *true* guards are used as intermittent transitions that are fired together with their predecessors.
- If an operand of an *alt* or *opt* fragment has an InteractionConstraint, then the constraint is added as a guard to the transition that represents processing the constraint atom.
- Self-loops are added in the following cases to the states:
    - for states, where there is an outgoing transition labelled with a message event, a self-loop is added with a label obtained by the conjunction of the negated version of the events found in the outgoing transitions of the state (representing that other, non-specified messages can be ignored, but not the ones depicted in the original scenario),
    - for states, where there is an outgoing transition labelled with 'true', no self-loop is added (these states are intermediate states that represent silent events).
    - for every other states, a self-loop is added with a 'true' label, meaning that in these states other events, which are not relevant for the scenario, could occur.

This construction is illustrated in Figure 34, which is created for the scenario depicted on Figure 33. The two messages are in different operands of a *par* fragment, thus they can appear in any order; this is reflected in states 4–9.

The textual representation of the automaton is generated using a tool implementing the semantics defined in this section. It can be later visualized using the GraphViz package.

Using the semantics outlined in this section an automaton can be generated for each requirement scenario.

4.1.3 Extending the scenario with timing constraints

The monitor generation procedure of the former section has to be extended to handle timing constraints. This enables runtime monitoring of real-time requirements.

The proposed solution relies on the theoretical results providing finite abstractions of continuous time systems [21]. Timeout automaton serves as the underlying formal representation from which the monitor is generated.

The construction of the low-level representation for monitor synthesis consists of the following steps: The timed sequence chart formalism is transformed into a sequence chart formalism containing only untimed events. This syntactically restricted class of sequence charts are then processed by the formerly introduced algorithm and an observer automaton is constructed. This automaton is then extended with the timing information and transformed into the observer statechart formalism. The overall workflow is depicted on Figure 35.

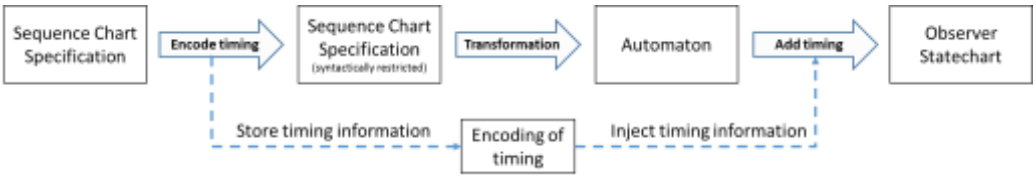

Figure 35. Extended workflow for timed sequence charts

In the following we introduce the algorithm that transforms timed sequence charts into the syntactically restricted fragment.

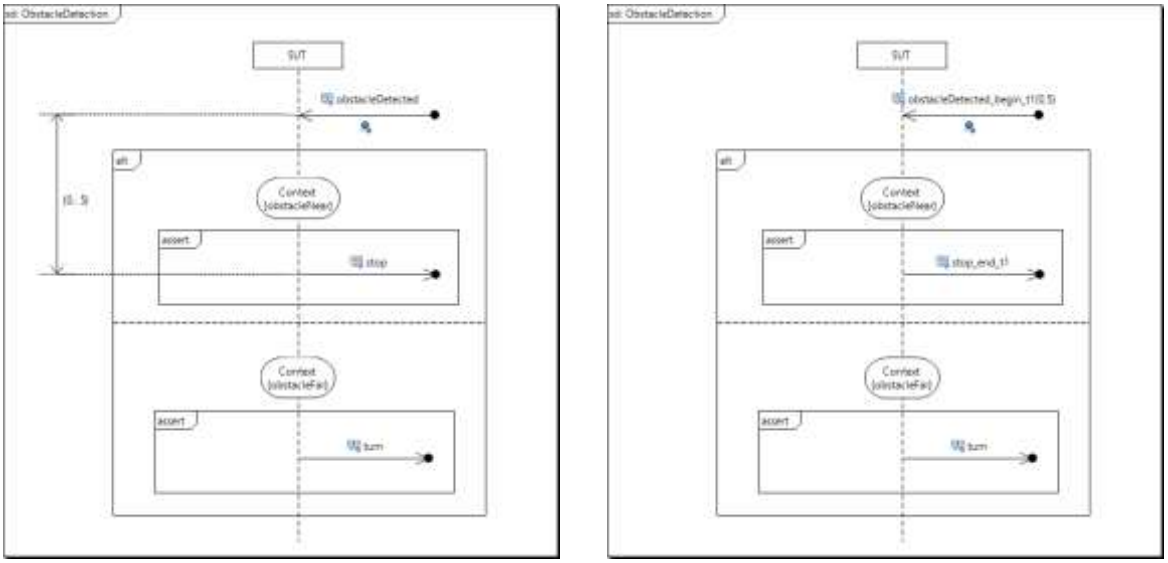

Figure 36. Example sequence chart specification with context and timing constraints

The transformation rules producing untimed sequence charts rewrite the sending and receiving events in a way that message exchange rules with timing constraints are eliminated. The starting point and also the endpoint will contain the relevant information about timing: the starting point of a time related event will encode the starting of a timer with the corresponding parameters. In our example in Figure 36 the starting point of timing is the event *obstacleDetected*, this way the renamed event is `obstacleDetected_begin_t1(0,5)`). The endpoint of the timing constraint in the example is the sending of the *stop* message: this will be renamed according to that timer which is invalidated by the message sending. The event corresponding to the sending of the stop message is renamed to `stop_end_t1` representing the information that in that time point the timer t1 will be invalidated.

From the untimed sequence chart representation the automata generation algorithm (see in the previous subsections) will generate the automata presented on the left side of Figure 37. In this figure, white circles mark states where the default behaviour (i.e. the transition taken on an undefined event) is to jump to the initial state, whereas states depicted as grey circles belong to the assertion, thus in this case the default behaviour is to jump to the error state.

The next step is then to extend the automaton with the timing information. The start of a timer is represented by an assignment to a related clock variable (e.g., *t1:=5*), timeout is represented by a specific event (e.g., *t1 over*) while the deactivation is similarly an event (e.g., *deactivate t1*). Three rules are defined to represent the timing information in the automaton and define the transition triggered by the timeout event. The rules are applied according to the position of the initialization and closing events of a time constraint:

1. The starting and closing event of the time constraint are in the condition part of the sequence chart: timeout event will lead back to the initial state.

2. The starting and closing event of the time constraint are in the assertion part of the sequence chart: timeout event will lead to the error state.
3. The time constraint starts in the condition part and it is finished in the assertion part of the sequence chart: pseudo state is introduced to memorize the occurrence of the timeout. In case the trace continues into the assertion part of the automaton, it will lead to the error state. If the trace will not continue to the assertion part then the trace is continued normally.

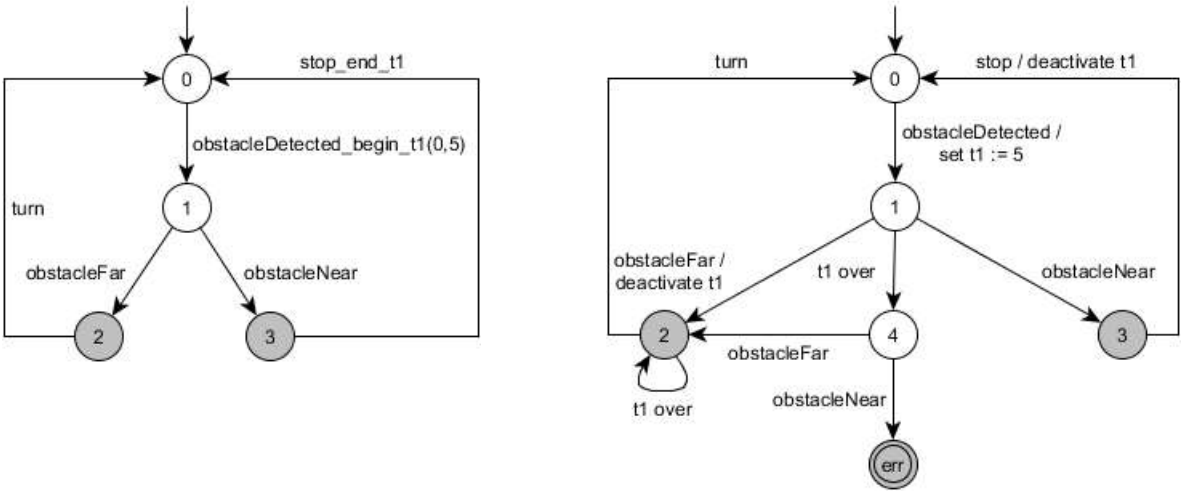

Figure 37. Automaton representation before and after the time transformation

The transformation from the automaton to the timed automaton representation is illustrated on the example of Figure 37. The automaton on the left side of the figure is the output of the automaton generation algorithm presented previously. In the example, the third rule is applied: as the time constraint connects an event in the condition part and another event in the assertion part. As it can be seen on the figure, the state number 4 is introduced to memorize if a timeout has happened. In case, when the trace would continue into the branch of the assertion with the time constraint, the trace will be declared erroneous (*err* state). If the trace continues to the branch without the time constraint, the trace will not mark erroneous and the runtime monitoring is continued.

4.2 Mapping from automata to observer statecharts

In this section we briefly overview how the automaton representation is transformed into the intermediate observer statechart formalism. The rationale is that in this way the same source code generation technique can be used as in case of statechart specifications (section 3.3).

The output of the algorithm presented in the former section is an automaton with timeout-based behaviours. Representing it in the observer statechart language is straightforward. We apply the following rules:

- An individual timer is declared for each time constraint.
- A timer is activated and set to a timeout value at the initial event of the time constraint.
- Timeout event triggers the transition corresponding to the transition of the automaton labelled with the timeout.
- Timer is deactivated if the corresponding action is present on the transition in the automaton.

In the following example, the observer statechart of the automaton of Figure 37. is presented. It starts with the pattern and signal declarations. As the property uses one time constraint, only one timer is declared in the monitor description (timer timeout). One state called err is labelled erroneous. The transitions are defined according to the rules introduced above.

```
specification ObstacleDetection {

	class Obstacle

	pattern obstacleNear(o : Obstacle)
	pattern obstacleFar(o : Obstacle)

	signal obstacleDetected
	signal stop
	signal turn

	statechart monitor := {
		// ...
	}
}
```

```
statechart monitor := {
	timeout timer

	region main {
		initial init
		state start
		state detected
		state over
		state matched_far
		state matched_near
		@ERROR state error

		transition from init to start
		transition from start to detected
			upon obstacleDetected / set timer := 5
		transition from start to start by default
		transition(o : Obstacle) from detected to matched_far
			case obstacleFar(o) / deactivate timer
		transition(o : Obstacle) from detected to matched_near
			case obstacleNear(o)
		transition from detected to detected
			upon obstacleDetected / set timer := 5
		transition from detected to over when timer
		transition from over to matched_far case obstacleFar(o)
		transition from over to error case obstacleNear(o)
		transition from over to start by default
		transition from detected to start
			by default / deactivate timer
		transition from matched_far to start upon turn
		transition from matched_far to error by default
	}
}
```

5 Monitoring on the basis of temporal specification

In this chapter the CaTL (Context-aware Timed Propositional Linear Temporal Logic) based monitor synthesis approach is described. The corresponding tool-chain (as introduced in Section 2.3) implements the following steps:

1. Pattern-based construction of temporal properties (see details in Section 5.1).
2. Mapping CaTL temporal logic expressions to so-called evaluation blocks (see in Section 5.2).
3. Synthesis of the source code of the monitor on the basis of the evaluation blocks (see in Section 5.3).

5.1 Pattern-based formalization of temporal properties

Requirements described in natural language are often imprecise and easy to misunderstand. However, using precise mathematical formalisms, like CaTL, has the risk that the expressions become complicated, so it is difficult to write and change them.

