A Model-Based Approach to the Design, Verification and Deployment of Railway Interlocking System

Arturo Amendola[1], Anna Becchi[2], Roberto Cavada[2], Alessandro Cimatti[2], Alberto Griggio[2], Giuseppe Scaglione[1], Angelo Susi[2], Alberto Tacchella[2], and Matteo Tessi[1]

[1] RFI Rete Ferroviaria Italiana – Osmannoro, Firenze, Italy
amendola.arturo@yahoo.com, {m.tessi,g.scaglione}@rfi.it
[2] Fondazione Bruno Kessler – Povo, Trento, Italy
{abecchi,cavada,cimatti,griggio,susi,atacchella}@fbk.eu

Abstract. This paper describes a model-based flow for the development of Interlocking Systems. The flow starts from a set of specifications in Controlled Natural Language (CNL), that are close to the jargon adopted in by domain experts, but fully formal. From the CNL, a complete SysML specification is extracted, leveraging various forms of diagrams, and enabling automated code generation. Several formal verification methods are supported. A complementary part of the flow supports the extraction of formal properties from legacy Interlocking Systems designed as Relay circuits. The flow is implemented in a comprehensive toolset, and is currently used by railway experts.

Keywords: Model-Based Design · Interlocking Systems · Functional Specifications · Code Generation · Formal Verification.

1 Introduction

Functional specifications of complex embedded systems are often written in natural language, but can be ambiguous and subject to different interpretations. This phenomenon emerges in several domains such as avionics and automotive, and is particularly evident in the specification of railway Interlocking (*IxL*) systems. Different regulations and technical specifications expressed in natural language documents, together with complex legacy relay electrical diagrams, have to be reconciled and interpreted to design and evolve digital systems. Such representations usually fail to provide a high-level information about the overall system logics since the design and the actual implementation of the system is highly interconnected. Moreover, the work on these documents requires a strong legacy domain knowledge which is vanishing in these days.

All these aspects contribute to produce different interpretations by Interlocking system suppliers, making railway infrastructure managers weaken or even losing their knowledge of the systems, and ultimately get locked-in to the specific supplier providing the system instances.

In this paper, we propose a Model-Based, tool-supported methodology for the specification, implementation and verification of interlocking systems for the Italian Railway Signaling network. The aim is to ensure product standardization, smooth specification of requirements, and automated code generation and verification of the system. Moreover, this research is intended to support a systematic strategy for the migration of legacy relay systems to computer-based systems.

The approach has several distinguishing features. First, the approach relies on a Controlled Natural Language (CNL) to support railway experts having deep knowledge on regulations and provisions, who are not trained in formal methods, in writing the specification of interlocking procedures using their own language. The CNL, in fact, has been defined to be very close to the jargon adopted by domain experts, yet it is unambiguous. From the CNL, models in *SysML* and C/Python code are automatically generated, thus retaining full traceability. The methodology does not allow to manually modify the automatically generated models and code. Second, the interlocking logics are generic, i.e. specified on station types rather than on a single station. The configuration process with respect to a specific station, also model-based, is out of the scope of this paper. This poses the problem of analyzing not only the interlocking procedures once instantiated on a single station, but also the correctness of the generic procedures with respect to a class of stations. Third, in order to manage legacy, the approach supports the digitalization of relay circuit documents, and the automated extraction of formal models and properties, to allow model checking and co-simulation [2, 6–8]. Finally, the validation of the interlocking logic is supported by a number of formal verification engines, ranging from CNL checkers, to single FSM analyzers, to different model checking engines.

The project is ongoing: the proposed methodology and support tools are currently in use by domain experts, and various kinds of verification engines are being integrated and optimized to specific properties of interest.

The paper is structured as follows. Section 2 introduces the overall approach. Section 3 covers the formalization of the railway specifications; Section 4 their modelling in *SysML*, Section 5 describes the transformation of *SysML* into code. Section 6 discusses the management of legacy relay *IxL*, and Section 7 outlines the verification approach. Section 8 concludes the paper.

