MODELING OF RAILWAY LOGICS FOR REVERSE ENGINEERING, VERIFICATION AND REFACTORING

F. FLAMMINI[1,2], A. LAZZARO[1] & N. MAZZOCCA[2]
[1]ANSALDO STS, RAMS Unit, Via Nuova delle Brecce 260, Naples 80147, Italy.
[2]University of Naples "Federico II", Dipartimento di Informatica e Sistemistica, Via Claudio 21, Naples 80125, Italy.

ABSTRACT

Model-based approaches are widespread both in functional and non-functional verification activities of critical computer-based systems. Reverse engineering can also be used to support checks for correctness of system implementation against its requirements. In this paper, we show how a model-based technique, using the Unified Modeling Language (UML), suits the reverse engineering of complex control logics. UML is usually exploited to drive the development of software systems, using an object-oriented and bottom-up approach; however, it can be also used to model legacy non-object-oriented logic processes featuring a clear distinction between data structures and related operations. Our case-study consists in the most important component of the European Railway Traffic Management System/European Train Control System: the Radio Block Center (RBC). The model we obtained from the logic code of the RBC significantly facilitated both structural and behavioral analyses, giving a valuable contribution to the static verification and refactoring of the software under test.
Keywords: control software, modeling, railways, refactoring, reverse engineering, verification.

1 INTRODUCTION AND BACKGROUND

A large class of fault tolerant systems is constituted by mission and safety critical apparels employed in industrial control applications, like defense, aerospace and transportation. In fact, most control systems nowadays feature complex computer-based architectures. Critical control systems require a number of thorough verification and validation activities, which are regulated by international standards (see for instance [1, 2]). With reference to software verification, such activities can be roughly divided into static and dynamic ones: static analyses do not require code execution, while dynamic ones require execution on the target hardware or in proper simulation environments. Theoretically, static analyses should precede dynamic ones in the system life cycle, with the aim of finding gross grain errors in coding style and structure. Once these errors have been detected, the testing phase can begin with the aim of detecting mostly functional non-conformities [3]. However, in practice code inspection proves useful to find errors which are very difficult to find by testing alone, and it usually accompanies testing till the very end of the project, being also useful for diagnostic purposes and to predict behavioral impact of code modifications. More specifically, in Fig. 1, we provide the well-known V-Life Cycle model used for software development (and prescribed by most international safety standards, see e.g. [2]): the step circled by a light-grey dashed line represents where the activity described in this paper correspond to the verification stage (right part of the diagram).

The control software of many critical control systems is written, totally or partially, using application-specific or legacy languages. The choice of using such languages is justified by several factors, e.g. the dependability of the (validated or highly proven in use) development environment, the ease of using the 'natural language like' syntaxes, the already available debugging tools, etc. Featuring proprietary syntax and usually missing the support for object-oriented programming structures, the drawbacks of such languages are that no existing

ISSN: 2041-9031 (paper format), ISSN: 2041-904X (online), http://journals.witpress.com
DOI: 10.2495/SAFE-V1-N1-77-94

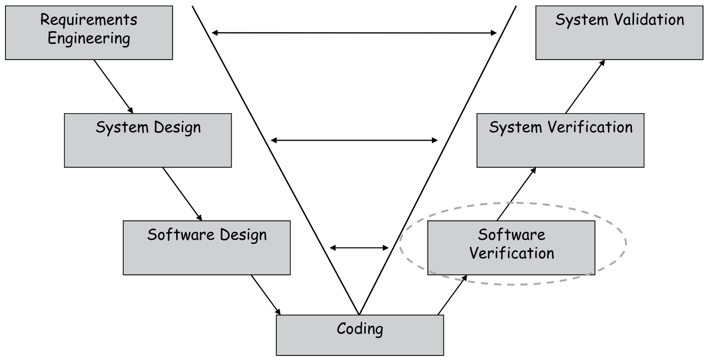


Figure 1: V-Life Cycle model for critical software development and validation.

methodology or tool can be directly applied to them in order to accomplish the objectives of test engineers in the static verification phase. Among such objectives we list the following:

- To obtain extensive documentation describing in detail how the control logic works, which is necessary for any kind of testing and maintenance activities.
- To trace the architecture and behavior of the control logic into the higher level software specification, allowing for an easy verification of compliance.
- To optimize code reliability and performance, possibly by means of refactoring techniques, that is behavior preserving transformations [4].

Such results can be achieved by means of a proper model-based reverse engineering approach. Reverse engineering is the process of analyzing a system to identify its components and their interrelationships and to create a representation of the system in another form or at a higher level of abstraction [5]. Reverse engineering support with a proper bottom-up modeling of software implementation allows for:

- An easier static verification of high level functional requirements, as the model is more compact, expressive and easily readable also by people non-skilled in programming (e.g. system analysts or application domain experts).
- The availability of a flexible representation which can drive the refactoring work, involving code manipulation aimed at software improvement.

In this paper, we describe the application of a model-based reverse engineering approach for code verification and improvement purposes. Such an approach is aimed at performing a static analysis – both at the structural and behavioral levels – of the software used for a specific class of railway control systems. The modeling language which best suits the bottom-up modeling of logic code, regardless of its syntax, is the Unified Modeling Language (UML). One could argue that a fundamental constraint for the advantageous use of UML in such context is the use of object-orientation in logic code. However, although most legacy or application specific control languages do not feature a specifically object-oriented syntax, they can be object-based, or used as such, with well known advantages (e.g. data encapsulation). In such approaches, logic processes are usually associated to a set of data and possible operations of a well distinguished entity (either physical, e.g. a sensor or an actuator, or logical) of the control system. In cases where such rules are generally not respected, the approach described in this paper could still be followed, but a preliminary object-oriented reengineering and design review would be necessary, whose description is not in the scope of this work.

The use of UML for verification purposes is not as formal as could be code-based model checking [6], but it easily allows for governing the complexity of real-world systems; furthermore, it constitutes the basis of a series of possible analyses and refinements, as explained in this paper.

