JOT
JOURNAL OF
OBJECT TECHNOLOGY

Automating Model Transformations for Railway Systems Engineering

Nils Weidmann[*], **Shubhangi Salunkhe**[†], **Anthony Anjorin**[‡], **Enes Yigitbas**[*], **and Gregor Engels**[*]
[*]Paderborn University, Germany
[†]DB Netz AG, Germany
[‡]IAV GmbH Ingenieurgesellschaft Auto und Verkehr, Germany

ABSTRACT Model-Based Systems Engineering (MBSE) enables system development and analysis on a suitable level of abstraction. In the context of railway systems engineering, system verification is of major importance as software failures can cause serious damage. At DB Netz AG, a railway infrastructure manager that operates large parts of the German railway system, the challenge of enabling both high-level system modelling *and* formal system verification is addressed by employing SysML, a widespread systems modelling language, and Event-B, a formal systems modelling language particularly suited for automated system verification. In the currently applied completely manual development process, engineers (i) create models using SysML, (ii) translate relevant parts of these models to Event-B for verification, (iii) possibly improve the Event-B models based on verification results, and finally (iv) reflect these improvements in the original SysML models. This process is both tedious and error-prone, clearly indicating a need for an increase in the level of automation.
In this paper, we argue that steps (ii) and (iv) can be viewed as a coupled forward transformation and a backward synchronisation, respectively, as the SysML models cannot be completely reconstructed from their Event-B counterparts. Exploiting this observation, we demonstrate that steps (ii) and (iv) can be suitably automated using a *bidirectional* transformation (bx) language. With Triple Graph Grammars (TGGs) as a rule-based bx language, we establish a tool chain connecting the modelling tools used at DB Netz AG for SysML and Event-B. We show the feasibility of our automation solution by solving three representative case studies provided by DB Netz AG. Based on these case studies, we conduct a qualitative evaluation via semi-structured interviews with domain experts.

KEYWORDS bidirectional model transformations, model-based systems engineering, tool integration

1. Introduction

Model-Based Systems Engineering (MBSE) advocates using models to support all phases of systems engineering by introducing a suitable level of abstraction and automation (Hoang 2013). Models can be used to improve communication between stakeholders, enable traceability across varying levels of abstraction, and boost productivity via code generation and other model transformations (Sendall & Kozaczynski 2003). MBSE, already standard practice in domains such as defense and aerospace engineering, is also gaining popularity in the *railway* domain, where MBSE tools are used to create a standardised system architecture, functions, and interfaces for railway systems (Amendola et al. 2020). As failures of *safety-critical systems* such as railway systems can lead to serious damage, a formal verification of expected system behaviour is very important. To limit roll-back and re-implementation costs, the verification and validation of safety requirements should be integrated into the early stages of development. (Freund 2012)

At DB Netz AG, a railway infrastructure manager that operates major parts of the German railway system, an MBSE

JOT reference format:
Nils Weidmann, Shubhangi Salunkhe, Anthony Anjorin, Enes Yigitbas, and Gregor Engels. *Automating Model Transformations for : Railway Systems Engineering.* Journal of Object Technology. Vol. 20, No. 3, 2021. http://dx.doi.org/10.5381/jot.2021.20.3.a10

process is to be introduced for interface standardisation as part of the EULYNX initiative[1]. To support high-level systems modelling, the PTC Integrity Modeler[2] is used for creating Systems Modeling Language (SysML) (Object Management Group 2019) state machines. While simulation-based testing can be used to identify software faults in SysML models, a formal proof of correctness with respect to expected system behaviour is not supported. To overcome this limitation, Event-B (Abrial & Hallerstede 2007; Hoang 2013) – a formal method for system-level modelling and analysis – is used to verify safety properties via proof obligations for all possible system configurations. As a simulation and verification environment for Event-B, the RODIN platform (Butler & Hallerstede 2007) is used.