2 The Specification and Validation Approach

The Model-Based approach aims at guiding and supporting the analyst from the definition of an informal specification of the system to its formalization, verification and validation and deployment. As shown in Figure 1, it is based on two independent confluent flows. The entire process can be summarized in three main steps:

– Flow A on the left; the formalization of an input set of documents related to the Interlocking system, such as natural language specifications, topological and signalling data about railroad tracks and stations, performed by

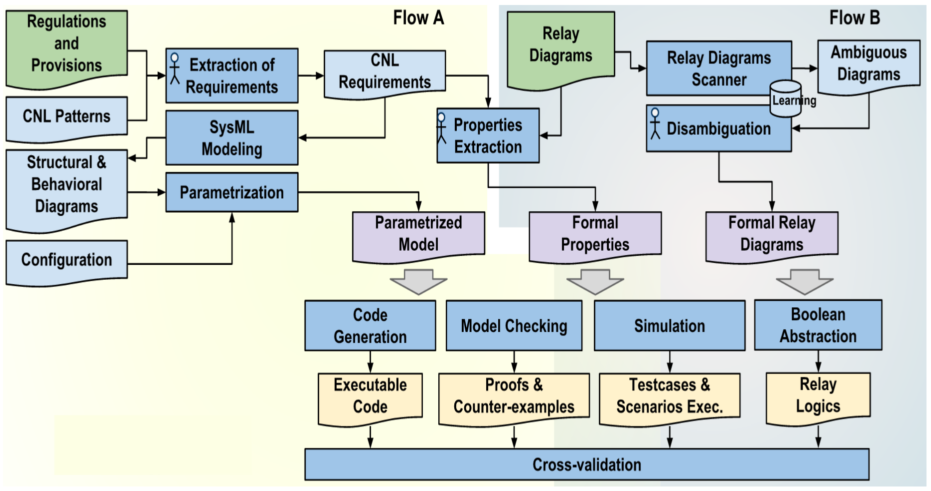


Fig. 1. The general approach

the experts, the automated modelling of the requirements, the automated generation of code and its verification.

– Flow B on the right; the automated extraction and formalization of legacy relay logics, represented by line schematics.
– The artifacts produced in Flows A and B are then used by the analysis steps such as testing or simulation and cross-validation among the different artifacts.

The first step of Flow A aims at the elicitation of domain knowledge through the manual translation of informal textual railways regulations and provisions, into a set of specifications in CNL using a set of predefined specification patterns. The CNL specifications are then automatically formalized and modeled as Block Definition and State Machine *SysML* diagrams, to capture the overall taxonomy of railway elements (together with the structure of relationships among them), and the behavior of the railway logics, respectively. The definition of a computational model allows the specification of a formal semantics for the *SysML* model to enable formal verification, simulation and automatic code generation. The railway logic is abstract, i.e. not referring to a particular configuration of railway stations. Abstraction is allowed by parametric cardinalities in relations among functional blocks. An abstract logic has none or partial configuration, whilst a complete configuration makes the logic concrete. Concretization allows the generation of optimized code tailored for specific configurations, and the application of standard techniques for performing formal analysis such as verification. However, although very challenging, being able to perform formal analysis on abstract models is a prime project goal, as it allows for the distribution of formally verified models prior to their instantiation.

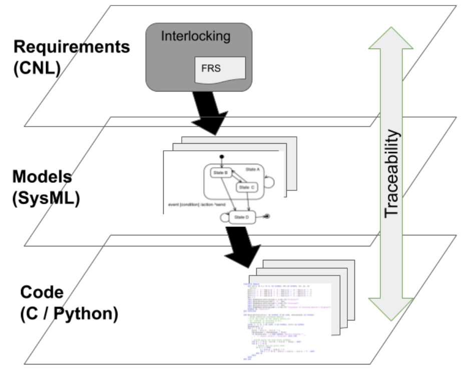


Fig. 2. The three levels development process

Flow B aims at eliciting the control logic that is implicitly implemented in legacy relay circuits. After its formalization, this knowledge is used to produce safety and functional properties, test cases and scenarios, a boolean representation of the relay logics. Given the complexity, size and number of relay diagrams, the formalization of these artifacts is performed using a semi-automatic approach based on a combination of image recognition techniques. This produces a formally defined model that allows the application of formal methods.

Finally, the artifacts produced in the two flows allow the co-simulation with formal model from flow A, and other cross-validation techniques.

The development process for the Interlocking system follows a *V* model and is based on three levels of activities and related artifacts as described in Figure 2. In the first level the Functional Requirements Specifications (FRS) are manually extracted from the domain documents, in the second level the *SysML* models are automatically derived from the CNL specification, at the third level the C/Python code is automatically produced from the models. A complete traceability is maintained among the artifacts in the three levels allowing the designer to connect the generated code to the requirements that originated it.