As described in D34.10, experience shows that safety requirements are typically based on patterns [6] [7]. Accordingly, a method was developed which helps to specify complex CaTL requirements by *composing and parameterizing requirement patterns*. The Pattern Composition Tool designed and implemented to support this method contains a collection of the most often used patterns and also provides the possibility to create and integrate custom patterns. Based on the defined formalism and the set of composition rules, it allows the representation, parameterization, and composition of the patterns in a graphical way and export the resulting requirement in CaTL format.

5.1.1 The patterns

The composition and parameterization of the following patterns (Figure 38) are supported:

Occurrence Patterns are used to express properties related to the existence or to the lack of existence of certain events in the pattern scope (here the scope can be global, before another event, after another event, or between another events). They have been classified into four subtypes:

- Absence, also known as “never”. The event will never occur within the scope.
- Universality, also known as henceforth. The event will always occur within the scope.
- Existence, also known as eventually. The event occurs at least once within the scope.
- Bounded existence. The event occurs a fixed number of times within the scope.

Order Patterns are used to express requirements related to pairs of events in the pattern scope.

- Precedence. P event has always to precede Q event within the scope.
- Response, also known as Follows or Leads-To. Event P has always to be followed by event Q within the scope.
- Chain Precedence. A sequence of Pi events has always to precede a sequence of Qi events within the scope. It can be regarded as a generalization of the Precedence pattern.
- Chain Response. A sequence of Pi events has always to be followed by a sequence of Qi events within the scope. It can be regarded as a generalization of the Response pattern.

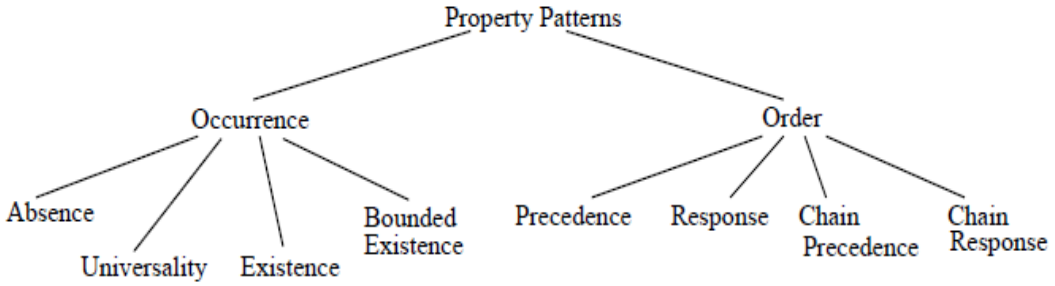

Figure 38. The classification of the temporal requirement patterns

To develop the Pattern Composition Tool, it was necessary to define a formalism to describe the patterns, work out a solution to parameterize them, and construct a set of rules for the composition of the patterns. These artefacts are summarized in the following.

The pattern formalism (represented by the EMF metamodel of the pattern language in Figure 39) consists of the following main parts:

- The representation of the Boolean (or, negation) and temporal (until, next) operators.
- The representation of the atomic formula that consist of atomic propositions (referring to system properties carried by events), timing constraints (referring to relations between clock variables and timing constants), context constraints (referring to context fragments and context operators that describe changes), and property constraints (referring to properties of context objects). An atomic formula with *hot* temperature is mandatory, while *cold* formulas are optional. Note that the semantics of these expressions is precisely described in the semantics of CaTL, the background temporal logic (see deliverable D34.10).
- The representation of the context. In Figure 39, the context (i.e., context fragments consisting of nodes and connections among them) are depicted as part of the pattern metamodel. Note that it can be specified separately and attached to the pattern metamodel.

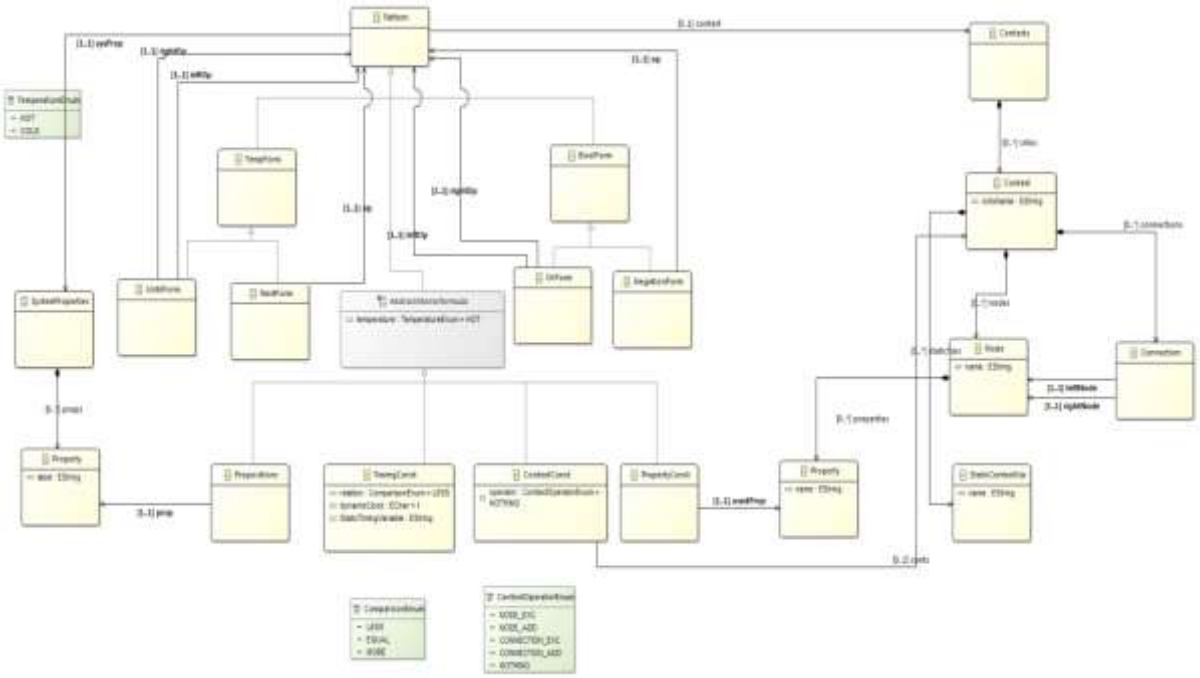

Figure 39. The metamodel of the pattern language

The main elements of the concrete (graphic) syntax that are used by the tool are depicted in Figure 40.

Modelled artefact	Metamodel element	Graphic representation	Example proposition
Next state	NextForm		
And operator	AndForm		
System property	Propositions		Disconnected
Timing constraint	TimingConst		$t < t_0 + 5$
Context constraint	ContextConst		$e_1 \sim e$
Object property	PropertyConst		e_1.a.speed < 10

Figure 40. The concrete syntax of the pattern language

As example, the representation of the property “After start, the next event is ‘Connected’ that is followed by the event ‘Disconnected’ in less than 5 time units, where speed of object ‘a’ is less than 10 in the ‘e1’ context” is given in Figure 41.

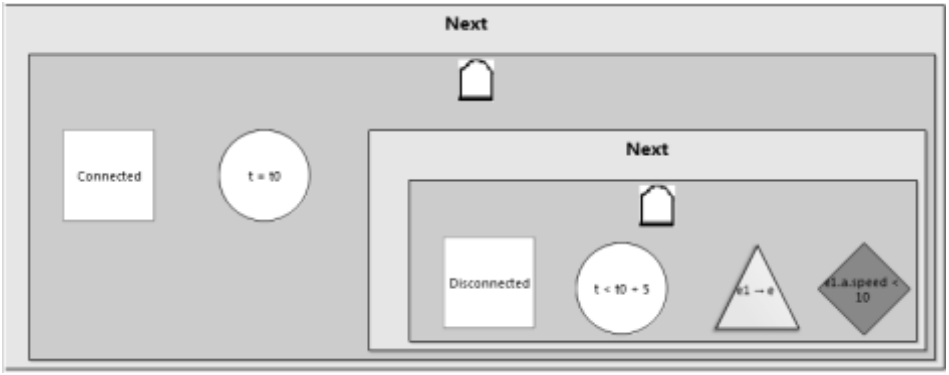

Figure 41. An example property constructed in the tool using the concrete syntax

The above mentioned categories of generic patterns as well as user-defined patterns (as extensions) are collected into a pattern store from which the patterns can be copied to the graphical editor and then configured (parameterized by giving the concrete names of events, properties, context fragments, timing constants etc.). The rules of composing and parameterizing patterns are determined by the syntax of the language (given by the metamodel above).

We defined a mapping from complex requirements (composed using the patterns and represented internally in the tool using the pattern language) to expressions of CaTL. This mapping is relatively straightforward as the pattern language followed the syntax and semantics of CaTL.

The textual CaTL representation belonging to the example presented in Figure 41 is the following:

X(connected and (t0=t) and X(disconnected and (t<t0+5) and (a.speed < 10) and e1~e)))

5.1.2 The workflow of pattern composition

The steps of pattern composition and CaTL expression generation are summarized in Figure 42. Note that the requirements that are composed from basic elements and patterns from the repository can also be stored as user-defined patterns for further use.

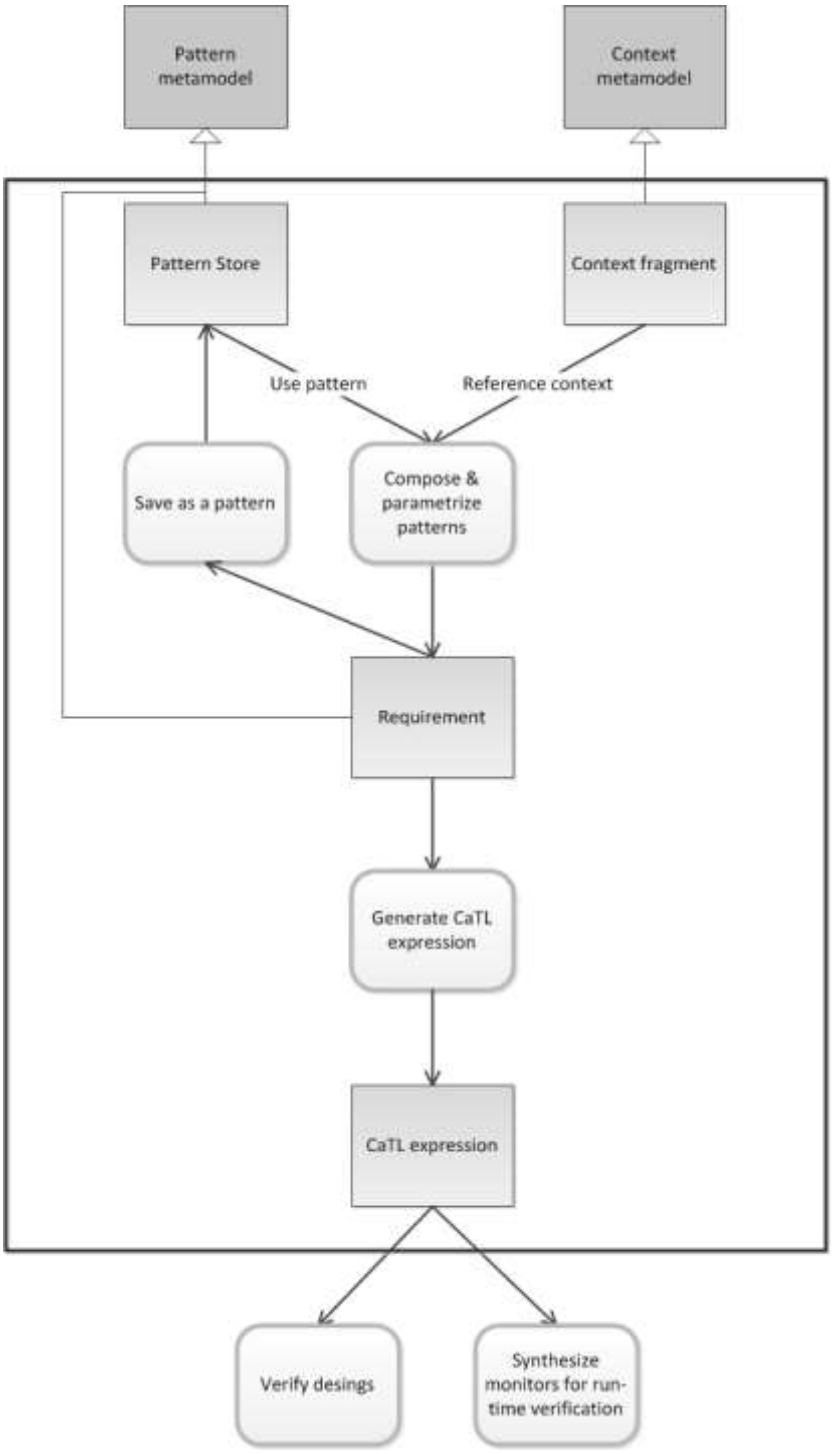

Figure 42. The steps of pattern composition and CaTL expression generation

5.2 Mapping from temporal logic (CaTL) to evaluation blocks

In this section first the use of CaTL is summarized on the basis of D34.10 then the tableau-based approach for runtime verification of CaTL properties (using so-called evaluation blocks) is presented. The so-called evaluation nodes of the tableau are implemented in the monitor.