The approach presented in this paper was applied to a real world case-study: the Radio Block Centre (RBC) of ERTMS/ETCS (European Railway Traffic Managements System/ European Train Control System) [7]. Presently, the certification process of two RBCs developed by our company included the verification approach described in this paper; these RBC are already operational, controlling one of the newly developed Italian high-speed railway lines, connecting Turin to Novara. (The line, activated for the Winter Olympics of Torino 2006, has been operational since February 2006. It is part of the Turin–Milan line, which is still under development.) To give an idea of its complexity, the control logic of the RBC is constituted by more than 250,000 lines of code. RBC software is mainly written using a legacy language, highly 'proven in use' in past projects (above all in interlocking applications; see [6]). Due to the similarity of this process-based (As it will be clear in the following of this paper, a process of the logic language refers to attributes and operations of a control entity and must not be confused with a CPU scheduling unit (though the origin of its name is actually related to the cyclic Real-Time scheduling of the operations).) with object-based languages, UML-based modeling was quite straightforward and the graphical representation provided, as expected, very useful documentation and analysis facilities.

The remainder of this paper is organized as follows: Section 2 locates the approach in the context of related works; Section 3 provides a general description of the approach, regardless of any specific control system, dividing it into three main steps; Section 4 presents ERTMS/ ETCS and then shows by an example the application of the approach to the control logic of the Radio Block Center; Section 5 summarizes results and future developments.

2 RELATED WORKS

The work presented in this paper is based on the UML. UML is widely accepted as the de facto standard in software engineering; its syntax and semantic have been standardized by the Object Management Group (OMG) [8]. UML is used by a number of commercial automated modeling and development frameworks (see for instance [9]).

Several applications of UML not strictly related to software design can be found in the research literature. In [10] the authors present a methodology in which the validation of system dependability properties since early design stages is based on translation of UML views into formally analyzable models. In the field of UML based performance verification it is significant to cite the work described in [11]. The article referenced in [12] describes an approach for automatically generating and executing system tests from UML behavioral diagrams.

A significant amount of literature is available on UML-based refactoring (e.g. [13]) and on reverse-engineering. A survey of industrial best-practices of reverse engineering is provided in [14]. Many theoretical works are available on reverse engineering, often dealing with specific issues (it may suffice to cite the IEEE Working Conference on Reverse Engineering). There also exist tools realizing the reverse engineering step starting from C++ or Java code (see e.g. [15, 16]). However, such tools cannot operate or be customized to work on proprietary control languages (unfortunately a common scenario).

Several research works are also available on UML-based reverse engineering, see e.g. [17–19]: in the latter, which has some common points with the work presented in this paper, reverse engineered Sequence Diagrams are compared with design diagrams for the verification of conformance of the implementation to the design. With respect to the approach presented in this paper, which also includes structural verification and refactoring based on more UML views, in [19] the verification is limited to behavioral aspects represented by means of interaction diagrams (not considering the obvious difference in the criticality of the application domain).

Considering static analysis (Both in literature and common practice the terms 'static analysis', 'code review/inspection' and 'code walk-through' usually indicate different activities; however, in this paper we will generically refer to 'static verification' as the set of activities which are performed without code execution, in contrast to dynamic verification (i.e. testing), requiring simulation.), code inspection [20] and related verification activities, many works exist in the literature (see for example the approach developed at Verimag for the static analysis [21]): while they can automate many checks, when it comes to performing non-trivial inspections their common limitation consists in the fact that the verifier is not aided by any more or less formal model which would facilitate the work by providing proper views on the code (in practice, only function call-graphs [22] and code navigators are used).

The automatic verification of properties on a system model is known as model-checking. Several approaches to model-checking real-time control systems have been proposed and successfully applied; some directly on code (see [6, 23, 24]), others on UML models (see [10, 25, 26]). However, it is generally accepted that formal verification is rarely adequate to govern the complexity of real world systems, and their contribution can be considered as complementary to the ones provided by traditional approaches (i.e. static-analysis, code inspection, structural and functional testing, etc.), as witnessed by several past experiences (see e.g. [6] for railway and [27] for aerospace applications).

Despite the relevant amount of literature on these subjects, to the best of our knowledge there is no work dealing with the integration of model-based reverse engineering approaches in the life-cycle of critical systems for both verification and refactoring purposes, which is the main topic of this paper. An early application of UML to the analysis of railway control logics has been described in [28], in which, however, the authors did not yet abstract the methodological aspects, which are necessary to generally and systematically integrate the approach in the life-cycle of industrial control systems.

3 THE GENERAL MODELING AND VERIFICATION APPROACH

Traditional industrial approaches of static verification are mainly aimed at ensuring that the code respects the rules of quality standards, that is to say:

- it does not feature prohibited statements,
- defensive programming controls are regularly implemented,
- function length is limited,
- comments are regularly and correctly employed,
- etc.

Besides such verifications, checklists can be used in order to verify other properties which are possible to check without program execution (e.g. variables are correctly initialized, the number of iterations and stop conditions of cycles is correct, etc.) [20]. While some controls can be automated, many others are left to the sensibility of testers who perform code inspection (or 'walk-throughs'), and therefore are quite error prone. This is generally considered not a main issue, as most of residual errors are usually more easily found by means of dynamic analyses (i.e. software testing). In other words, it is generally accepted that static verifications are meant to find as many errors as possible before code execution, but they cannot assure extensive error detection or compliance to high level requirements. Nevertheless, in practice the impact of static means of analysis becomes more significant as code complexity grows, because of the limitation of the test-set. Furthermore, the test-case limited execution time often reduces the probability to detect latent errors due to wrong assignment of variables: it can happen that a test-case is considered as passed because monitored variables are assigned correct values, but if execution continued in a certain way, the system would hang (hopefully) or even produce incorrect output. Finally, as functional test effectiveness is usually measured by means of decision coverage and rarely by means of data-flow coverage (whose exhaustiveness is nearly always impossible to achieve in practice), test engineers tend to say that some 'data-related' errors can only be detected by code inspection, hence the importance of such a verification phase.