The current development process is depicted in Fig. 1 in an informal notation inspired by Stevens (Stevens 2017). In step (i), engineers create state machines (S) as SysML models. Relevant parts of these models are then (step ii.1) translated *manually* by experts in formal methods to semantically equivalent UML-B (Snook & Butler 2006) state machines (U). In step (ii.2), these UML-B models are transformed automatically into Event-B code (E) using the UML-B plug-in of RODIN. The generated Event-B code can then be verified against safety properties. Results and insights gained from the formal verification are reflected in the Event-B code (E') in step (iii), and then have to be manually propagated back to the original SysML models (S') in step (iv).

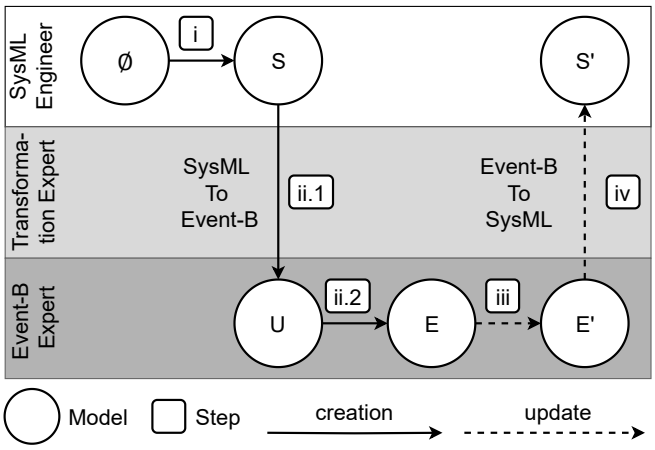


Figure 1 Process Overview

The current process is tedious and error-prone due to the manual transformation from SysML to UML-B state machines, as well as the equally manual backward propagation of corrections in the Event-B code to the original SysML models. The intermediate UML-B representation is not required for the verification itself, but rather a necessary concession to keep the manual transformation manageable, as writing Event-B code directly is challenging. It is moreover impossible to certify these manual steps of the process as there is no transformation specification that could be reviewed by experts.

[1] https://www.eulynx.eu/
[2] https://www.ptc.com/en/products/windchill/modeler

In this paper, we argue that the SysML to Event-B forward transformation and coupled backward synchronisation are best viewed as a Bidirectional Transformation (BX) (Abou-Saleh et al. 2016), as the SysML models cannot be simply reconstructed from their Event-B counterparts. Instead, applied changes in the Event-B model should be (incrementally) propagated back to the SysML model based on the same consistency relation, being a key property of BX languages. We apply Triple Graph Grammars (TGGs) (Schürr 1994), a declarative rule-based approach to BX, to encode the knowledge of practitioners about the concrete application scenario in a consistency relation from which both rules for the forward transformation (initial generation of the Event-B code) and backward synchronisation (propagation of potential fixes to the SysML model) can be automatically derived. At the same time, the generation of an intermediate UML-B model can be completely omitted, merging (ii.1) and (ii.2) into a single transformation step. Traceability information between informal requirements and the modelled system, specifically for safety properties, can further be maintained. We demonstrate the feasibility of our approach by solving three representative case studies provided by DB Netz AG. Based on these case studies, we provide a qualitative evaluation by conducting expert interviews with employees of the company to assess the applicability of the proposed solution in practice. Our contribution is therefore twofold: On the one hand, we present a TGG-based BX from SysML to Event-B as an example for bridging semantic gaps between a semi-formal and a formal language. On the other hand, we investigate and evaluate a practical application scenario of a model-to-model transformation from SysML and Event-B in the railway systems engineering domain. The identified potential and limitations of our approach can be used to drive further research towards improving the practical applicability of model transformation technology.

The remainder of this paper is organised as follows: After providing an overview of related approaches (Sect. 2), we analyse SysML and Event-B to identify the relevant language constructs for the required transformation (Sect. 3). By adding further knowledge about the transformation process at DB Netz AG, we define a consistency relation using TGG rules in Sect. 4. The implemented tool chain is sketched in Sect. 5. Section 6 presents a qualitative evaluation of our approach based on three representative case studies provided by DB Netz AG, used to conduct semi-structured interviews with domain experts. Finally, Sect. 7 concludes the paper and proposes directions for future research.