The entire process is supported by the AIDA toolchain, based on the Eclipse platform, that allows the specification of the requirements and their transformation into *SysML* models and code. The toolchain includes tools for the formal verification of the artifacts produced by the methodology and for the extraction, representation and formalization of relay circuits to support flow B.

3 Formalization of the Railway Specifications

The requirements for the *IxL* is extracted from a set of railway domain documents such as diagrams describing the structure of a railway station, railway regulations and provisions. The resulting Functional Requirements Specification (FRS) document consists of a set of cards each one containing the definition of the structure and behavior of a so called Class of Logic that is the representation of the elements of a railway that are relevant for the *IxL*, such as the safety logics of physical devices (e.g. *Railroad Switch*, *Train Track*, different kinds of *Signal*s), or the safety logics of higher level entities (e.g. *Train Itinerary*, *Section Block*). The specification of the FRS is performed by domain experts carrying out three main activities:

– Analysis of the railway documents to identify the relevant aspects and concepts that should become part of the Functional Requirements Specifications;
– Specification of the FRS as a collection of classes using the CNL that allows the expert to specify in a textual form the structure of the class and its behavior through the definition of the associated Finite State Machine using a language that is very close to their jargon;
– Definition of tracing links between the resulting FRS and the railway documents from which it originated.

In order to guide the specification of the FRS, a set of CNL patterns have been defined also analysing specification documents produced by the experts in previous projects. In particular, a document describing a class contains two main sections: one for the definition of the attributes of the class, the other for describing the behavior in terms of the states and transitions of the Finite State Machine (*FSM*) of the class. Each transition in the *FSM* is described using the CNL and is characterized by a set of triggering conditions and a set of effects that specifies the actions to be performed by the class when changing its state. An example of triggering conditions and effects are:

```
conditions: verify that the control Position is not equal to Normal
            verify that cdb is free and not locked
            verify that the timer TOWait is expired
...
effects: assign to control Position the value Normal
         activate the timer TOWait
```

The specification of the FRS by the experts is supported by the modeling tool AIDA that provides an editor and a set of syntactic and semantic checks for the correct writing of the classes in CNL.

3.1 Architecture of the *IxL*

The *IxL* is made of a set of Class of Logic, each possibly interacting with:

– the environment: receiving *Manual Commands* which are asynchronous events sent by the user; sending state information for visualization; reading hardware signals from the station plant; writing actuations to the station plant.
– other classes: synchronously reading/writing state of linked classes; sending/receiving *Automatic Commands* which are asynchronous events possibly carrying data information.

The classes are organized hierarchically by linking them in a *use-a* relationship. Relations among links are possible within a class, e.g. class *Itinerary* having a pair $\langle Signal, TrackSection\rangle$. The hierarchy is made by constraining the possible interactions among the classes. Only higher-level classes can send *Automatic Commands* to lower-level classes; only classes at higher or same level can write state of same or lower-level classes; a class can read the state of all other classes.

3.2 Structure of a class

Each class has an interface to rule the above described possible system interactions (*Manual Commands*, *Automatic Commands*, I/O from/to the plant), an internal state and a deterministic *FSM*. The internal state is made of: *configuration parameters* and *configuration lists* (which contain links to other class instances to implement *is-a* relationship), both of which are fixed when instantiating the class; variables, whose type can be boolean, integers, literal sets, ordered literal sets, timers and counters; the current state of the associated *FSM*. *functional macros* can be defined for the information-hiding of reading state. *procedural macros* can be defined for the information-hiding of writing state and sending *Automatic Commands*.

The *FSM* has a set of named states, a set of initial transitions to determine the initial state, and the set of transitions among states. Each transition is characterized by a *guard*, an *effect*, a *priority*. Guards are functions of reception of manual and automatic commands, the internal state (including visible state of linked instances and functional macros). Effects contain state assignments (including accessible sate of linked instances), sending of *Automatic Commands* and invocation of procedural macros. Priority setting is simplified for the modeler by defining four categories of priority (from the railways domain), and by associating each transition to a category.

3.3 Execution model

The execution of the *IxL* logics is performed by a periodic task, which at each cycle performs the sequence depicted in Figure 3:

1. Reads and latches inputs from the user (*Manual Commands*) and from the plant (I)
2. Executes the *IxL* logics (E)
3. Writes outputs for the user interface and for the plant (O)

The Execution phase (E) is carried out by a *Scheduler*, which executes all the active class instances according to three phases:

Manual Phase all instances that received one or more *Manual Command* are activated. Each instance can process a single *Manual Command* by executing a single transition, if the corresponding guards are enabled. If no transition can be executed, then the *Manual Command* is lost.