However, there are two significant limitations in traditional static analysis approaches. The first one is that they are not focused on behavioral analysis, in the assumption that system behavior is hardly checked statically; however, a behavioral check at the static analysis stage would be highly advantageous, as it could:

- highlight unconformities hard to be found by testing and/or at an early stage of system development;
- evaluate the impact of code modifications on system predicted behavior.

The second limitation is that traditional static analysis hardly helps verifiers to suggest system level design reviews, reengineering or code restructuring aimed at improving software implementation, with effects on reliability and performance. Some efforts toward such objective can be made with general purpose programming languages, like C, for which specific tools are available providing function call graphs and other high level informal views on the code (see e.g. [22, 29]). However, the effectiveness of such views in helping verifiers to find errors and improvement possibilities is quite limited.

An attractive and feasible way to govern code complexity is represented by model-based approaches, allowing to facilitate traditional analyses as well as enable new ones. As mentioned in Section 2, while model-driven software development literature is quite rich, model-based verification literature is often focused on dynamic analyses (i.e. software testing) and/or limited by the use of single representations (e.g. state diagrams).

The main novelty of the model-based methodology proposed in this paper is the extensive application of UML to the static analysis of a piece of critical software. On the base of our successful experience, we believe that the systematic integration of such an approach in system life-cycle is very advantageous both in terms of effectiveness (more errors are revealed before the dynamic verification stage) and efficiency (engineers are facilitated in their static analysis work by exploiting high-level graphical representations). Furthermore, the approach also facilitates the improvement of software design by enabling UML-based refactoring techniques. In fact, the so-called ‘design review’ is an important aspect of software verification, given its potential impact on maintainability, reliability and performance.

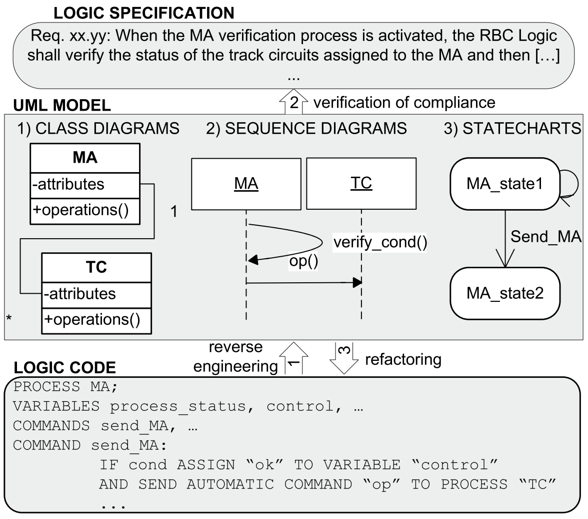
UML is undoubtedly the most comprehensive language for software modeling. In fact, an extensive UML model:

- abstracts code implementation in order to facilitate comprehension of both code structure and behavior;
- can be more easily checked by hand or even by using specific model checkers, whenever available;
- provide means to support code review and improvement (i.e. refactoring).

In order to employ such a UML-based approach in the static analysis stage of legacy systems, the software must be reverse engineered. Such operation is syntax specific and automatable. Once the model is available, it can be checked for verification against functional requirements and refactoring, as explained in the following sections.

An important general aspect of the approach is that it also embodies in a semi-formal and structured way the Agile Modeling idea of ‘modeling to understand’ [30], which stresses the importance to have as many views on the code as possible. Therefore, besides enabling systematic verification approaches (e.g. based on traceability or model-checking), modeling itself can reveal errors as it provides useful insights about code structure and behavior.

In summary, our approach can be divided into three main steps, to be executed sequentially (the former providing the output for latter): (1) reverse engineering, consisting in building the UML model from the logic code; (2) verification of compliance with logic specification, by means of a traceability study; and (3) refactoring, that is code restructuring based on the available and already verified UML model. Such approach is represented in Fig. 2 and described in the following sections.



Req. xx.yy: When the MA verification process is activated, the RBC Logic shall verify the status of the track circuits assigned to the MA and then […]
...

```
PROCESS MA;
VARIABLES process_status, control, …
COMMANDS send_MA, …
COMMAND send_MA:
         IF cond ASSIGN “ok” TO VARIABLE “control”
         AND SEND AUTOMATIC COMMAND “op” TO PROCESS “TC”
         ...
```

Figure 2: Three steps scheme of the modeling and verification approach: (1) reverse engineering; (2) verification of compliance; and (3) refactoring.

3.1 Step 1: Reverse engineering

The first step consists in building a UML model of the logic code, using proper diagrams, with the main aim to compare such representation (obtained 'bottom-up') with the high-level logic requirements in order to verify the compliance of system implementation with its specification.

While building the diagrams, it is also possible to verify 'on-the-fly' the basic rules of a good object-oriented programming paradigm (e.g. data encapsulation into objects) and plan a first level restructuring of the code, which however will be implemented only after logic behavior is verified (see Section 3.3 on code refactoring). Furthermore, by using a modeling environment which is not only a diagram drawing tool, but also a syntax verifier, it is possible to automatically check the correctness of the model in terms of some classes of errors, e.g. call of an undefined operation. This may appear as a trivial control, but in practice such errors are often destined to remain latent until the related code is exercised; however, for critical systems some code blocks are hardly exercised, with the risk of causing system failures (hopefully, safe shutdowns) in the rare cases in which they are needed, e.g. to manage specific exceptions.

The most important structural view which can be built from the code consists in class diagrams, statically showing the relationships between logic processes. Class diagrams provide a static view which is able to give test engineers at a glance and integrated representations of software architecture [31].

Among behavioral views, sequence diagrams best suit to represent the dynamic aspects of logic processes, by highlighting process interactions in terms of execution of operations and data structure modifications. Sequence diagrams allow an easy comparison of process behavior with the one requested by the high-level specification, which are written in natural language in the form of input–output relations (see Section 3.2).