5.2.1 Specification of properties using CaTL

CaTL, that is used to precisely express temporal properties for checking context-aware real-time behaviour, supports the following features:

- *Explicit context definitions*: Context may appear in the properties as condition for a given behaviour. For example, in an autonomous robot, in context of a nearby obstacle a slowdown command is required.
- *Timing*: Timing related criteria can be expressed. For example, the slowdown command shall occur in 10 time units after the observation of the nearby obstacle.
- *Modality*: Properties may define mandatory or optional behaviour.
- *Requirement activation*: Ordering between the required properties is supported. For example, violation of the slowdown property given above shall activate the subsequent monitoring of an emergency stop action.

In our approach this language may be used in two ways. First, it is available as a direct property specification language to formalize properties. Second, it can be used as an intermediate language, when its expressions are (1) constructed using patterns using the pattern composition tool (Section 5.1), or (2) mapped from scenario languages like extended Message Sequence Charts [1] [2] or Property Sequence Charts [3]. In any case, the resulting properties form the input of the monitor synthesis.

CaTL is an extension of the Propositional Linear Temporal Logic (PLTL) [8] that is particularly popular in runtime verification frameworks. PLTL expressions can be evaluated on a trace of steps, in which each step can be characterized by *atomic propositions,* i.e., local characteristics of the step. In our approach we call these atomic propositions in general as *events*, and the trace of steps is the *trace of events* (here events are considered as observations by the monitor, like a sent/received message, input/output signal, function call, started/expired timer, entered/left state, change of context, change of configuration, predicate on the value of a variable etc.). Besides the usual Boolean language operators, basic PLTL has the temporal operators X (next), U (until), G (globally) and F (future, eventually) that can be applied on events[6].

To be able to interpret this kind of PLTL expressions on finite traces, so-called *finitary semantics* is used that allows the evaluation of properties at any time. A three-valued logic with “true”, “false” and “inconclusive” values is applied, where the evaluation is “inconclusive” if (especially in case of a partial observation trace) no final verdict can be given. The semantics is impartial (a final verdict must not change when a new step is evaluated) and anticipatory (verdict is given as early as possible) [12].

To support the expression of context dependence and real timing in CaTL, we defined extensions of PLTL.

- Timing extensions: The basic PLTL cannot specify real-time requirements as it is interpreted over models which retain only the temporal ordering of the events (without

[6] In general, to handle data in the properties, PLTL can be combined with first-order logic and theories (e.g., to check sequence numbers, the theory of natural numbers with inequality) [11]. In this framework PLTL atomic propositions are substituted with first order data expressions that are evaluated with respect to a particular step (applying direct data processing to reason about data), and the temporal operators describe the ordering of these steps.

precise timing information). We apply the *Timeout based Extension of PLT* [9] that uses an explicit global clock (dynamic clock variable representing the current time) and static timing variables. Using this extension, the property "an alarm must be raised if the time difference between two successive steps is more than 5 time units" is expressed as *G((t0=t) implies X((t>t0+5) implies alarm))*, where *t* is the clock variable and *t0* is a timing variable.

- Context related extensions: In properties to be monitored, the contextual condition is referenced in the form of *context fragments* which are (partial) instance models of the context metamodel as described in Section 2.2.2. Context fragments are represented by static context variables, while the current observed context is represented by a dynamic variable. For example, a context fragment *e1* can specify that an instance of *LivingBeing* is in a *tooClose* relation with the *Robot* instance. The property "*it is always true, that if the component is connected and it is in the e1 context, and it will be disconnected in the next step, then eventually it will be in the e2 context*" is expressed as *G((connected and (e1~e) and X(disconnected)) implies F(e2~e))* where e is the variable representing the observed context.

Accordingly, the basic vocabulary of CaTL consists of a finite set of propositions (events), static timing variables and static context variables (these static variables are implicitly quantified with a universal quantifier). Moreover, two dynamic variables are used that represent the *current time* (clock variable t) and the *current observed context* (context variable e). The set of *atomic formulas* consists of the following elements:

- *Propositions* are events in the observed trace (each step may include multiple events). Each proposition can be evaluated to true or false in each step.
- *Property constraints* are predicates over properties of an observed context object.
- *Timing constraints* are defined as inequalities on the timing variables, constants and the dynamic clock variable.
- *Context constraints* are defined using a compatibility relation between context definitions and the current observed context. Context definitions can be created from context variables and operators as object exclusion, object addition, relation exclusion and relation addition. A context definition e1 is compatible with the current context e (denoted as e1 ~ e) if and only if there exists a bijective function between the two object sets e1 and e which assigns to each object in e1 a compatible object from e. Two objects are compatible, if and only if both have the same type and have the same relations to other objects.

To form CaTL expressions, these atomic formulas can be connected by using Boolean operators and PLTL temporal operators. Note that for each atomic formula, a modality can be assigned, where *hot* means a mandatory formula, and *cold* means an optional one.

In summary, PLTL atomic propositions are extended with context constraints (to be evaluated with respect to the observed context) and timing constraints (evaluated with respect to the current clock). These expressions are evaluated with respect to a particular step, without affecting the evaluation of the temporal operators. The precise semantics of CaTL is described in deliverable D34.10.

5.2.2 Tableau-based verification of CaTL properties

There are several approaches in the literature to synthesize specific monitoring code from PLTL specifications, based on rewriting (e.g., [14]) or automata theory (e.g., [15]). For the finitary semantics of the extended PLTL, the monitor can be synthesized as a finite-state machine [13]. The PLTL formula is mapped to a Büchi automaton with state labelling to give evaluation ("true", "false" or "inconclusive") in each step of the observed trace.

Another approach is an optimized tableau-based synthesis of monitors [10]. In this case a direct iterative decomposition of PLTL formulas is used. We apply this approach.

5.2.2.1 The use of evaluation blocks

The main idea is that the evaluation of a PLTL expression in a given step of the trace depends on (1) the events observed in a given step, evaluated on the basis of the Boolean operators on the events and the current-time part of temporal operators, and (2) the events observed in the future, related to the next-time part of temporal operators. Accordingly, the PLTL formula (with its temporal operators) is rewritten to separate sub-expressions that are evaluated in a given step (*current-time expressions*) from sub-expressions to be evaluated in the next step (*next-time expressions* with the X operator). On the basis of this rewriting, so-called *evaluation blocks* (EB) are constructed. The internal logic of an EB is the current-time expressions, where inputs are the events observed in the current step and the results of the next-time expressions. In the following, let us examine the construction of the evaluation blocks in more detail.

Let us consider the potential events as Boolean values in each step (i.e., the event given in the property is observed or not). If the evaluation of an expression depends on the next step, then the result of the evaluation in the next step is taken into account as an ? (unknown) result. This happens only if case of the X (next) or U (until) operators (note that using rules $F\ p = T\ U\ p$ and $G\ p = \neg(F\ (\neg p)) = \neg(T\ U\ (\neg p))$ the other temporal operators can be expressed using X and U). For handling situations like this, a three-valued logic is used, with the following values: T (true), ⊥ (false), and ? (unknown). The truth table are presented in Figure 43.

$a \wedge b$	⊤	⊥	?
⊤	⊤	⊥	?
⊥	⊥	⊥	⊥
?	?	⊥	?

$a \vee b$	⊤	⊥	?
⊤	⊤	⊤	⊤
⊥	⊤	⊥	?
?	⊤	?	?

a	$\neg a$
⊤	⊥
⊥	⊤
?	?

Figure 43. Truth tables for the ternary (three-valued) logic

Using ? (unknown) value as result of the next-time expressions the original formula is evaluated using the rules of the three-valued logic. If its result is ? then the evaluation is suspended for the current step (say s0) until the ? value can be resolved by the result of the next-time expression which is evaluated on the next step (s1) of the observation. The result of the evaluation of the next-time expression is returned to the evaluation for s0. Of course, it is possible that the evaluation for s1 also returns ?, so the evaluation must be continued over the subsequent step(s) recursively.

The evaluation of an expression on a step is carried out by an evaluation block (EB). The schematic view of an evaluation block is presented on Figure 44. It has 3 interfaces for input and output as follows:

- Result (output): Result of the evaluation of the expression.
- Current (input): The events (represented as Boolean truth values) observed in the current step.
- Next (input): If the evaluation depends on the result of the next step then the evaluation of the next-time expression is collected on this interface. Until the evaluation in the next step results in a Boolean value, an ? (unknown) value is considered here.

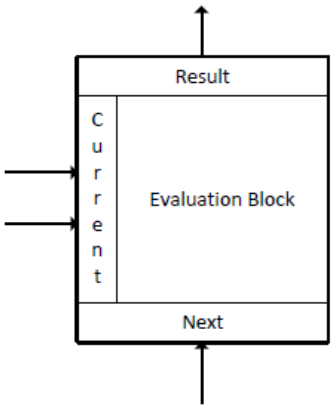

Figure 44. Ports of an evaluation block

New evaluation block is created (and connected to the Next interface) if an EB cannot decide to true or false based on the events of the current step. If a new EB is created then the formula to be evaluated by the next EB, i.e., the next-time expression is got as follows:

- In case of *Xp* the next-time expression is *p*, while the current-time expression is empty (there is no need of logic operation with the current events).
- In case of *p U q*, since $p\ U\ q = q \vee (p \wedge X\ (p\ U\ q))$, the next-time expression is *p U q*, while the current-time expression is $q \vee p$ which is connected to the Next interface with an $\wedge$ operator. With this solution the handling of the U operator is "postponed" to a succeeding EB.

The current-time expressions and the next-time expressions of the basic Boolean and temporal operators are collected in the following table.

Logic operator	**Current-time expression**	**Next-time expression**
$\neg p$	$\neg p$	-
$p \wedge q$	$p \wedge q$	-
X p	-	p
p U q	$q \vee p$	p U q

Table 1: Current-time expressions and next-time expressions

In Figure 45 an EB belonging to the expression G(r→(p U d)) is presented where r, p, and d are events and → denotes the 'implies' operator. The next-time expressions are at the bottom interface of the EB.

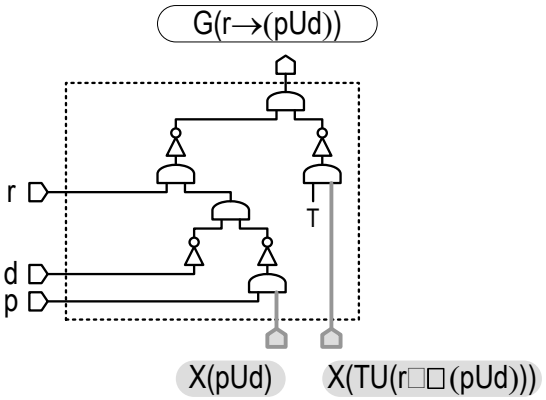

Figure 45. Example of an evaluation block

The next-time expression is the basis of forming the next EB. Accordingly, as the results of the next-time expressions come as the outputs of the EB belonging to the next step, a chain of EBs is formed. Because the evaluation is carried out on finite traces, at last at the end of the trace the X operator can be evaluated (to false), so there is an exit condition for this recursion. Moreover, in case of repeated observations no new EBs shall be created (as the evaluation of the expression does not change).