State Phase every instance having a transition enabled in the current state is activated. Each instance processes a single transition. The phase terminates when all activated instances have performed one state transition.

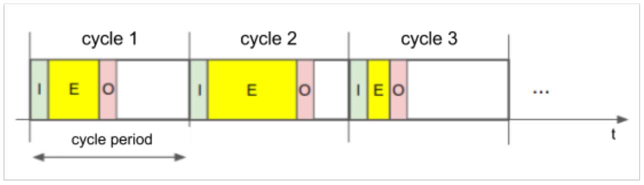


Fig. 3. Delta Cycle

Automatic Phase all instances that received one or more *Automatic Command* are activated. Since in the effects of any transition other *Automatic Commands* can be sent, in this phase a given instance may get activated multiple times. If a command cannot be processed (as unexpected or as the truth value of guards do not enable any transition) the command is lost. This phase is repeated until fixpoint is reached, i.e. all instances reaches a quiescent state where no further *Automatic Commands* are sent.

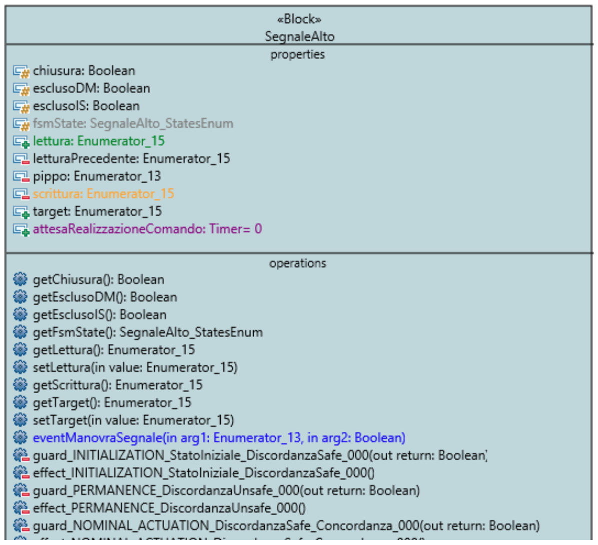
The execution is deterministic. The three phases are executed in the same order; the instances in each phase are also fired in the same order, although this order is not known to the *IxL* designer. The execution of each *FSM* is also deterministic, as the priority uniquely determines the transitions that the scheduler activates. The execution runs to completion by construction: in the *Manual Phase* and in the *State Phase* at most one transition per instance is executed; in the *Automatic Phase*, structural constraints on the system hierarchy assure the absence of execution loops.

4 Automated *SysML* model generation

4.1 *SysML* for Model-Based System Engineering Applications

SysML (System Modeling Language - `https://sysml.org/`) is a generic language for architecture design widely adopted in Model-Based System Engineering (MBSE) applications It supports specification, analysis, design, verification and validation of a wide range of systems, such as hardware, software, information, processes, resources and structures. *SysML* is a dialect derived from *UML* 2, defined as a *UML* profile. In 2006 the Object Management Group (OMG) adopted OMG *SysML* as a standard, and manages it maintenance. The current version is OMG *SysML* 1.6. *SysML* diagrams also in *UML* are: *Activity*, *Block Definition* (*Class* in *UML*), *Internal Block* (*Composite Structure* in *UML*), *Sequence*, *State Machine*, and *Use Case*. Diagrams that are unique in *SysML* are *Requirement* and *Parametric*. Diagrams in *SysML* are mainly split into two groups:

Structural Diagrams Allow the definition of the system entities to be modelled, relations among them, and their structural content, like properties, operations and other member attributes.



```
«Block»
SegnaleAlto
properties
chiusura: Boolean
esclusoDM: Boolean
esclusoIS: Boolean
fsmState: SegnaleAlto_StatesEnum
lettura: Enumerator_15
letturaPrecedente: Enumerator_15
pippo: Enumerator_13
scrittura: Enumerator_15
target: Enumerator_15
attesaRealizzazioneComando: Timer= 0
operations
getChiusura(): Boolean
getEsclusoDM(): Boolean
getEsclusoIS(): Boolean
getFsmState(): SegnaleAlto_StatesEnum
getLettura(): Enumerator_15
setLettura(in value: Enumerator_15)
getScrittura(): Enumerator_15
getTarget(): Enumerator_15
setTarget(in value: Enumerator_15)
eventManovraSegnale(in arg1: Enumerator_13, in arg2: Boolean)
guard_INITIALIZATION_StatoIniziale_DiscordanzaSafe_000(out return: Boolean)
effect_INITIALIZATION_StatoIniziale_DiscordanzaSafe_000()
guard_PERMANENCE_DiscordanzaUnsafe_000(out return: Boolean)
effect_PERMANENCE_DiscordanzaUnsafe_000()
guard_NOMINAL_ACTUATION_DiscordanzaSafe_Concordanza_000(out return: Boolean)
```