State diagrams (or 'state charts') also constitute an important behavioral view which can be employed to check process state transitions. Other types of UML diagrams (e.g. Use-case and Activity) can be used as additional views, e.g. to simplify the understanding of complex operations. For example, Use-case Diagrams (which are missing in Fig. 2) are often used to model the triggering of the interactions (i.e. scenarios) between external actors and internal processes, described in details by means of Sequence Diagrams (see Sections 4.2.1 and 4.2.3).

3.2 Step 2: Verification of compliance

Several types of verifications are possible on the UML diagrams. As first, it is important to verify that functionalities specified in logic requirements are present in the code and no unnecessary functionalities are present. This sort of coverage and traceability analysis for functions is facilitated by verifications on the UML models. A second straightforward verification is that sequence diagrams should only contain the processes involved, as specified by high-level requirements. Beside specification-based verifications, state diagrams also offer verification possibilities only based on software implementation: the reachability of all process states must be guaranteed, the occurrence of 'sink states' (i.e. states with no outgoing transitions) should be prevented, etc. Such kind of analyses can be performed informally, exploiting the know-how and skill of system experts, but can also be (at least partially) automated. State diagrams can also be model-checked, which can be advantageous for specific pieces of software, e.g. the ones managing communication protocols.

A formal traceability analysis can be obtained by a hierarchical superposition of sequence diagrams, with the aim of matching the ones obtained top-down from the high-level logic specification with those obtained bottom-up by logic code modeling. To perform this, partial sequence diagrams have to be linked to build up complete scenarios. In other words, after process operations are modeled singularly in sequence diagrams, they have to be linked together in order to form a complete scenario, from the first triggering of an operation to the reaching of a stable state, in order to be traceable on the high-level system specification.

3.3 Step 3: Refactoring

Analysis, refinement and optimization by code restructuring constitute the last step of the process. A generally accepted definition of refactoring can be found in [4]: 'the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure'.

In our approach, refactoring is performed on the UML diagrams and then implemented top-down in the logic code. The availability of a model of the software under analysis, featuring complementary views, allows for an easy detection of the so-called 'smells' in the code [13]. Smells are simply defined as code structures that suggest the possibility of refactoring, like degenerate classes, e.g. too large or too small, featuring only data, pleonastic (just forwarding method calls), etc. In particular, for critical systems, defensive programming controls have to be added to check and react on inputs that are illegal or incompatible with process status, in order to tolerate casual (e.g. soft-errors in memories) as well as possibly systematic faults. Moreover, refactoring can involve the grouping of condition checks, the re-ordering of checks (weighted by occurrence probability and/or by the number of necessary steps to perform the check of the condition) and other specific performance optimizations. The use of sequence diagrams allows test engineers to precisely weigh the condition checks in terms of needed interactions between logic processes, and thus in terms of elaboration cycles.

As for the respect of object-orientation, whenever necessary, operations must be added in class diagrams in order to let processes modify variables of other processes only by using specific methods and not by directly accessing external attributes. Such a modification also allows moving defensive programming controls from the calling methods to the newly added ones, possibly gaining in reliability, readability and code length.

The behavioral impact of such modifications can be easily checked by means of UML sequence and state diagrams. In particular, state diagrams also allow for behavioral refinements, e.g. aimed at moving/grouping states in composite ones. As compliance with high-level requirements has already been verified in the previous step, it is necessary that such transformations preserve the logic behavior of the system.

4 AN INDUSTRIAL APPLICATION: THE RADIO BLOCK CENTER

In our experience with the RBC, almost all the activities described in Section 3 were performed by hand; in fact, while the building and verification of the model is theoretically automatable, when the verification process started we were not completely confident about the worth of automating it (see Section 5 for future developments). As aforementioned, the structure of RBC logic processes well suits to be modeled using an object-based language: logic processes can be easily thought of as classes, with their internal 'variables' constituting the attributes, and their 'commands' being the operations. In particular, the 'process status' variable has been represented in a specific state diagram, as it triggers most of the process behavior and thus is very critical. The modeling tool we used to build UML diagrams was the Rational Rose Modeler Edition [9]. While building the model we exploited all the syntactic controls and automations of the Rational Rose Modeler application, which constituted a first congruity check on the model under construction. Some examples: as the model grows up, the already defined links are automatically added by the application; in sequence diagrams, it is not possible to call an operation which has not been already defined in class diagrams; the accessibility of attributes and operations (public, protected, private) is automatically checked; etc. Next section will provide a general introduction to ERTMS/ETCS.

4.1 ERTMS/ETCS level 2 system implementation

ERTMS/ETCS is the specification of a standard aiming at improving safety, performance and interoperability of European railways [7]. In Italy, the so-called Level 2 specification of ERTMS/ETCS is implemented in high-speed railway lines. ERTMS/ETCS specifies three main subsystems: trackside, on-board and lineside. The trackside subsystem is the ground part of the overall system, and manages railway signaling to ensure safe train separation and routing. The most important trackside subsystem is the Radio Block Center, which has the aim of collecting track and train information in order to provide trains with necessary data: Movement Authorities (MA), that is the distance trains are authorized to move on; Static Speed Profiles (SSP), that is maximum speed limits allowed by the track; possible Emergency Stops (ES). The on-board control system performs train protection by controlling train speed against the elaborated dynamic speed profiles (or braking curves). The lineside is constituted by the so-called balises, which are devices positioned between the track lines, which have the aim to transmit geographical positioning data to the trains passing over them; such data is used by the on-board system in order to build and send position reports to the trackside.

The Radio Block Center is responsible of providing train headways by using data received in train position reports and delivering the correct MA, SSP and ES messages to the trains. In order to detect the necessary route and Track Circuit (TC) occupation status, RBC is connected to the national Interlocking (IXL) system, which is not standardized in ERTMS. Moreover, the RBC is usually able to manage a limited number of track sections, thus it must be connected to other adjacent RBCs in order to allow for the so-called train Hand-Over (HO). The Hand-Over is the complex procedure that manages the passage of a train from the area supervised by a RBC to the area of adjacent RBCs. In ERTMS level 2, the on-board and trackside communicate by the GSM-R radio network using the Euroradio protocol (see Fig. 3).