Let us consider a simple example showing how the evaluation blocks are used. Let's evaluate the *a U b* formula, where *a* and *b* are events that are represented by Boolean values in each step according to the current observation. Let the trace observed having two steps: in the first step *a* is true, but *b* is false, then in the second step both are true. As it was already mentioned, the *U* (until) operator can be rewritten as follows: $a\ U\ b = b \vee (a \wedge X\ (a\ U\ b))$. The separation of the current-time and next-time expressions results in an EB that is instantiated and evaluated in the following way:

1. The first instance of EB denoted as $EB_0(a\ U\ b)$ is created.
2. $EB_0(a\ U\ b)$ is evaluated having the events of the first step and considering *?* (unknown) as the value of the next-time expression (the value at the Next interface of $EB_0(a\ U\ b)$. The evaluation returns *?* (unknown), as $b \vee (a \wedge X\ (a\ U\ b)) = \bot \vee (T \wedge ?) = ?$ in this step. This way a new EB is necessary. The new EB has the same type since the next-time expression is *a U b*.
3. The second instance of EB denoted as $EB_1(a\ U\ b)$ is created.
4. In the second step, having the truth values belonging to the events, $EB_1(a\ U\ b)$ evaluates to *T* (true), as *b* is true.
5. *T* is returned to resolve *?* at the Next interface of $EB_0(a\ U\ b)$.
6. Having this value at the Next interface, the evaluation of $EB_0(a\ U\ b)$ results *T* as well.
7. The evaluation of the original *a U b* formula is ready, the final result is *T*.

The evaluation chain (i.e., the connected evaluation blocks) is depicted in Figure 46.

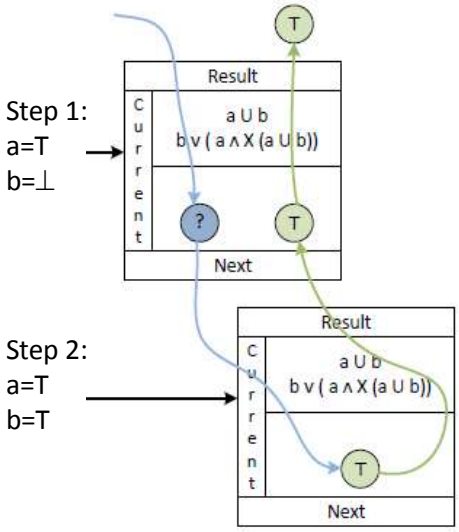

Figure 46. Example of an evaluation chain

The separation of current-time expressions and next-time expressions belonging to the original property expression in a recursive way provides a finite set[7] of evaluation block types. These block types can be considered as "templates" which are instantiated in the evaluation chain. The block types depend only on the property expression (and not from the trace to be evaluated), this way the construction of the EB types can be performed offline, during monitor synthesis.

5.2.2.2 Modality of expressions

The CaTL formalism allows defining cold atomic formulas that are written as <*af*>, where *af* is an atomic formula. For example, <*a*>∧*b* is a property expression, which will check whether *b* is true, if and only if *a* is already true. If *a* is false, but *b* is true, then the evaluation becomes inconclusive. Therefore *a* can be considered as a precondition of the satisfaction of the property.

The "inconclusive" evaluation means that the requirement is neither passed nor violated (this is definitely not the same as "unknown" in the three-valued logic). The inconclusive result is denoted as ∅ in the following. To handle ∅ as a value, the evaluation blocks must use four-valued logic instead of the three-valued logic, where the four values are: T (true), ⊥ (false), ? (unknown) and ∅ (inconclusive). Note that the chain of EBs does not give ? as a final result, because if the current-time expression results in ? then new EBs are created to be evaluated in the next steps until an exact value, i.e., one of T (true), ⊥ (false), or ∅ (inconclusive) can be given. The truth tables used for the evaluation of the current-time expressions are given in Figure 47.

$a \wedge b$	$\top$	$\bot$	?	$\varnothing$
$\top$	$\top$	$\bot$	?	$\varnothing$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
?	?	$\bot$	?	?
$\varnothing$	$\varnothing$	$\bot$	?	$\varnothing$

$a \vee b$	$\top$	$\bot$	?	$\varnothing$
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	?	$\bot$
?	$\top$	?	?	?
$\varnothing$	$\top$	$\bot$	?	$\varnothing$

a	$\neg a$
$\top$	$\bot$
$\bot$	$\top$
?	?
$\varnothing$	$\top$

Figure 47. Truth tables for the four-valued logic

[7] The set is finite as X operators in X p constructs are resolved in one step, while p U q constructs regenerate themselves. Accordingly, after a sufficient number of deriving steps the next-time expressions will be empty or they regenerate themselves. Precise proof is found in [10].

5.2.2.3 Evaluation of time and context related expressions

The handling of time and context require the adaptation of the evaluation block approach presented in the previous subsections. The handling of the current time and context is done through the clock and the context variable. At each EB, the clock variable will be substituted with the current time at the evaluated step and the context variable will be replaced with the current observed context.

However, the handling of static timing or context variables requires a bit more care. Two states of a variable must be distinguished: free (no value has been assigned) and bound (when it has a value). At first, all static variables are free. If a current-time expression of an EB contains a *var* = *value* atomic formula, where *var* is a free static variable, then *var* will be immutably bound to value (i.e., it has to be true in all succeeding EBs). For this reason two new ports are added to the EBs: the input and the output valuation port (see in Figure 48). The EBs receive previous valuations, which will be used at the local evaluation (the valuations cannot be altered, but new valuations can be added). When creating new EB for the succeeding state, the received set of valuations expanded with locally created valuations are forwarded to the next EB.

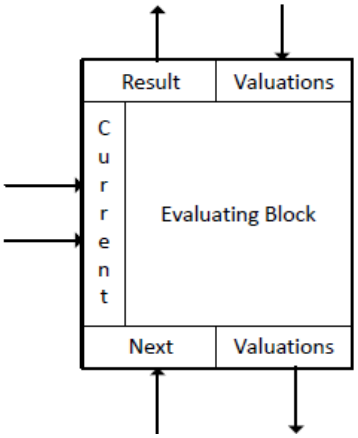

Figure 48. Interfaces of an evaluation block handling valuations

The evaluation of clock constraints with the just introduced valuations is relatively straightforward.

However, the efficient evaluation of context constraints is a more complex context matching task. A solution of the context matching problem is presented in Section 2.2.2. The general idea is to map the contexts to graphs consisting of vertices (labelled with the type and identifier of the object) and edges (labelled with the relation between the objects). The graph matching process may involve multiple context fragments (as the typical use case of monitoring addresses not only one, but several properties), thus the result of the simultaneous matching process is a set of valuations between the context fragments and the observed context. Before the matching is calculated, all context fragments (from all properties) are represented in a so-called *decomposition structure*, which is optimized for storing multiple similar graphs in a compact form by storing the common sub-graphs only once. The context matching algorithm based on this decomposition structure can efficiently search for all possible valuations between an observed context and the context fragments.

5.3 Monitor source code structure on the basis of evaluation blocks

On the basis of the concept of evaluation blocks and evaluation chain, the synthesis of monitor components includes the following steps:

- The construction of the finite number of EB types on the basis of the CaTL expression.
- Generating source code for the internal logic of EB types. Note that the hardware analogy presented in the figures of Section 5.2.2.1 is just a straightforward illustration of the idea; from software engineering point of view the logic circuits correspond to data types and evaluation functions. Similarly, valuations are handled by software variables.
- Generating an *execution context* that is responsible for receiving the events, runtime instantiation of the EB types and connecting these according to the evaluation chain defined by the next-time expressions.

Considering the evaluation of the current-time expressions in the EBs as functions, the evaluation of the formula *a U b* (see the example presented in Section 5.2.2.1) is demonstrated on the schematic sequence diagram in Figure 49. Here VerificationEngine represents the execution context; EB_0 and EB_1 are the two instances of the evaluation block; while s0 and s1 are the data structures that provide the truth values belonging to observed events in the two steps of the trace.

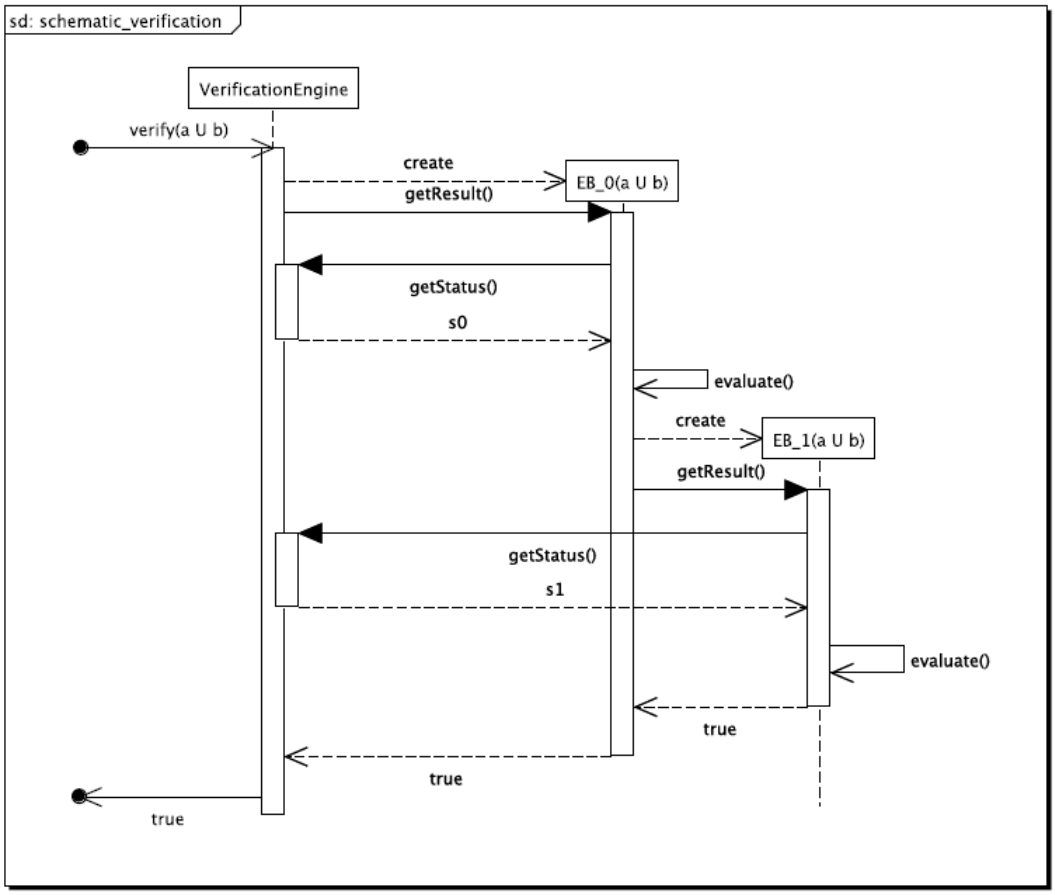

Figure 49. Evaluation of a temporal formula using two evaluation blocks

Memory needs of monitors are at most linear with the length of the trace. Repetition of blocks is reduced as the result of evaluation does not change if the same events are observed re-

petitively, this way it can be deduced that the memory needs are linear with the number of changes in the observed combinations of events in the trace.

On the basis of the structure of evaluation blocks, several source code variants can be generated. The most straightforward one is the direct object-oriented representation of evaluation blocks (with ternary logic and context related expressions). In case of resource-constrained systems a more 'compact' pure C-based monitor implementation would be useful. In the following we list some considerations regarding the code generation.