Fig. 4. Fragment of `SysML::Block`. Attributes' colors denote different typologies

Behavioral Diagrams Define the dynamic behaviour of the entities, both in terms of internal dynamics and interaction among them.

SysML allows meta-model extensions through the use of *stereotypes*, to create new elements of the meta-model derived from the existing ones.

4.2 Modeling of the abstract *IxL* System

The abstract *IxL* system is modeled as an Object-Oriented system of classes communicating by mean of *Operations*, and whose execution is ruled by the Scheduler and scheduling schema described in Section 3.3.

The automatic translation process takes the FRS as input, and generates a *SysML* model as output. The structural part of each class is translated as `SysML::Block`. Each class attribute is translated to a block's `UML::Property`, which have been stereotyped to model its domain-specific typology, like *Manual Commands*, *Automatic Commands*, parameters, linked instances, variables, plant inputs and outputs, etc. Primitive types exploit basic *UML* types (boolean, integer, etc.), while the other types like Timer, Counter, Enumeratives, Records, etc. are generated into a dedicated `UML::Package`. A stereotyped `UML::Operation` is used to model each attribute's getter, setter, macro, guard function and effect procedure. Each *Manual Command* and *Automatic Command* is translated into a `UML::CallEvent`, and a `UML::Operation` get associated to the event to model its behaviour. Fig. 4 shows a sample of a `SysML::Block` taken from the domain.

Each class's *FSM* is translated as a `UML::State Machine`, which is associated to the corresponding `SysML::Block`. Each state of the *FSM* is translated as a `UML::State`, each transition is translated with a `UML::Transition` extended with stereotypes that provide information about priority, its category and scheduling phase. The `UML::Transition`'s guard and effect consist of `UML::Operation` calls to the operation containing the guard expression and effect procedure, respectively (Fig. 5).

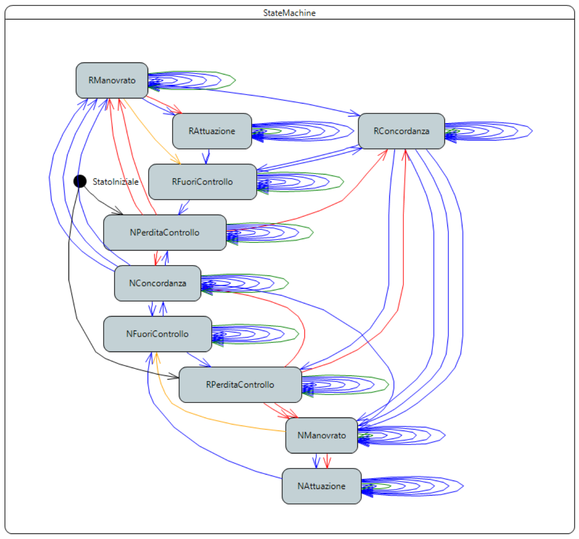


Fig. 5. `UML::State Machine` of the logics of a *Railroad Switch*. Transition colors denotes the typology

Guards expressions and effects statements are translated into a concrete language that is formally-defined subset of C extended with an object-oriented notation, and corresponding Abstract Syntax Tree get attached to the *Operations* corresponding to each guard/effect. The following example shows a CNL guard and the automatically translated code:

```
all the following
 verify that the variable PowerSupply is equal to true
 verify that the control Position is not equal to Normal
 verify that cdb is free and not locked
 verify that the timer TOWait is expired

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
return (
   (PowerSupply == true) &&
   (! (Position == Normal)) &&
   all(CdBRec iter: cdb, iter.cdb.getFree() && ! iter.cdb.getLocked()) &&
   TOWait.isExpired())
```

The evaluation context is the class instance containing the `UML::Operation` associated to the guard, so *PowerSupply*, *Position cdb* and *TOWait* are instance's attributes. Notice the Object-Oriented dotted notation to call linked instances' *Operations*, like in *cdb.getLocked()*. As *cdb* is a list of records, functional flavors for quantifiers (*all(...)*) are provided by the concrete language.