The software architecture of the RBC is depicted in Fig. 3. It consists of some ‘application’ processes (written in a safe subset of the C programming language), which manage the interaction of the RBC with the external entities (i.e. other subsystems and the RBC operator), and some ‘logic’ processes, which implement the control logic of the RBC; the latter are written in an application specific logic language, as already mentioned above.

The Logic Manager has to interpret logic language, translating it into executable code, and to schedule the logic processes. The shaded rectangles in Fig. 4 represent the logic processes, which are the target of our work. For instance, the ‘MA’ process, shown in Fig. 3, verifies the integrity of the Movement Authority assigned to the train against all the significant track and route conditions. Such conditions are received from the interlocking system by means of the ‘IXL interaction’ application process and are managed by the ‘TC’ logic process, which stores Track Circuit physical conditions in its internal variables. If the MA integrity verification fails (for instance a track circuit is not clear or involved in an emergency condition), the

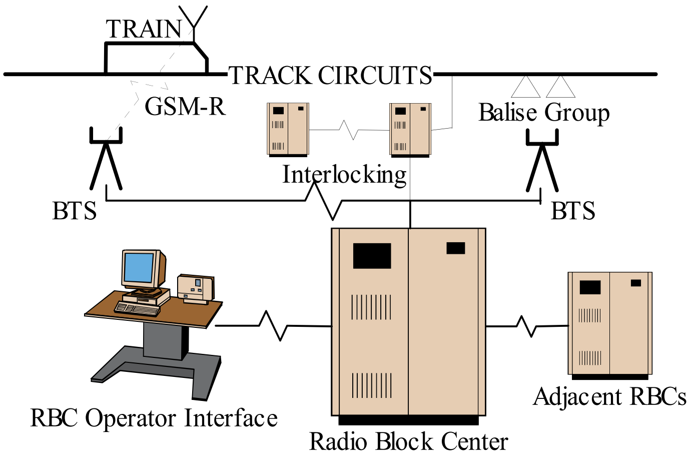


Figure 3: ERTMS/ETCS level 2 reference architecture for the trackside subsystem.

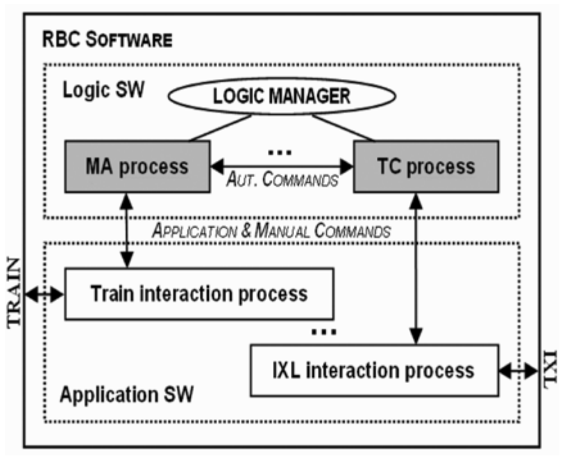


Figure 4: The software architecture of the Radio Block Center.

MA process has to command the sending of an emergency message to the 'Train interaction' application process, which will manage the sending of the proper radio message to the train. In this simple example, the MA and TC processes interact with the application processes to manage data from and to the external subsystems; moreover, they interact with each other, as the MA process asks the TC process for the status of track circuits, in order to verify the integrity of the movement authorities.

There are many other logic processes which behave in a similar way. They all can feature:

- a set of data variables and a special 'process status' variable;
- a set of 'operations' which can be activated:

  by application processes or the RBC operator by means of 'manual commands';

by other logic processes, issuing 'automatic commands'; directly, when the process status variable is assigned a certain value.

Typical process operations are:

- verify conditions on internal variables or on the ones of different processes;
- assign values to variables which can be either internal to the process or belonging to other processes (the Logic Manager does not restrict the visibility and modifiability of variables);
- issue 'automatic commands' to other processes;
- issue 'application commands' to application processes (e.g. 'send MA').

In order to exhaustively explain the reverse engineering procedure, it would be necessary to introduce all the syntactic constructs of the logic language (which have been briefly described in [6]) and the related modeling rules. However, an exhaustive explanation of modeling rules is not in the scope of this work, for two main reasons: as first, the rules, being language specific, would not be of any interest out of the industrial context of railway control systems produced by Ansaldo Segnalamento Ferroviario (not considering confidentiality issues); second, once defined the rules, the reverse engineering phase is mechanical and hence of no theoretical relevance. Of course, the translation procedure must be complete and sound, covering all the syntactic constructs: starting from the low level logic language elements, it is possible to unambiguously identify, e.g. the activation of a process operation, naming conventions, comments, state variables, etc. This is facilitated by the absence in the language of dynamic binding, pointers or references; in fact, the language was specifically developed to be understandable by non-programmers. The result is that the reverse engineering procedure can easily be automated, in analogy with Computer Aided Software Engineering (CASE) tools featuring 'round-trip' engineering facilities suitable for general purpose languages (see e.g. [15]). Differently from such tools, the representation provided by the proprietary tool we are presently developing is application specific, that is to say it provides a representation fitting the specific needs of test engineers (a general and difficult to check representation would not prove useful for verification purposes).

Next section will provide an example application of the model-based reverse engineering approach to the RBC case-study.

4.2 Reverse engineering, static verification and refactoring of the RBC

In this section, the modeling technique from RBC logic language to UML diagrams is described by referring to the simple 'Change of Traction Power' (CTP) logic process. To give an idea of its relative complexity, such process is nearly 20 times smaller in terms of lines of code with respect to the MA process (the latter being very difficult to describe in detail without an in depth knowledge of the RBC software architecture).

The CTP process has to manage manual commands of activation/deactivation of a change of traction power line section coming from the RBC operator. The activation/deactivation commands must be accepted only if the track circuit in which there is a change of power is free and not included in a MA assigned to a train. In the following we present the UML based modeling and analysis for such example process.