- The enumerated types (binary and ternary data types and the expression identifiers) are mapped to *enum* constructs or simple numeric constants can be used.
- The evaluation block types are to be mapped to programming language classes (C++, Java and C#) or *struct*s in C. The reference to the previous block can be implemented by a reference or a pointer.
- The "registers" on the block interfaces can be implemented by arrays addressed by enumeration identifiers. Storing the values of atomic expressions on the left interfaces require a single bit per expression while ternary values on the top and bottom interfaces need at least two bits per expression (storing bit patterns is supported by C bit-fields, the C++ bitset data type template, etc.).
- The procedures are either static methods of the related base classes or implemented by virtual member functions in derived classes. A key performance issue is the implementation of the ternary logic evaluation: these expressions can be translated to C++ preprocessor macros that are expanded into inline expressions and compiled to few machine code instructions by the compiler.

6 Monitor interfaces

In this section we overview the functionalities and interfaces of the monitor components.

6.1 Functionalities of the monitor

The monitor implements various functionalities that are summarized in this section.

1. Observing the behaviour of the monitored components in order to detect erroneous behaviour. The monitor receives *events* from the observed system. Each event represents some change in the observed system, for example sending a message, receiving a message, entering a state, leaving a state, changing the value of a variable etc. Events may have parameters. The monitor checks the allowed sequence of events on the basis of properties specified by a statechart model, by a sequence diagram, or by a temporal logic formula. This checking is implemented by the automatically generated *event evaluation function*.

Due to the diversity of the potential sources of events, the general idea is that manually written (typically short) *event forming functions* are used to identify events and send these to the monitor (by calling the event evaluation function with the event as call parameter). Examples of such event forming functions include:

- The function is registered to a relevant ROS topic (as a subscriber), extracts the event by processing the received message, and sends the related event to the monitor. E.g., event “CommandedToRight” represents that the command to turn right is sent to the actuator.
- The source code of a component is instrumented by inserting an extra function call when the state of the component changes (e.g., when an observed state variable is updated in a given code block). This function sends the related event to the monitor. E.g., event “DirectionToUp” represents that the direction variable is set to “Up”.

2. Invoking guard functions to check conditions related to the changes represented by events. When an event is received, the monitor may check external conditions to decide whether the event is allowed or erroneous. The condition can be specified in the requirements: as a guard function in a statechart, as a guard condition in a sequence diagram, or as a predicate in a temporal logic formula. These conditions are implemented as manually written *guard functions* that return Boolean (true or false) values. Examples of guard functions include:

- Checking values of global variables that are not involved in forming of events.
- Checking internal configuration parameters of the observed system (e.g., to conclude that it is realistic to observe the checked event given the current configuration of the system).
- Checking concrete parameters of the external context of the observed system (e.g., to conclude that an event representing a command is not allowed in the given context to behave safely).

3. Checking the progress of time in order to evaluate timing related constraints in case of incoming events. The timing related constraints can be specified in the requirements as follows: timeout for being in a given state of a statechart after receiving an event, timeout between events in a sequence diagram, or relative timeout in a temporal formula. These checks are based on a timeout mechanism that requires platform functions in the observed system as follows:

- Setting a timeout (activating an alarm event relative to the current time).
- Cancelling a timeout (deactivating the alarm).
- Receiving an alarm as a specific timeout event when the alarm interval has passed.

4. Invoking error processing function when error is detected in the behaviour of the observed system. The processing of errors may be a complex functionality, this way the general idea is to implement processing in a manually *written error processing function*. It is called by the monitor in the following cases:

- The event received by the monitor is considered erroneous as it is not a valid successor of the previous event.
- In case of a statechart requirement model: A state is reached that is annotated as erroneous in the model.
- In case of a statechart requirement model: A transition is executed that is annotated as erroneous in the model.

The error processing function has two events as parameters: the last event that was accepted by the monitor and the current event that is detected to be erroneous by the monitor (or that caused to reach the erroneous state or transition).

6.2 Interfaces of the monitor component

According to the above mentioned functionalities, the monitor component consists of the following functions:

- Event evaluation function (automatically generated from the property specification).
- Event forming functions (written manually).
- Guard functions (written manually).
- Timeout related functions (written manually).
- Error processing function (written manually).

The automatically generated event evaluation function (simply called in this section as monitor) interacts with the manually written functions on the following interfaces:

- *Event interface*: Provided by the monitor to receive and evaluate events.
- *Guard interface*: Used by the monitor to call guard functions.
- *Timer interface*: Used by the monitor to set, cancel, and receive timeout events.
- *Error interface*: Used by the monitor to call the error processing function.

The integration of these functions is illustrated in Figure 50.

In summary, the generated monitor is configurable as it can be coupled with arbitrary event forming functions, guard functions, and platform-dependent timeout functions. Similarly, user defined error handling functions define the reaction of the monitor to the detected errors.

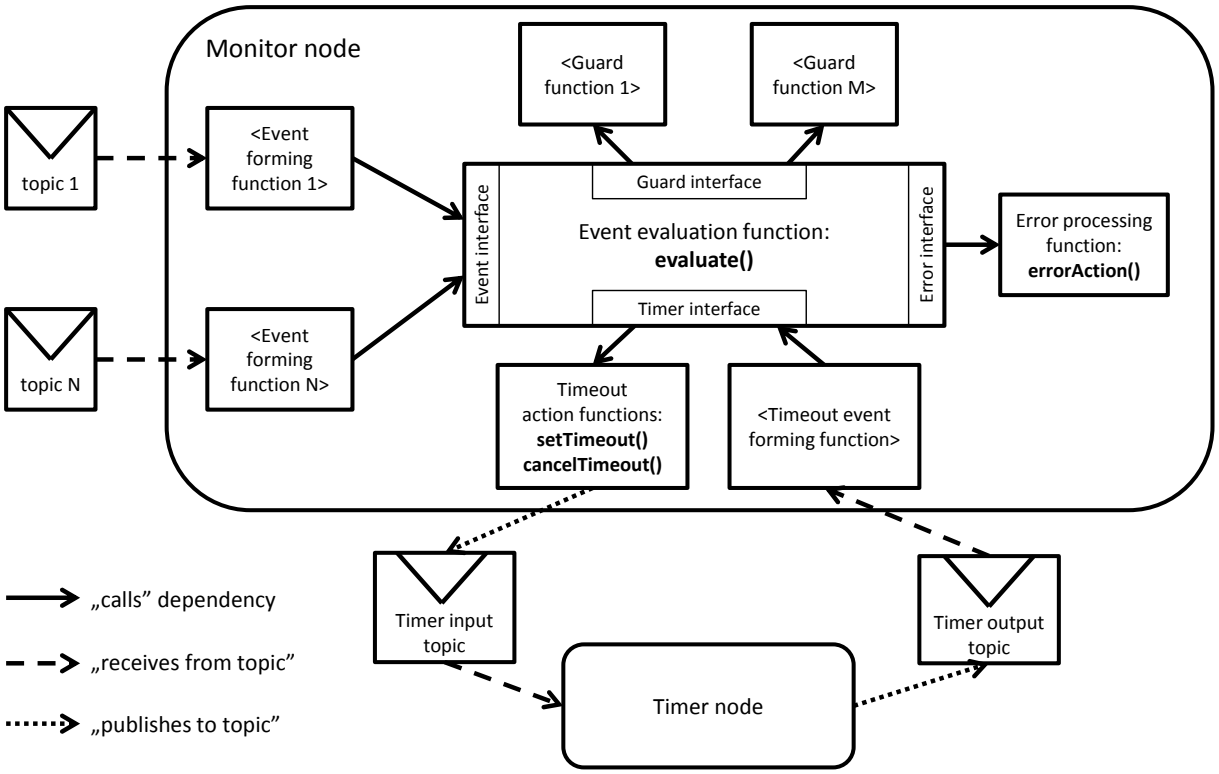

Figure 50. The monitor functions and interfaces

6.3 Implementation of the interfaces

An implementation of the functions and interfaces is presented in the following subsections.

6.3.1 The Event interface

Events are identified by their names as used in the property specification (i.e., in the requirement model), e.g., "DirectionToUp". Events may have an associated parameter which is an integer value (e.g., "SetSpeed(100)" used in the requirement model is handled as an event with name "SetSpeed" and parameter 100). The *event forming functions* pass these events to the *event evaluation function*.

The event evaluation function *evaluate()* can be called by the event forming functions in two forms. An event without parameter is passed as its name string, while an event with a parameter is passed as its name string and parameter value integer. The signatures are the following:

```
// Evaluating an event without parameter
void evaluate(const char* eventName);

// Evaluating an event with a parameter
void evaluate(const char* eventName, int eventParameter);
```

Examples for calling the monitor (by an event evaluation function):

```
// Evaluating the DirectionToUp event
evaluate("DirectionToUp");

// Evaluating the SetSpeed(100) parameterized event
evaluate("SetSpeed", 100);
```

If the events are represented by integers then as alternative an integer based version of the *evaluate()* function can be used:

```
// Evaluating an event represented by an integer
void evaluate(int eventNum);

// Evaluating an event with a parameter
void evaluate(int eventNum, int eventParameter);
```

6.3.2 The Guard interface

Guard conditions are to be implemented in the form of C++ functions that are called by the monitor and return Boolean values. The name of a guard function and its parameters are specified in the requirement (e.g., in the statechart model).

For example, when the statechart model includes the guard function *checkConfigurationFull()* then the signature of the related function that shall be implemented is the following:

```
// Guard function to check the configuration
bool checkConfigurationFull();
```

6.3.3 The Timer interface

The implementation of the timer functionality depends on the concrete functions or API available on the platform. A possible implementation in ROS is to have a *timer node* that receives timeout activation and cancel messages on a *timer input* topic and sends an alarm message on a *timer output* topic.

The monitor calls the following timer action functions to activate or cancel an (expected) timeout event. The body of these functions shall be implemented manually.

```
// Activation of a timeout (asking for timeout event)
void setTimeout(const char* eventName, int timeoutLength);

// Cancelling a timeout that has already been set
void cancelTimeout(const char* eventName);
```

Example for activating a timeout event “StartTimeout” for 200 milliseconds:

```
setTimeout("StartTimeout", 200);
```

The timeout shall be passed to the monitor in a similar way as the other events; the name of the event shall be the same as set in the *setTimeout()* function. The timeout event forming function is implemented manually (similarly to the event forming functions). This function shall call the *evaluate()* function of the monitor to pass the timeout event.

```
// Evaluate the timeout event StartTimeout
evaluate("StartTimeout");
```

6.3.4 The Error interface

The error processing function *errorAction()* is to be written manually. It is called by the monitor in case of detecting an error, with parameters containing the currently detected erroneous event and the last accepted event.

```
void errorAction(const char* current, const char* lastAccepted);
```

6.4 The file structure

The monitor functions are located in the following files (the file names are in accordance with a requirement model called <reqName>, for example in a *<reqName>.statechart* file):

- **<reqName>.cpp**: This file is automatically generated and contains the implementation of the *evaluate()* function on the basis of the <reqName> requirement model.
- **<reqName>_out.h**: This file is included by *<reqName>.cpp*. It shall contain the following manually written functions (that are called by the *evaluate()* function):
  - errorAction()
  - setTimeout() – if timeout is included in the requirement model.
  - cancelTimeout() – if timeout is included in the requirement model.
  - Guard functions – with names and parameters exactly as given in the guard conditions in the requirement model.
- **<reqName>_in.h**: This header file is to be included by the .cpp file that contains the manually written event forming functions. It contains the declaration of the *evaluate()* function (accordingly, this is a file with fixed content that should not be modified manually).

6.5 Example: Monitoring the Turtlesim node

The interfacing of the monitor is demonstrated using the classic ROS Turtlesim[8] node. Note that the following code excerpts are intended to help understanding the role of the interface functions and do not form a complete source code.

The monitor checks the behaviour of the *Turtlesim* node by observing the commands received by *Turtlesim* from a *Controller* node on the topic *cmd_vel* (Figure 51). The events that are checked represent the direction of the movement of the turtle: “Up”, “Down”, “Left” and “Right”.

