Summary: The UML (Unified Modeling Language) is a general-purpose visual modelling language that is designed to specify, visualise, construct and document the artefacts of a complex software system under design before committing it to code and thus address the entire application development lifecycle. The paper deals with the use of the UML in object-oriented modelling of the railway interlocking and signalling systems. Using the UML enables to design the complex control software on a higher qualitative level than other approaches did. Efficiency of this process can be increased by using an UML-compliant programming environment. For the sake of better understanding a simple example is presented.

1. Introduction

Railway interlocking systems are control systems with built-in safety-related functions. Safety is considered to be one of their basic properties and different techniques and approaches are used to analyse it (see e.g. [1], [2]). There are special requirements that must be met during particular phases of railway interlocking system lifetime. For the reason of high complexity modelling these systems may be rather problematic.

This paper generally concentrates on object-oriented approach to designing a model of the railway interlocking system. The UML is a single, common and widely usable modelling language, that evolved from Booch, OMT, OOSE and other OO methods. The UML allows software developers to specify, visualise and construct the artefacts of a system under design before committing it to code. Officially adopted by the Object Management Group (OMG) in November of 1997, the UML promotes the building of reusable components and helps to establish more economical and efficient development [3]. The UML defines several graphical diagrams as shown in Fig. 1. A number of other derivative views can be provided. The UML supports higher-level development concepts such as components, collaborations, etc. and is the basis of many SW tools. A set of criteria can be defined to select a suitable UML based tool (general features of tool environment, methodical analysis and design support, code generation, multi-user support, platform, reverse engineering, documentation, on-line help, technical and customer services, consulting and training services, pricing etc.). The authors present their practical experiences from using a standardised modelling language UML, supported by the object-oriented UML tool Rhapsody™ (a trade mark of I-Logix). Rhapsody
is a "visual" design tool for developing object-oriented software that allows doing the following:

- **Analyse** (define system requirements, identify necessary objects, and define their structure and behaviour using UML diagrams);
- **Design** (trace requirements to the design, taking into account architectural, mechanistic, and detailed design considerations);
- **Implement** (automatically generate code from the analysis model, then build and run it from within Rhapsody);
- **Test** (animate the application on the local host or a remote target to perform design-level debugging within animated views).

The paper deals with the first two activities. Analysis is the software development activity for studying and formulating a model of a problem domain and focuses on what is to be done; design focuses on how to do it.

Rhapsody implements Use Case Diagrams, Sequence Diagrams, Object Model Diagrams and Statechart Diagrams and goes one step further, translating their semantics into full, production-quality C++ code.

![Fig. 1 Diagrams contained in the UML](image)

**2. Requirement Specification for Object-Oriented Application - Control of Train Movement along the Railway Line**

*General requirements*

Let's assume the railway line equipped with three-element automatic block system that uses track circuits to monitor vacancy of track sections (TSs). The automatic block normally uses a basic block condition (BBC) that is additionally, under specific situation caused by train movement, temporarily changed to a full block condition (FBC). By default, block signals of the actual traffic direction are normally in clear position (green lights on), block signals of the opposite (false) traffic direction are off. Either traffic direction is always defined for optional number of trains. Block signals are merged with distant block signals. The last block signal simultaneously works as an entry distant signal.
**Operation requirements:**

When controlling train movements inside line sections, operation requirements for railway interlocking and signalling system result from relevant national standards and regulations. For the reason of limited extent of this paper only names of these operations are specified:

- Default (basic) state;
- Change of traffic direction;
- Return indication;
- Inter-relation of block signals of the same signalling point;
- Inter-relation of signal lights (green, yellow, red) inside a block signal.