5 Automated Code Generation

The code generation process takes as input the *SysML* model and produces two different artifacts:

- A first version of the code, written in the Python programming language, is intended for debugging and simulation purposes, running on a host machine.
- The second version, written in the C programming language, is intended for deployment on the target platform and contains the code that will be effectively executed on the field.

The overall structure of the two versions is very similar and will be explained in the next subsection. The details of the two implementations are, of course, quite different: whereas the Python version fully leverages the dynamic nature of its execution environment, the C version allocates every data structure either statically or at initialization time, as soon as the configuration parameters are known.

5.1 Structure of the *IxL* code

In both versions of the *IxL* code each Class of Logic is translated to a separate module. In the Python version, to each Class of Logic corresponds a single Python class derived from a common base class. In the C version, each Class of Logic is translated into a separate compilation unit, that is a `.h`/`.c` file pair.

The code generated for each Class of Logic implements the interface of the Class, as described in the corresponding Block Diagram of the *SysML* model, and the execution logic of the associated Finite State Machine. In more detail, each module contains:

- the declaration of the local variables of the state machines, as extracted from the *SysML* model;
- for each variable, a pair of getter and setter methods;
- the interface procedures `init` and `exec`, whose job is, respectively, to initialize the variables of the FSM and to pick the correct transition for the state machine, by checking the current state and the relevant guards in turn;
- for each transition specified by the state machine, a *guard* function that checks if the triggering conditions of that particular transition hold, and an *effect* procedure that performs the corresponding set of effects.

The generation process itself is driven by a template engine that operates on a data structure which is preliminarly populated directly from the *SysML* model.

The code produced by the generation process is parametric on the number of instances of each Class of Logic and on the inter-relationships between the various Classes. In order to be executed, this code must be instantiated according to a specific configuration, as described in subsection 5.3.

5.2 Execution of *IxL* code

Each instance of a Class of Logic represents a distinct *process*. The execution of such processes is orchestrated by a *scheduler* whose code is generic, that is it does not depend on the specific station configuration.

The scheduler execution is based on a loop (delta cycle) with three distinct phases: *input reading*, *elaboration*, and *output writing*. At every cycle, the elaboration is performed on the inputs extracted from the snapshot taken at the previous step (input latching).

In the elaboration phase each process is activated in turn by the scheduler according to a fixed order, which is specified ahead of time and is part of the configuration data, so that it can be customized depending on the characteristic of each specific station.

The execution of the processes takes place according to a policy of *cooperative scheduling*: the scheduler halts its execution until each process yields back control. In this way, execution of the *IxL* is guaranteed to be deterministic and purely sequential, which eliminates every possible problem stemming from concurrency.

Inter-process communication is realized through the exchange of *Automatic Commands*, as described in Section 3. The scheduler is responsible for forwarding the *Automatic Commands* from the source process to the target one.

Each execution cycle is divided in the three phases described in Section 3.3: *manual phase*, *state phase*, and *automatic phase*.

5.3 The code configuration mechanism

As pointed out above, the *IxL* code is parametric: the number of instances of each Class of Logic, and the mutual references between the various instances, are not known at code generation and must be supplied at a later time in order to instantiate the code for a particular station. This data is globally referred to as the "configuration".

The instantiation mechanism is different for Python and C code. In the first case, the Python runtime reads the configuration at startup from a JSON file and dynamically fills the corresponding data structures. This is appropriate for execution on a host machine, where access to a filesystem can be taken for granted. On the target platform, however, the resources available to the *IxL* code are minimal, and are provided by a tailor-made OS. In this case the configuration data is stored in a binary file with the same layout as a direct memory dump of the corresponding C data structures. This binary file is loaded at system initialization in a specific memory area which is subsequently marked as read-only. The initialization routine of the *IxL* code is then called to set up appropriately the pointers to the configuration data for each instance of a Class of Logic. Various checks are performed to ensure that no pointer remains uninitialized and that configuration data is correctly validated.

6 Automated Formalization of Relay Schematics

Relay diagrams are standard representations of networks of electro-mechanical components. We now describe how to extract the controlling logics left implicit in these circuits, hidden by non-trivial physical laws. The behaviour of the network is induced by the characteristics of the components, such as resistance values, and by both wired and remote connections. As an example, a relay can open or close a switch, possibly belonging to a different circuit, depending on the value of the induced magnetic field.