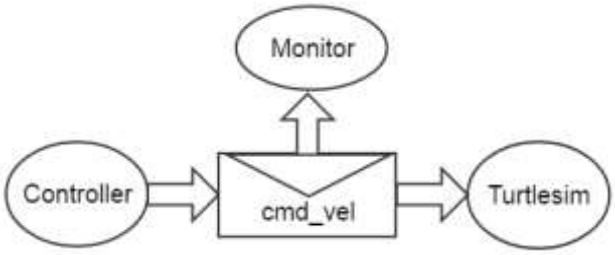

Figure 51. Monitoring the Turtlesim node

The monitor node consists of the following functions:

Event forming function: The event forming function *msgTurtleHandler()* is subscribed to the *cmd_vel* topic:

```
ros::NodeHandle nh;
ros::Subscriber subTurtle =
     nh.subscribe("turtle1/cmd_vel", 1000, &msgTurtleHandler);
```

On the basis of the *geometry_msgs::Twist* messages received on the *cmd_vel* topic, events are formed and passed to the *evaluate()* function by the *msgTurtleHandler()* function in the following way:

```
#define Up      1
#define Right   2
#define Down    3
#define Left    4

int dir = Right;// initial direction
```

[8] http://wiki.ros.org/turtlesim

```
void msgTurtleHandler(const geometry_msgs::Twist& msg){
     if (msg.angular.z != 0){
          if (msg.angular.z == -M_PI/2){
               switch(dir){
                    case Right: dir = Down;
                         break;
                    case Down: dir = Left;
                         break;
                    case Left: dir = Up;
                         break;
                    case Up: dir = Right;
                         break;
               }
          } else if (msg.angular.z == M_PI/2){
               switch(dir){
                    case Right: dir = Up;
                         break;
                    case Down: dir = Right;
                         break;
                    case Left: dir = Down;
                         break;
                    case Up: dir = Left;
                         break;
               }
          }
     } else { //it is an event for the monitor
          switch(dir){
               case Right: evaluate("Right");
                    break;
               case Down: evaluate("Down");
                    break;
               case Left: evaluate("Left");
                    break;
               case Up: evaluate("Up");
                    break;
          }
     }
}
```

As it turns out, the event forming function calls the *evaluate()* function by providing the new direction as the checked event.

Event evaluation function (monitor): The requirement "*the sequence of Up events is always directly followed by a Right event*" is specified using the statechart model in Figure 52. Note that an error is detected if a Left or Down event is received in state UpReceived (i.e., after a sequence of Up events) since only Up and Right events are considered in the model as allowed events in this state. (The "Event." prefix is used in the Yakindu statechart model as the name of the event interface and can be ignored.)

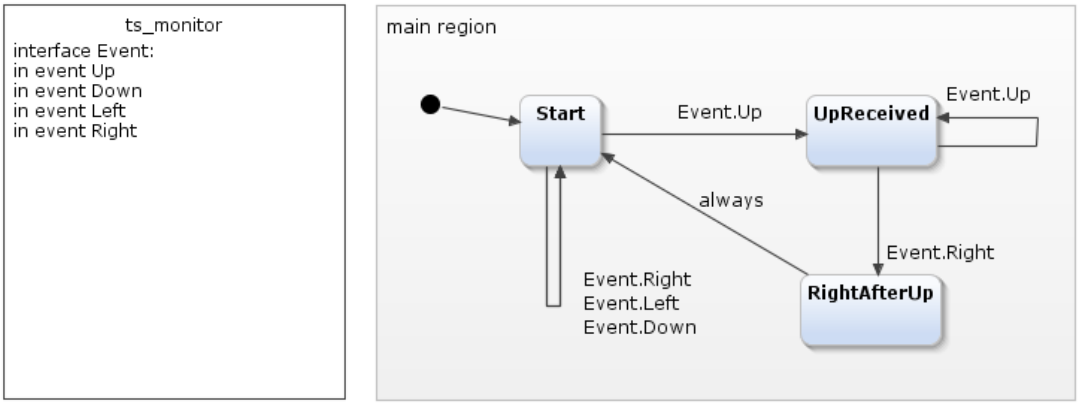

Figure 52. The statechart model belonging to the checked property

One could implement manually an event evaluation function *evaluate()* as follows:

```
int state = 0;
void evaluate(const char* event){
      switch(state){
            case 0: // Start
                  if (strcmp(event, "Up")==0) state = 1;
                  break;
            case 1: // UpReceived
                  if (strcmp(event, "Up")==0) state = 1;
                  else if (strcmp(event, "Right")==0) state = 2;
                  else if (strcmp(event, "Left")==0)
                        errorAction(event, lastEvent);
                  else if (strcmp(event, "Down")==0)
                        errorAction(event, lastEvent);
                  break;
      }
      if (state == 2){  // RightAfterUp
            state = 0;
      }
      lastEvent = event;
}
```

The *evaluate()* function that is generated automatically from the statechart requirement model is more complex as it implements a general and systematic way of evaluating the events:

- Thread-safe event queue (optionally, for handling multiple events),
- Handling multiple timeouts,
- Implementation of the detailed statechart semantics with concurrent regions, fork and join transitions, etc.).

Note that the generated code can be simplified in case of introducing restrictions regarding the occurrence of multiple events / timeouts and the syntax of the requirement model.

The main functions that are found in the code generated from the statechart model include

- *evaluate()* function for receiving events;
- *takeStep()* function for handling state transitions on the basis of the statechart model,
- *fireTransition()* function to implement a single state transition in the monitor.

As an excerpt, the *evaluate()* function is presented (that calls *takeStep()* which then calls *fireTransition()*):

```
void evaluate(const char* event) {
  if(!initialized) {
    initialized = true;
    init_statechart();
  }
  int i = 0;
  while(i < sizeof(events)) {
    if(strcmp(event, events [i])==0) {
      lastProcessed = currentlyProcessed;
      currentlyProcessed = i;
      pushIntoQue(outerQue, Event (i));
      break;
    }
    ++i;
  }
  takeStep();
}
```

Error processing function: The event evaluation function *evaluate()* calls the following simple error processing function *errorAction()*:

```
void errorAction(const char* current, const char* lastAccepted){
     ROS_INFO_STREAM("Error is detected.”);
}
```

Timer functions: In this example, let us define a timeout in the requirement statechart model in the following way (Figure 53): in the Start state, if there is no event received in *StartTO* time (set as 100 milliseconds) then the Timeout state is reached (this state is annotated as an error state this way the *errorAction()* function is called when this state is entered).

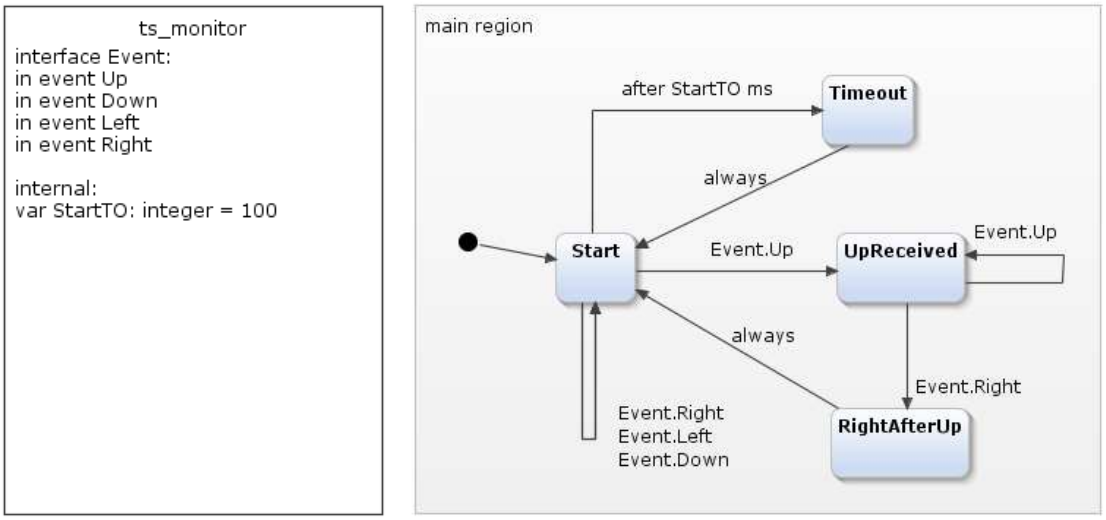

Figure 53. The requirement statechart model with a timeout

When the *evaluate()* function is called, then internally in this function the state transitions are traversed on the basis of the statechart model. Transitions that reach the Start state set the timeout by calling the *setTimeout()* function:

```
// Setting the timeout
setTimeout("StartTO", 100);
```

Transitions leaving the Start state (except the transition triggered by the “after StartTO ms” trigger) cancel the timeout by calling the *cancelTimeout()* function:

```
// Cancelling the timeout
cancelTimeout("StartTO");
```

The “StartTO” event is to be passed to the *evaluate()* function by a timeout event forming function. In this example, it is the *timeoutCallback()* that is able to handle several timeout events.

```
void timeoutCallback(const monitoring_comp::TimeoutPub::ConstPtr&
timeout) {
  for (auto i = timeout.events.begin(); i < timeout.events.end();
++i){
    evaluate((*i));
  }
}
```

As discussed in Section 6.2, the interface functions *setTimeout()*, *cancelTimeout()* and *timeoutCallback()* shall be implemented manually.

Let us assume that a ROS timer node is used. It can be reached using the topics *timeout_request* (as *timer output topic* from the point of view of the monitor, see Figure 50) and *timeout_status* (as *timer input topic* from the point of view of the monitor).

Accordingly, the functions *setTimeout()* and *cancelTimeout()* publish to *timeout_request*, while the *timeoutCallback()* is subscribed to *timeout_status.*

```
ros::init(argc, argv, "monitoring_component");
ros::NodeHandle n;
nodeHandle = &n;
ros::Publisher outChan =
     n.advertise<monitoring_comp::TimeoutSub>("/timeout_request",
1000);
publisher = &outChan;

ros::Subscriber sub =
     n.subscribe("/timeout_status", 1000, timeoutCallback);

void setTimeout(const char* eventName, int timeoutLength) {
  std_msgs::TimeoutSub msg;
  msg.event = String(eventName);
  msg.nsec = timeoutLength * 1000 * 1000;
  (*publisher).publish(msg);
};

void cancelTimeout(const char* eventName) {
  std_msgs::TimeoutSub msg;
  msg.event = String(eventName);
  msg.nsec = -1;
  (*publisher).publish(msg);
}
```

7 The usage of the monitor synthesis tool-chains

In this section we outline the usage of the developed monitor synthesis tool-chains.

As common technology background behind the processing of requirement models, the Eclipse technologies can be mentioned. The Yakindu Statechart Tools, the Papyrus sequence diagram editor and the Sirius based Temporal Pattern Composition Tool have an Eclipse Modeling Framework (EMF) based model representation. In addition, the developed intermediate language (that is an XText based language for textual editing purposes) also has an EMF representation. Figure 54 summarizes the basic technologies used for model processing. The outputs of model processing (like the intermediate statechart model in Figure 54) are used by the code generator tools to produce the monitor code.

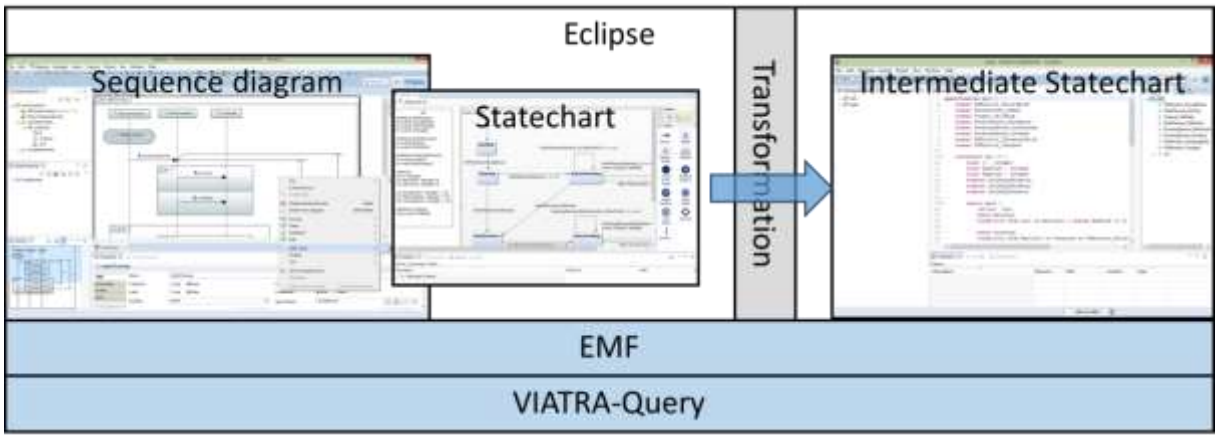

Figure 54. Technology overview

7.1 Monitor synthesis on the basis of behaviour specification

The requirement model is constructed in the Yakindu Statechart Tools. It consists of a graphical statechart model and the definition of its interfaces (note that these are not the monitor interfaces but the declaration of the input and output elements that are included in the statechart). The interface definition consists of the following parts:

- The Event interface that contains the events processed by the statechart,
- The Action interface that specifies actions (if any),
- The Guard interface that specifies guard functions (if any),
- The Internal interface that defines the specification variables and the parameters (e.g., timeout values).