![Fig. 2 Default state of block signals under traffic direction from A to B](image)

**3. Analysis of Object-Oriented Application - Control of the Signalling Point 2-3**

**3.1 Use Case Diagram**

Use Case diagram (see example in Fig. 3) can show typical interactions between the system under design and external objects, called actors, who may want to interact with it. Use Case diagrams employ specific symbols. Stick figures are used to specify external objects that interact with a use case of the system. A use case is one use of the system under design, drawn as an ellipse. System boundary (drawn as a rectangle) represents the environment outside the system under design; use cases are inside the system. Links define semantic relationships between classes, have at least two ends, each of which is connected to a class. Generalisation arrows between either use cases or actors show generalisations (the arrow head points to the parent use case or actor).
3.2 Sequence Diagram

Sequence diagrams can show possible scenarios in the execution of a use case. A single use case can be associated with many different scenarios. Each sequence shows how the participating objects communicate by passing messages to each other over time. Vertical time lines are instance lines that show the sequence of messages that an object processes and states that it enters over its lifetime. A column of short diagonal lines indicates the system border. In Fig. 4 there is shown an example of the sequence diagram for the train passing the block signal S3.

![Sequence Diagram](image)

Fig. 4 Example of the sequence diagram for train passing the block signal S3

3.3 Object Model Diagram

Object Model diagram shows the static structure of a system. It shows the types of objects in the system, the attributes and operations that belong to those objects and static

![Object Model Diagram](image)

Fig. 5  Object Model Diagram
relationships that can exist between classes (types) and constraints that may apply. In Fig. 5 there is shown a such a diagram as processed in the Rhapsody tool (for the reason of limited space only simplified form of view is used - the class name and no attributes, no operations). Little pictures of a camera located in the top right-hand corner of objects illustrates existence of statechart diagrams designed for these objects. Inheritance refers to the derivation of one class/object from another. The child object has all the parent's attributes and may have its own special attributes. The arrow head points to the parent class.

3.4 Statechart diagram

Statecharts define the behaviour of objects, including the various states that an object can enter into over its lifetime and the messages or events that cause it to transition from one state to another. State diagrams illustrate the responses and actions taken by the object as a result of the stimuli, which are generally in the form of messages.

State is a condition in which an object finds itself. Transition labels use the following format:

\textit{trigger [guard] / action}

Each statechart must have a default transition, i.e. a transition to the default state of the object. A default transition can have neither a trigger or nor a guard. In Fig. 6 there is shown a statechart diagram for the object \textit{Signalling Point SP2_3}. This diagram consists of two (in our case, two is not the final number of) orthogonal states, i.e. independent states that an object can be in at the same time (also known as “AND” states). The dotted line separates the compartments of orthogonal states. The diagram also contains condition connectors (a circled C letter) that split a single segment of a Statechart into several branches with guards enclosed in square brackets labeling the branches. Whichever guard is true determines which state the object will branch to. Branches can contain triggers and branching segments can be nested. Condition connectors are a way of visually representing if-then-else conditions.

Rhapsody as any other UML based tools has a checking mechanism implemented that allows to select more than 30 different checks to be performed on the model or on a part of the model before Rhapsody generates code. Checks detect potential problems, e.g. name conflicts, that cause code generation errors if not corrected. Correcting problems encountered by these checks increases the likelihood that the code will generate successfully. The checks can be classified according to error severity to two different levels - errors and warnings. Apart from the checking mechanism, Rhapsody animates the actual running application, rather than just simulating it. Animated views such as Animated Browser, Animated Sequence diagrams, and Animated Statecharts, allow us to observe the application as it is running and perform design-level debugging. The animation control panel allows us to step through the program, set and clear breakpoints, and inject events to observe the system's reactions to hypothetical situations in quasi-real time. Finally, we can generate an output trace. Running animation involves the following steps: creating a component, creating a configuration for our component, generating component code, building the component application and running the component application.

This work has been supported by the Grant Agency of the Slovak Republic VEGA, grant No. 1/8182/01 “Theoretical foundation for calculation of acceptable risk in control of transport process, especially the railway one”.
4. References


   http://www.omg.org