4.2.1 Use-case diagrams

Use-case diagrams represent how a logic process can be used by application processes or by the operator using manual commands; therefore, they represent the externally triggered high-level procedures (see Fig. 5). Usually, at least one use-case diagram is specified for each system level scenario (e.g. Start of Mission, Hand-Over, etc.).

Use-case diagrams are significant in representing all the possible functional scenarios involving the logic process, hence giving the possibility to verifiers to check the validity of the actions performed on the process at a very high abstraction level; however, they reveal not so useful for refactoring purposes.

4.2.2 Class diagrams

Class diagrams are built starting from a static software architecture view (data structure with internal/public attributes, operations), also showing the relationships and interactions between processes (i.e. read/write of attributes and method calls). Class diagrams represent the backbone of the entire model, as the first modeling step consists in the definition of a class for each of the logic processes. The relationships between classes are determined by the 'access' or 'assign' statements (as aforementioned, all process variables are considered as public by the Logic Manager) and by issuing commands triggering process operations. Given the high number of associations (some processes can share up to 20 distinct links) a comprehensive class diagram would be very complex and hardly readable. It is preferable to build partial class diagrams, each one focusing on a single process and showing all the incoming and outgoing links to/from that process. As shown in Fig. 6, the CTP process features two local variables, a single refereed process (TC) and two operations. In case a process only accesses (or 'reads') data of another process (usually to check some conditions on the state of the other process), the link is modeled by a dependency relation (dashed line). In case, instead, the process also activates operations of the other process (by automatic commands), the link is modeled as a real association (full line). If the modeler does not need to build an

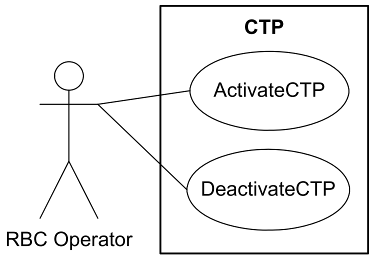


Figure 5: Use-case diagram for the CTP logic process.

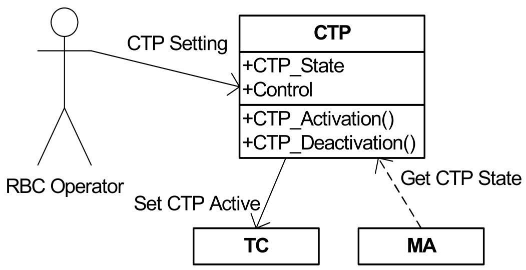


Figure 6: Class diagram for the CTP logic process.

executable model (as in our case), he/she does not need to be fully compliant to the UML standard, and can therefore adapt the notation to his/her needs (It is important to remark that the future re-use of the models for re-design or automated source code generation will be hindered by this approach; therefore, we are now moving toward an adherence to the official OMG standard; see Section 5.). For instance, we also associated a list of accessed variables and operations to relations, as a further documentation.

From the class diagram, we can see that the CTP process is used by the MA process: this is correct because the MA process has to manage the presence of a CTP section as additional information to be added to the outgoing MA message in order to inform the on-board system of the CTP procedure. From the CTP class diagram it is evident that at least a refactoring action is necessary: class attributes must be kept as private, and a Get_CTP_State() operation (which is missing in the diagram) must be added in order to access the CTP state.

4.2.3 Sequence diagrams

Sequence diagrams are specified for each operation, showing its implementation and the detailed interactions between logic processes involved in that operation. A full arrow has been used for check and assignment statements, specifying the nature of the operation as a comment. A normal arrow is used, instead, for the activation of an operation. For better clarity, complex diagrams have been detailed using linked notes. In Fig. 7, we report an example sequence diagram for the CTP process, showing the activation operation (the deactivation is very similar): when an activation command is received, the state variable of the associated TC (belonging to a properly declared list of linked processes) must be checked in order to verify whether it is assigned the 'free' and 'not requested' (for a Movement Authority) conditions; if such condition is fulfilled, the automatic command Set_CTP_Active() can be issued to the TC process. The behavior obtained by checking such diagram is perfectly adherent to the related requirement stated by system specification.

In general, the traceability and verification of compliance can be performed as follows. Using sequence diagrams, the execution process can be statically analyzed from the external activation of an operation (by a 'manual command', as reported in Use-case diagrams) to the reaching of a stable process state, e.g. the one that follows the sending of a message to an application process. A proper package named 'Scenarios', containing all scenario sequence diagrams, has been created for the traceability analysis. Each 'composed' or 'high-level' sequence diagram represents a scenario of, e.g. Start of Mission, Hand-Over, Emergency, and so on.

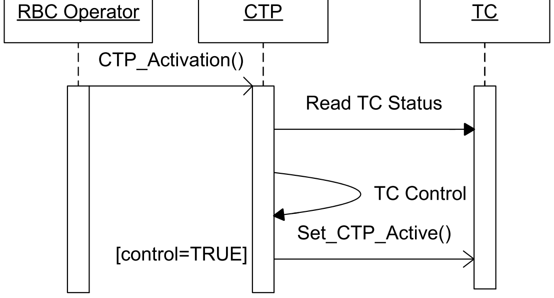


Figure 7: Sequence diagram for the CTP logic process.