An example of these interfaces in presented in Figure 55.

```
Monitor
interface Event:
in event ShouldDock
in event NoWay
in event PathExist
in event Docked
in event NotDocked
in event Charged
in event ChargingFails

internal:
var c: integer
var MaxTryP: integer=3
var MaxTryD: integer=3
var DockingTO: integer = 10
var PlanningTO: integer = 10
var ChargingTO: integer = 10
```

Figure 55. Interfaces for a statechart requirement model

The graphical statechart as requirement model is constructed using the standard elements of the statechart formalism (see the palette of elements on the right in Figure 56).

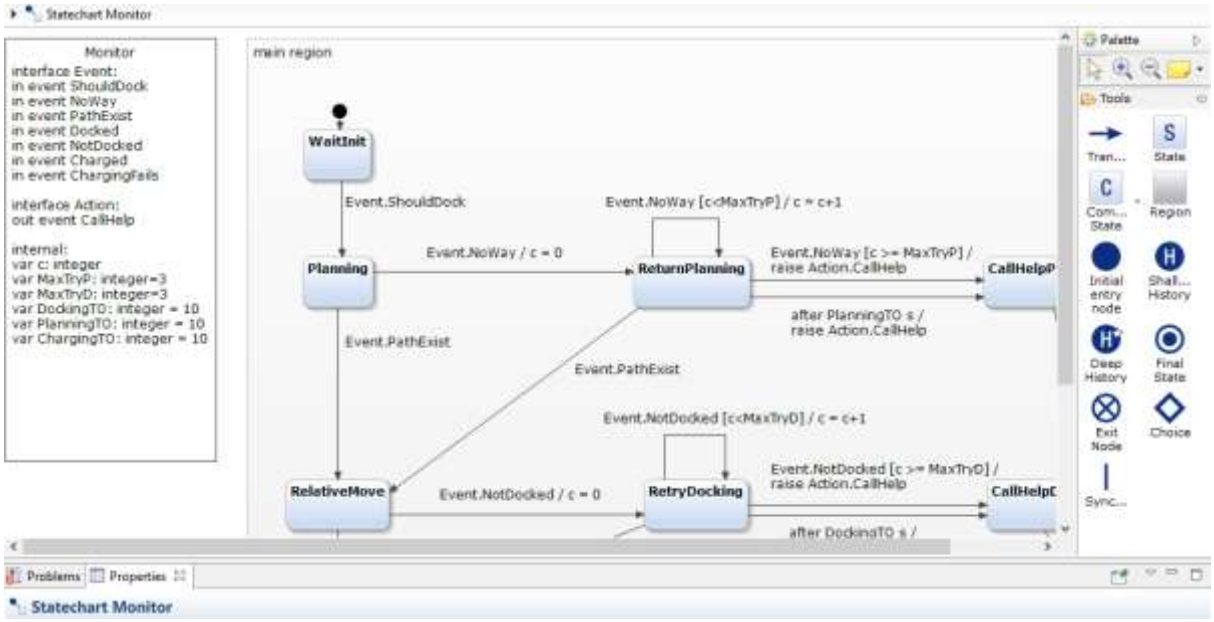

Figure 56. Construction of a statechart requirement model

The graphical statechart model is mapped to the textual intermediate representation that is displayed as coloured code. It is also possible to assemble the requirement model directly in the textual format if the designer prefers it.

Having the textual intermediate model, the monitor source code can be generated. The related command is integrated into the context menu of the Eclipse toolset (Tool/Export to Cpp, see in Figure 57).

The generated source code is available as a file structure described in Section 6.4.

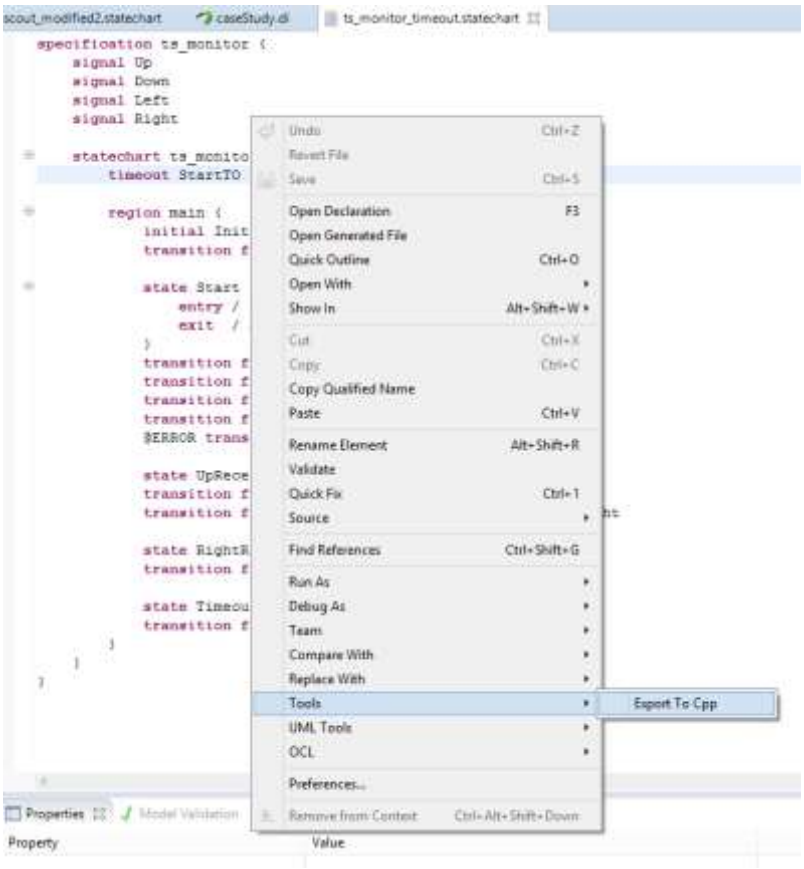

Figure 57. Generation of monitor code from the intermediate language

The statechart requirement model may include guard functions that evaluate context dependency by matching predefined context fragments with the actual context perceived by the robot application. It is assumed that the context fragments (from the requirements) as well as the context model are represented in C++ data structures generated from an EMF model (in case of the context model, this data structure is to be updated on the basis of the perception). Accordingly, a separate EMF based graphical editor is provided to construct the context fragments and context models (in the form of EMF class diagrams) and generate the corresponding data structures and the source code of the guard functions that perform the matching.

As depicted in Figure 58, this code generator is integrated into the context menu of the Papyrus EMF class diagram editor (xtUmlrt Generator/Generate C++ files).

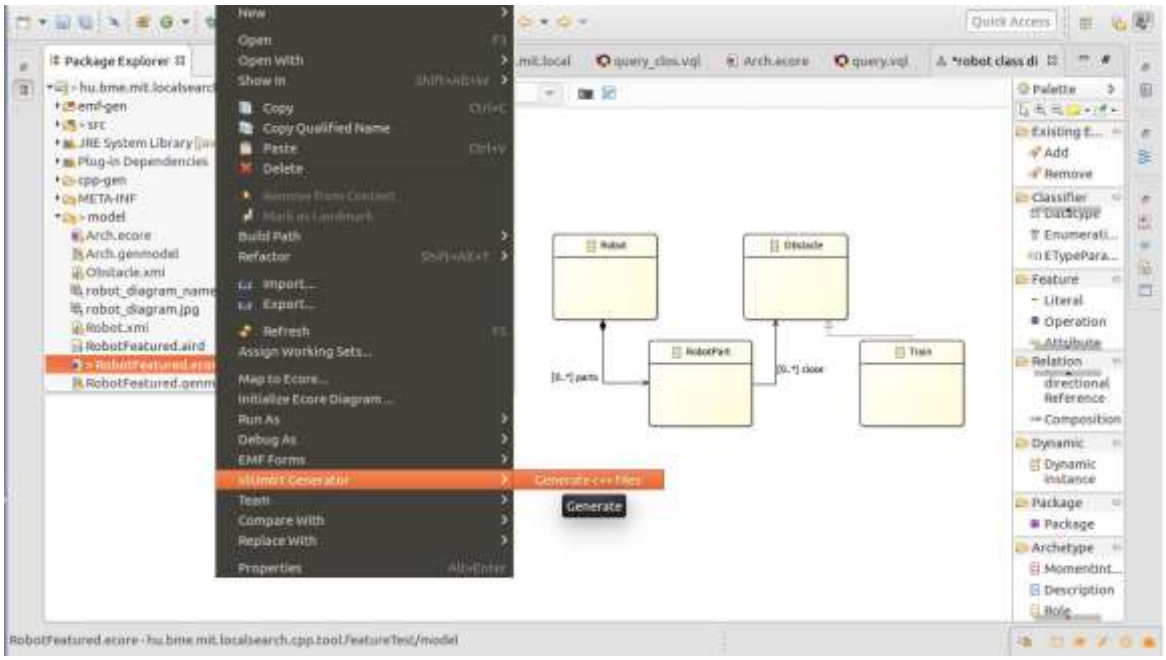

Figure 58. Context modelling and generation of the corresponding code

7.2 Monitor synthesis on the basis of scenario specification

In this case the developer specifies the requirement model in form of extended sequence diagram using the Papyrus tool.

Accordingly, the graphical scenario model can be constructed using the elements of the UML2 Sequence Diagram formalism (see the corresponding palette on the right of Figure 59) with the restrictions and extensions given in Section 4.

Having the scenario model, there are two ways of generating the monitor code.

- The scenario model is mapped to an automaton that is directly represented using the textual intermediate representation defined for the statechart models (see in Section 4). This representation can be opened and the monitor code generator can be invoked in the same way as given in Figure 57.
- The code generator is directly accessible in the sequence diagram editor using the context menu of Papyrus (UML Tools/Export to Cpp as presented in Figure 60).

The functions necessary for evaluating context/configuration dependency can be generated as described and presented in Figure 58.