Firstly, a graphical front-end based on an extension of the DIA diagram editor (`https://gitlab.gnome.org/GNOME/dia`) allows for the digital modeling of relay schemata that are originally provided as paper-based sheets. Digital modeling is done by overlaying components and connections on the top of the scanned images of the original sheets. Components are picked from a palette which is context aware to help choosing from hundreds of components. Picking components, entering data (name and other properties) and construction of electrical connections, is aided by OCR, template matching and deep learning image recognition techniques.

Each component, such as power supplies, switches, resistors and electrically-controlled contacts, is mapped into the library of known electrical elements. Here, each symbol is associated with a set of parameters and specifications. The XML format provides a modular and flexible description for the connections and the remote-controlling interactions between the tagged elements. The front-end provides also some sanity checks on the status of the network, like dangling terminals or connections with incompatible symbols.

The subsequent step is to convert the circuit in a formalism amenable for verification. Following [3], we adopt a methodology to analyze and understand Relay Interlocking Systems, by reduction to a Switched Multi-Domain Kirchhoff Network which is compiled in a hybrid automaton. A compiler implements a component-based translation of the XML file into the HyDI language [5]. The resulting automaton is the synchronous composition of the models of the components: shared variables implement remote discrete interactions, while wired connections are handled locally, by imposing Kirchhoff conservation laws between the voltage and current values at every node.

The independent modeling of single components can be done with different levels of abstraction. Due to internal inertial electro-mechanical phenomena, some components exhibit transient states between stationary condition. The precise approach requires considering complex differential equations defining continuous variables. A more pragmatic approach consists in extracting a hybrid automaton with a piecewise-linear abstraction of the dynamics. Namely, state changes of components exhibiting transient conditions, such as delayed relays, are modeled as discrete transitions happening within a constrained time interval.

The formalization of the relay diagram as a hybrid automaton allows for different kinds of analysis. The expected properties of both the components and the whole network can be checked with the HyCOMP back-end verifier [4]. In addition, it is possible to formally reason on the behaviours of the circuits as the language of the corresponding automata (or by one of their abstractions) for a more thorough comparison with the language of the generated code. The target of such a verification step is to check whether the legacy circuit, while being based on different internal electro-mechanical variables, responds on the railway elements in the same way of the digital implementation.

7 System Verification

The verification activities are carried out at three different levels: (i) verification of structural properties of the components, operating at the *SysML* level; (ii) verification of properties of the generated C code for a specific station/configuration; (iii) verification of properties of the abstract interlocking logic on a given set of configurations.

Structural checks The first level consists of a set of "light-weight" checks on the *SysML* model generated by the CNL translation. These checks are meant to verify structural properties of the model, that are independent from the actual application domain. Examples include absence of deadlock states in the state machines, mutual exclusion among different transition guards, absence of unreachable states and/or unreachable transitions (due e.g. to guards that are always true/false). These checks are essentially local to each generated state machine, and can typically be performed very efficiently. Although conceptually quite simple, they offer a very useful debugging aid to domain experts during the initial phases of development of new Classes of Logic.

Code verification The second level consists on the verification of the correctness of the automatically-generated implementation of the interlocking logic for a specific station with respect to a set of user-specified properties.

We follow an approach based on software model checking, in which the C implementation of the interlocking logic, combined with an abstraction of its execution environment and instantiated for a specific configuration, is translated into a symbolic transition system which can then be formally verified with state-of-the-art model checking tools such as nuXmv [1].

More specifically, the translation from the C implementation to the transition system for verification is performed as follows. First, the generated C code for the Classes of Logic is instantiated according to the specific configuration under verification. All dynamic data structures (e.g. lists and vectors whose size depends on configuration parameters) are statically allocated, and all indirect references (expressed as pointers in the C code) are statically resolved; Second, the code is simplified and specialized according to the configuration: static loops (with upper bounds depending on the configuration) are unrolled, dead code is eliminated, and methods of each Class of Logic are specialised for each of the instantiated objects. Then, a model of the execution environment (properly instantiated for the specific configuration) is added. This consists of a main scheduling loop that executes the different instances in a "scan cycle" mode: input acquisition (abstracted as nondeterministic assignment to the input variables), logic execution (according to the specified scheduling policy), output generation. Finally, the imperative program is compiled to a symblic transition system using standard techniques: inlining of all functions, removal of side effects, generation of a SSA form, symbolic encoding of the control-flow-graph and the program statements into SMT constraints.