By checking sequence diagrams, we discovered that in some cases the controls on the feasibility of a manual command were missing. This was easy to visually check on the diagrams as well as to be automated by defining a rule according to which such control must be performed for each command received by a process. In particular, the check allowed to detect a missing control in the ES process causing the RBC to safely shutdown when the user tried to send an emergency message to a non-existing train number. Usually, such errors are revealed in the functional testing phase, while the approach described in this paper allowed test engineers to detect them earlier in system V&V process. Furthermore, by tracing sequence diagrams into functional requirements, we discovered some pieces of code related to never referenced operations, which were inherited from an early specification: an application of refactoring, in this case, consisted in removing such pleonastic operations and attributes. Sequence diagrams also prove very useful for refactoring, of which we present in the following a quite general example. From the analysis of sequence diagrams, we realized that most of the RBC logic code was made up by condition verification statements. Such statements are clearly explicated in sequence diagrams, which also show their expected weight in terms of process interactions and thus of elaboration cycles (given the fixed policy, each interaction with an already scheduled process requires a further elaboration cycle). For instance, 'lighter' condition verifications only need an access to one or more internal attributes; 'heavier' ones involve the interaction of two or more processes, and can last several elaboration cycles in the fixed process scheduling scheme. Therefore, by properly grouping and shifting conditions we were able to predict and minimize the average number of controls performed by the RBC, with a significant improvement in system performance, which directly impacted on the number of trains the RBC was able to manage (final gain was about the 30%). For instance, if a TC is involved in an emergency condition, there is no need to check its occupation status: as stated by the functional requirements, immediately after the emergency is detected, an emergency message must be sent to the train. Thus, it would be natural to set such control as the first to be performed. However, the probability of an emergency is very low, and such a check would be false most of the times, wasting elaboration time. Therefore, starting from the consideration that an emergency must always include all the TC of a track section (at least 3), a more effective design review consisted in adding a new process named Track Section and making the MA process perform a single check on this process instead of performing it for each TC.

Please note that real-time constraint are not taken into account in the specific application (ERTMS/ETCS trackside subsystem) as the on-board system is responsible for reacting to excessive delays in receiving messages from the RBC. The reaction consists in the application of service or emergency brakes activated by a proper channel vitality monitor, as specified by the standard, which also protects from loss of radio packets due to GSM-R unavailability. Therefore, the delay in sending messages by the trackside subsystem only has an impact on system performance (which is tested apart) but not on its safety.

4.2.4 State diagrams

State diagrams show the transitions of the state variables of the logic processes against the possible triggering events. The CTP state diagram shown in Fig. 8 has been simplified to be self-explaining. By checking the diagram it is immediately clear that for the CTP process all input conditions at any state have been considered, and there are no unreachable or deadlock states (which have been detected and corrected for some other processes). In the design phase, the state machine of each process of the Radio Block Center has been kept enough

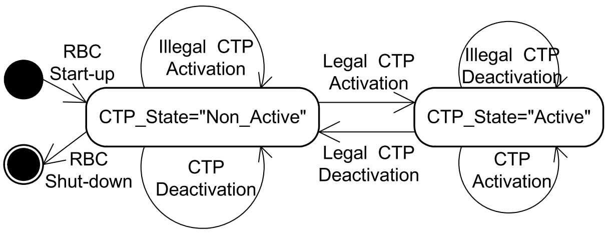


Figure 8: State diagram for the CTP logic process.

simple to be easily checked manually (this is part of the 'design for verifiability' approach which is widespread in safety-critical system development – even though the state machine associate to each process is quite simple, the complexity of system behaviour requires non-straightforward interactions among processes, whose extensive verification can only be accomplished by simulation). In case of more complex state diagrams, automated approaches for property verification (deadlock, liveness, safety, etc.) on UML state diagrams can be applied, as reported in literature (see e.g. [25, 26]).

State diagrams prove very useful in checking the correct implementation of the state machine related to each process, being suitable to informal, semi-formal and formal means of analysis. As for the informal approach, this was facilitated by the fact that the RBC state machine associated to a certain procedure was often completely specified in high-level system requirements. In all this cases, including, e.g. Hand-Over process, the direct comparison has been quite straightforward. On the formal side, a model-based analysis of the HO procedure has been possible by combining the state diagrams of two adjacent RBCs. This enabled verifiers to perform a combined check of the state-space evolution of both RBCs, allowing them to find and correct several errors which were very difficult to find by plain code inspection or simulation.

5 CONCLUSIONS, LESSONS LEARNT AND FUTURE WORKS

In this paper, we have reported our experience with a reverse engineering approach aimed at modeling the (legacy) control software of critical systems for verification and refactoring purposes. The approach allows code verifiers to manage the complexity of critical control software by understanding and analyzing in detail its structure and behavior; therefore, we believe it should be advantageously integrated in the verification process of critical systems developed or re-engineered using an object-based or similar approach. The main advantages of the approach are that:

- It provides a way to facilitate and improve traditional code inspection/walk-through analyses. In fact, our experience proved that the support of a model enhanced effectiveness and proved both more efficient and less error prone with respect to traditional less systematic approaches.
- The availability of a UML model of the software enables further analyses to be performed prior to code execution, including functional verification (exploiting diagram-requirements traceability), and reliability/performance optimization.

Having experienced the automated reverse engineering facilities of CASE tools, we believe that the approach works best if the reverse engineering criteria are tailored to the specific application. Therefore, the translation from legacy languages to C++/Java in order to exploit existing tools is very likely to produce unsatisfactory results. According to our experience, the effort should rather be concentrated on building easily readable/analyzable diagrams directly from the logic code. Once the reverse engineering rules have been defined, the main task of construction and maintenance of the UML model becomes quite trivial and possibly automatable.

The application of the approach to the Radio Block Center, a complex real-world case-study, allowed test engineers to significantly enhance both the effectiveness and efficiency of the usual static verification phase, traditionally performed by code inspection. With respect to previous projects of similar complexity, the static verification process was significantly speeded-up and the number of revealed bugs at this stage (before any dynamic analysis) was more than double (the final number of discovered errors remaining approximately the same). As a further advantage, the availability of a model also allowed test engineers to reduce the time to diagnose the errors detected during the subsequent functional testing phase.

We are now working on the automation of model construction by formalizing the translation rules from logic code to UML views and implementing them in an automatic tool. The verification of model properties is also partly automatable (reference [10] reports an effort in defining formal techniques, including model checking, to be used on UML diagrams). The traceability and verification of compliance with natural language requirements would obviously benefit from a formalization of the functional requirements. This requires a considerable maintenance effort which is presently under evaluation to estimate the cost/benefit ratio.