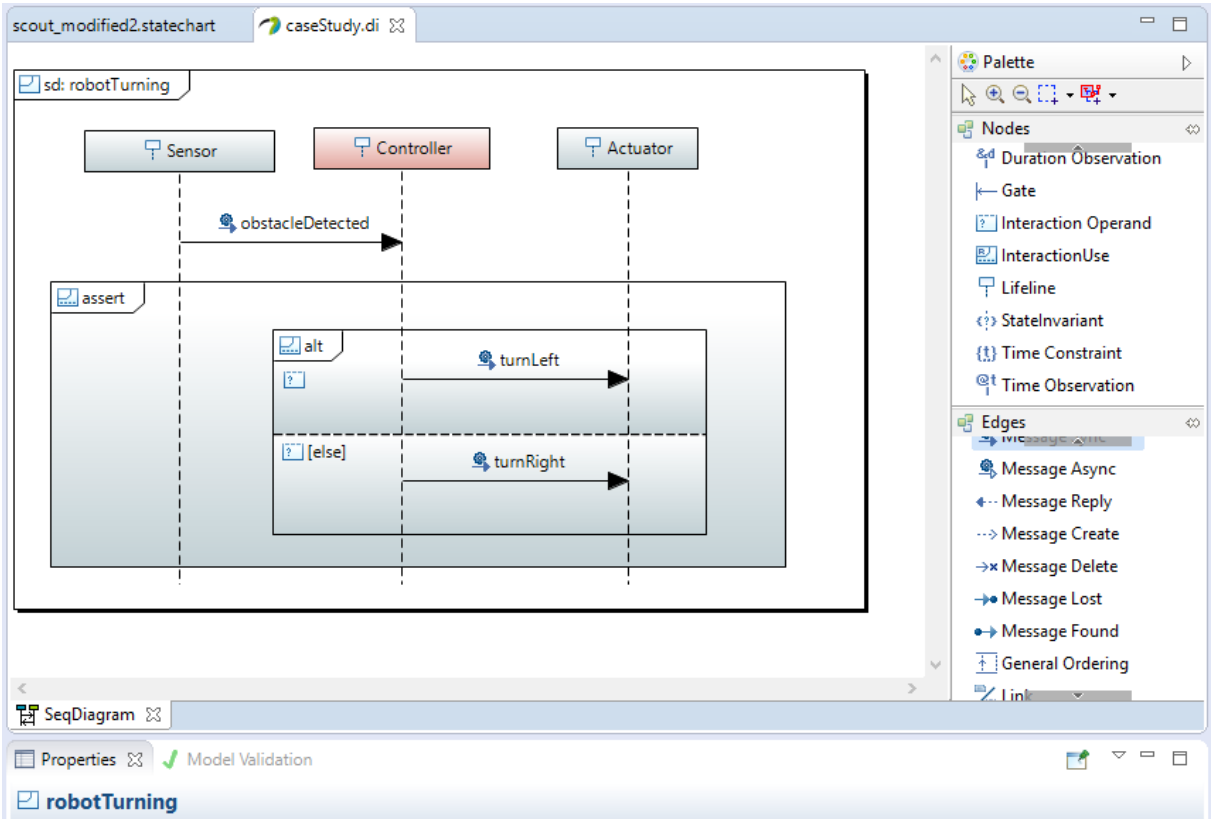

Figure 59. Construction of a scenario model

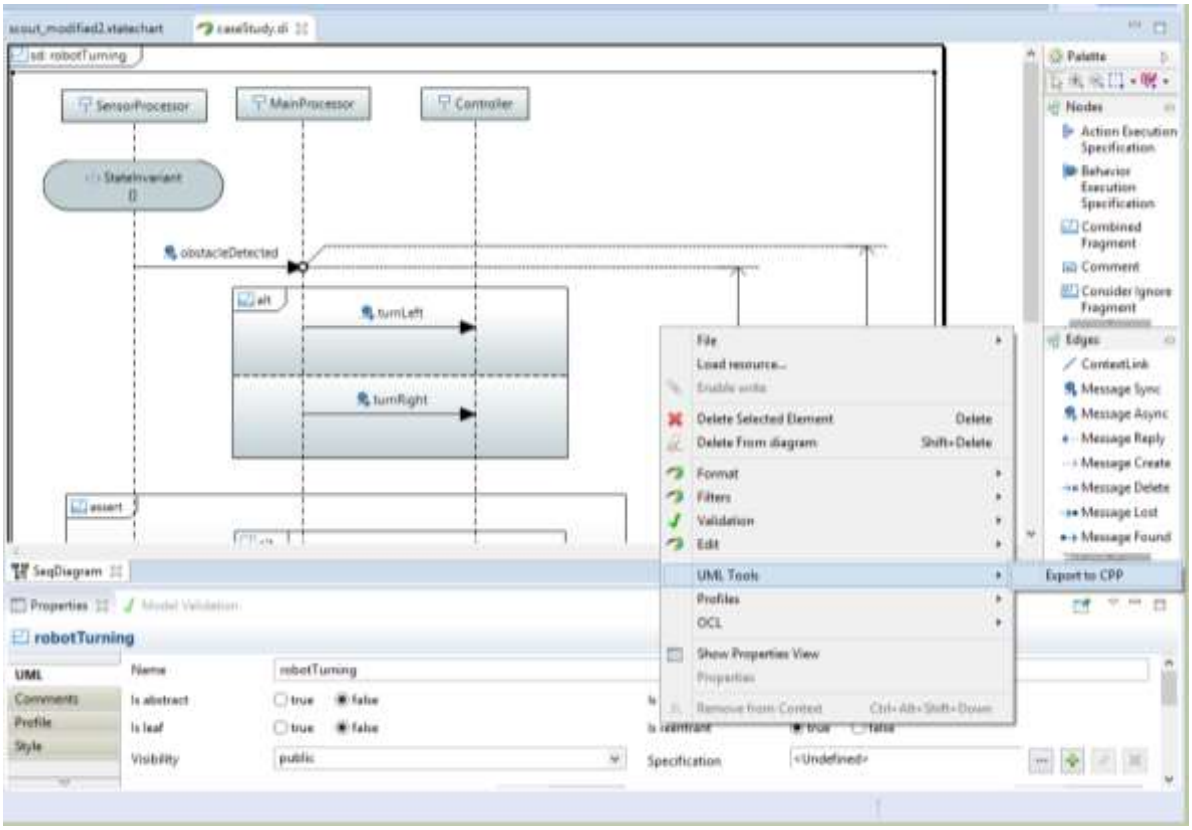

Figure 60. Generating monitor code from a scenario model

The resulting code (as C++ source file and the corresponding header files) can be accessed from the Project Explorer view of the tool (Figure 61). Note that the textual intermediate representation of the statechart is also generated and available.

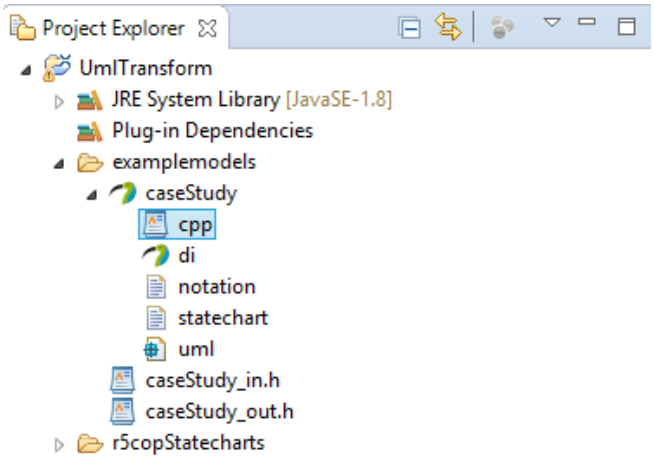

Figure 61. Result of the code generation as C++ files

7.3 Monitor synthesis on the basis of temporal specification

The temporal specification can be composed in graphical form using basic elements and predefined patterns as described in Section 5.1.

The graphical interface of the Pattern Composition Tool is depicted in Figure 62. Four main areas (identified by capital letters) are available as follows:

- The graphical editing of patterns is performed in area ‘A’. The elements displayed in this area are separated into three layers (CaTL layer, Context layer, and Store layer) that can be turned on and off during runtime.
- ‘B’ is the property area. Here the properties of the model elements can be set or modified (e.g., the name of an element, the corresponding reference, the parameters of expressions, etc.).
- Area ‘C’ presents the structure of the requirement (parts of the requirement and the corresponding representation files).
- ‘D’ is the Palette area. From this area elements and patterns can be copied to the editor area by drag-and-drop operations. The five groups of elements are the following:
    - Basic elements (atomic formulas as timing constraint, propositions, etc.);
    - Temporal logic operators (Next, Globally, etc.);
    - Boolean operators (And, Or, etc.);
    - Context elements (context fragments, nodes, connections);
    - Patterns (Absence, Existence etc. as presented in Section 5.1.1).

  The view of the pattern store with a few patterns is presented in Figure 63.

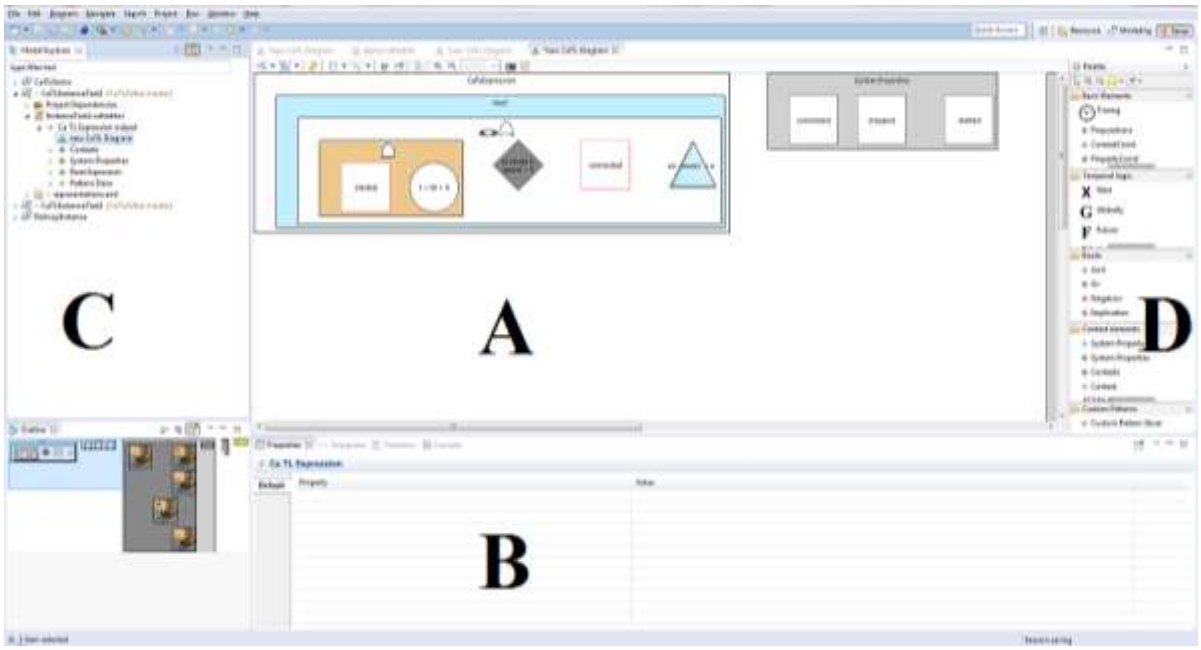

Figure 62. The graphical interface of the Pattern Composition Tool

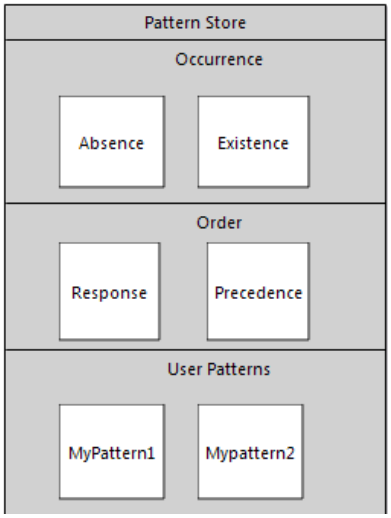

Figure 63. The pattern store with a few example patterns

The output of the Pattern Composition Tool is the LTL or CaTL expression that can be used for synthesis of monitors as given in Section 5.2. To do this, the LTL or CaTL expression is passed to a command-line tool that generates the C++ source files described in Section 6.4.

8 Conclusions

This deliverable described the tools designed for the synthesis of monitor components. Based on the languages used by the designers to specify the monitored properties, three tool-chains were designed:

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

We believe that these tools offer a flexible framework to specify properties according to the focus and level of completeness of the behaviour to be checked, and effectively support the generation of the monitor components.

This deliverable also presented the background algorithms used by the tools for processing the specified properties (requirement models) and for constructing low-level internal representations for source code synthesis.

The last sections contain the description of the monitor interfaces (in order to support the integration of the monitor components into ROS-based applications) and the demonstration of the usage of the tools.

The application and evaluation of monitoring will be covered by deliverable D34.50 (*Assessment of on-line verification and incremental testing*).

9 References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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