An advantage of approaches based on model checking is their capability of producing counterexample traces witnessing the violation of some property. In our flow, such traces are automatically translated into a high-level sequence of commands/controls from the environment (i.e. the actual train station) that can be used to drive the interactive simulator for the interlocking logic, so that the erroneous scenarios can be immediately visualized and understood by the domain experts. Moreover, the same approach can also be used for automatic test-case generation, given a target system state to reach.

The verification at the level of concrete/instantiated code has two drawbacks. First, scalability is a major issue. Even the smallest stations involve hundreds of instances, which quickly become thousands for medium-sized configurations. When performing full specialization of the code, the resulting transition system can significantly blow-up in size and become unmanageable for the model checker. As mitigation strategies, we are adopting various abstraction and simplification techniques, in which the transition system is generated incrementally, via a successive sequence of increasingly-precise approximations, guided by an analysis of the generated spurious counterexamples; ultimately however, the approach will still be limited by the size of the actual configuration. Second, the results of the verification are only valid for one specific configuration, and cannot be easily lifted to different configurations/stations.

Abstract verification The third level of verification consists in tackling the correctness of the interlocking logic independently from any specific configuration. From the formal point of view, this can be formalized as a parametric verification problem, in which each Class of Logic corresponds to a different process type. Although the problem of automatic verification of parametric systems has been extensively studied in the literature, the complexity of the task is significantly beyond the state of the art: current automatic techniques are typically focused on handling distributed protocols specifications, involving very few process types (on the order of 2-4) with a few tens of transition actions. In our context, instead, the parametric description consists of tens of different process types, with hundreds of transition actions. This is an extremely challenging activity, which however has the potential of offering significant advances to the state of the art.

8 Conclusions and Future Work

The paper presents a model-based methodology for the design, deployment and verification of Interlocking systems. The approach has several key feature. First, there is a strong connection among different abstraction levels of the design: Controlled Natural Language for requirements specification, and the automatically generated *SysML* models and C/Python code. Second, the verification is supported by various verification methods. Third, a strong connection with legacy systems: the specification documents of relay legacy systems are automatically modeled and formally analyzed to extract reference specifications.

The methodology is supported by tools that are currently being used by domain experts. In the future, we plan to add new verification and validation techniques specialized to the most common properties, to complement the formal verification techniques with model-based testing techniques for the generated code, and to address the problem of certification.

References

1. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th Intl. Conf, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
2. Cavada, R., Cimatti, A., Micheli, A., Roveri, M., Susi, A., Tonetta, S.: Othelloplay: a plug-in based tool for requirement formalization and validation. In: Bishop, J., Breitman, K.K., Notkin, D. (eds.) Proceedings of the 1st Workshop on Developing Tools as Plug-ins, TOPI 2011, Waikiki, Honolulu, HI, USA, May 28, 2011. p. 59. ACM (2011). https://doi.org/10.1145/1984708.1984728
3. Cavada, R., Cimatti, A., Mover, S., Sessa, M., Cadavero, G., Scaglione, G.: Analysis of relay interlocking systems via smt-based model checking of switched multi-domain kirchhoff networks. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603007
4. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Hycomp: An smt-based model checker for hybrid systems. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st Intl. Conf., TACAS 2015. London, UK. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 52–67. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_4
5. Cimatti, A., Mover, S., Tonetta, S.: Hydi: A language for symbolic hybrid systems with discrete interaction. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011. pp. 275–278. IEEE Computer Society (2011). https://doi.org/10.1109/SEAA.2011.49
6. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Formalizing requirements with object models and temporal constraints. Software and Systems Modeling **10**(2), 147–160 (2011). https://doi.org/10.1007/s10270-009-0130-7
7. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Validation of requirements for hybrid systems: A formal approach. ACM Trans. Softw. Eng. Methodol. **21**(4), 22:1–22:34 (2012). https://doi.org/10.1145/2377656.2377659
8. Ferrari, A., Gori, G., Rosadini, B., Trotta, I., Bacherini, S., Fantechi, A., Gnesi, S.: Detecting requirements defects with NLP patterns: an industrial experience in the railway domain. Empirical Software Engineering **23**(6), 3684–3733 (2018). https://doi.org/10.1007/s10664-018-9596-7