Much of the work described in this paper has been performed by hand because at the time it was conceived we were not yet confident that the advantages of the approach were such to justify the development of new tools; the only tool used in practice was the Rational Rose Modeler Edition [9], of which we exploited the sketching facilities and the model consistency verifications, as explained in Section 4. The success story of our experience has undoubtedly given a strong impulse to new tool developments, which are currently in progress. As described in Section 4.2.3, a missing control on the actability of a manual command could be discovered by parsing the sequence diagrams with an automatic tool, instead of using the visual and error prone human check. Automatic verification of state diagrams, as introduced in Section 4.2.4, also seems viable [25]. In order to achieve these objectives, a stronger formalization based on MDA [32] is also in progress, together with the systematic integration of the approach within the company standard practice for the certification of new products.

REFERENCES

[1] Heath, W.S., *Real-time Software Techniques*, Van Nostrand Reinhold: New York, 1991.

[2] CENELEC: EN 50126 Railways Applications – The specification and demonstration of Reliability, Maintainability and Safety (RAMS), 2001.

[3] De Nicola, G., di Tommaso, P., Esposito, R., Flammini, F., Marmo, P. & Orazzo, A., A grey box approach to the functional testing of complex automatic train protection systems. *LNCS*, **3463**, pp. 305–317, 2005.

[4] Fowler, M. *et al.*, *Refactoring: Improving the Design of Existing Code*, 1st edn, Addison-Wesley Professional, 1999.

[5] Chikofsky, E.J. & Cross, J.H., Reverse engineering and design recovery: a taxonomy. *IEEE Software*, **7(1)**, 1990. doi:10.1109/52.43044

[6] Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F. & Traverso, P., Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, **10(4)**, pp. 361–380, 1998. doi:10.1007/s001650050022

[7] UNISIG ERTMS/ETCS – Class 1 Issue 2.2.2 Subset 026, 2001.

[8] OMG Unified Modeling Language: http://www.omg.org/uml

[9] Rational Corporation: http://www.rational.com

[10] Bondavalli, A., Fantechi, A., Latella, D. & Simoncini, L., Design validation of embedded dependable systems. *IEEE Micro*, **21(5)**, pp. 52–62, 2001. doi:10.1109/40.958699

[11] Mirandola, R. & Cortellessa, V., UML Based Performance Modeling of Distributed Systems. *Proceedings of UML*, pp. 178–193, 2000.

[12] Briand, L. & Labiche, Y., A UML-based approach to system testing. *In Journal of Software and Systems Modeling*, **1(1)**, pp. 10–42, 2002.

[13] Astels, D., Refactoring with UML. *Proc. of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pp. 67–70, 2002.

[14] Tonella, P., Torchiano, M., Du Bois, B. & Systä, T., Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, **12(5)**, 2007, DOI 10.1007/s10664-007-9037-5, http://www.springerlink.com/content/30881032523123r2 doi:10.1007/s10664-007-9037-5

[15] Visual Paradigm Code Reverse: http://www.visual-paradigm.com/VPGallery/codeengine/CodeReverse.html

[16] Interactive Objects ArchStyler: http://www.interactive-objects.com/products/arcstyler

[17] Cung, A. & Lee, Y.S., Reverse Software Engineering with UML for website maintenance. *IEEE Proceedings of Working Conference in Reverse Engineering'00*, pp. 100–111, 2000.

[18] Riva, C., Selonen, P., Systa, T. & Xu, J., UML-based reverse engineering and model analysis approaches for software architecture maintenance. *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04).*

[19] Briand, L., Labiche, Y. & Leduc, J., Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, **32(9)**, pp. 642–663, 2006. doi:10.1109/TSE.2006.96

[20] Rady de Almeida Jr, J., Batista Camargo Jr, J., Abrantes Basseto, B. & Paz, P., Best practices in code inspection for safety-critical software. *IEEE Software*, **20(3)**, pp. 56–63, 2003. doi:10.1109/MS.2003.1196322

[21] Verimag IF: http://www-verimag.imag.fr/~async/IF/index.html

[22] Telelogic Logiscope: http://www.telelogic.com/logiscope/

[23] Bernardeschi, C., Fantechi, A., Gnesi, S., Mongardi, G. & Romano, D., A formal verification environment for railway signaling system design. *Formal Methods in System Design*, **12**, pp. 139–161. 1998. doi:10.1023/A:1008645826258

[24] Gnesi, S., Latella, D., Lenzini, G., Abbaneo, C., Amendola, A. & Marmo, P., An automatic SPIN validation of a safety critical railway control system. *IEEE International Conference on Dependable Systems & Networks*, pp. 119–124, 2000.

[25] Dong, W., Wang, J., Qi, X. & Qi, Z.-C., Model checking UML statecharts. *Eighth Asia-Pacific Software Engineering Conference (APSEC 2001)*, pp. 363–370, 2001. doi:10.1109/APSEC.2001.991503

[26] Gnesi, S., Latella, D. & Massink, M., Model checking UML statechart diagrams using JACK. *The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pp. 46–55, 1999.

[27] Havelund, K., Lowry, M. & Penix, J., Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, **27(8)**, pp. 749–765, 2001. doi:10.1109/32.940728

[28] Abbaneo, C., Flammini, F., Lazzaro, A., Marmo, P., Mazzocca, N. & Sanseviero, A., UML Based Reverse Engineering for the Verification of Railway Control Logics. *IEEE Proceedings of Dependability of Computer Systems (DepCoS'96)*, Szklarska Poręba: Poland, pp. 3–10, May 25–27, 2006.

[29] Source Dynamics Source Insight: http://www.sourceinsight.com/

[30] Agile Modeling: http://www.agilemodeling.com/

[31] Egyed, N., Medvidovic: Consistent Architectural Refinement and Evolution using the Unified Modeling Language. *Proceedings of ICSE 2001*, Toronto, **3463**, pp. 305–317, 2005.

[32] OMG Model Driven Architecture: http://www.omg.org/mda/