2. Related Work

Comparable to our contribution and focus in this paper, there have been numerous projects investigating the application of TGGs in an industrial context. Giese et al. present an approach for transforming SysML models to AUTOSAR[3] using TGGs (Giese et al. 2010). In contrast to our application, however, SysML block diagrams are transformed and not state machines. The application domain is also different, i.e., supporting the transition from system design to software design in the

[3] https://www.autosar.org/

automotive domain as opposed to supporting formal analysis for safety requirements in the railway domain. These differences also lead to a different set of relevant challenges: Giese et al. focus on transforming and synchronising large models in a scalable manner, while we focus more on comprehensibility and expressiveness, especially regarding attribute manipulation. Hermann et al. present an approach to translate satellite procedures from one language to another also using TGGs (Hermann et al. 2014). While Hermann et al. face similar challenges as we do, in this case comprehensibility and formal correctness of the transformation, the application domain is of course different (aerospace vs railway). Moreover, the supported development process is simpler than ours as Hermann et al. only require a forward transformation while we require both a forward transformation and a backward synchronisation. Blouin et al. report on their experience of using TGGs to develop a synchronisation layer between different tool environments for the Architecture Analysis and Design Language (AADL), the Open Source AADL Tool Environment (OSATE) textual editor and the Adele graphical editor (Blouin et al. 2014). While Blouin et al. also require true synchronisation (Adele does not cover the entire AADL language), their focus and set of challenges again differs from ours. Blouin et al. have to deal with very large metamodels (i.e., require many TGG rules), and are more concerned about establishing usable tooling (e.g., scalability) and less about comprehensibility and formal correctness. In general, existing industrial case studies with TGGs tend not to focus on a qualitative evaluation, investigating instead quantitative aspects such as scalability of the solution, which might arguably not be the strongest argument for BX in general and TGGs in particular. While previous TGG-based work on solving allocation problems in the software testing domain (Anjorin, Weidmann, et al. 2020) did include a qualitative evaluation, it was limited to a single test engineer and centred on consistency checking not synchronisation.

Concerning our choice of TGGs as a BX language, Anjorin et al. (Anjorin, Buchmann, et al. 2020) provide a recent overview and benchmark of various BX languages, and state that TGGs scale well in practice for transformation and synchronisation tasks. To solve our use case we could have used any equally mature BX language such as BiGUL (Ko et al. 2016). While we cannot (yet) back the following conjecture with any empirical evidence, we suspect that TGGs might be more comprehensible than, e.g., BiGUL in the context of our application scenario as relevant domain experts are familiar with visual modelling languages and the concept of transformation rules, as opposed to a functional approach and Haskell-like syntax. The situation is probably completely reversed for other application domains.

The general problem of SysML not being sufficiently formal has been addressed in different ways by numerous authors. Pais et al. present an approach to transform SysML state machines to Petri Nets (PNs) (Pais et al. 2014) to be able verify formal properties, generate code, visualise and execute the resulting PNs. As the authors use Atlas Transformation Language (ATL) for the transformation it remains unclear how insights gained from the formal analysis are reflected back to the SysML state machines. Huang et al. present a transformation of SysML activity diagrams to PNs also for formal verification (Huang et al. 2020). According to the authors, the execution semantics for Unified Modeling Language (UML) and SysML are not sufficiently precise to be unambiguous and thus they use the foundational UML (fUML) standard to provide a precise semantic definition of SysML activity diagrams. Again it is unclear how changes to the resulting PNs are reflected back to the corresponding activity diagrams. There are several further examples for transformations from SysML to formal languages such as Alloy (Anastasakis et al. 2010), NuSMV (Caltais et al. 2016), Promela (Caltais et al. 2020), or to CSP# processes (Ando et al. 2014). While it is possible to verify the transformation results with state-of-the-art model checker in each case, the subsequent incorporation of findings remains a manual task. In general, we argue that the workflow of applying a "formalising" transformation, gaining insights from a formal analysis, and reflecting these insights back to the initial models in a productive manner is a clear application of BX languages.

3. Domain Analysis

In this section, we identify syntactic elements required for the transformation from SysML to Event-B.

3.1. Running Example

As a running example, we introduce a small and simple case study provided by DB Netz AG. The case study originates from a technical specification and requirements document describing a *point machine* interface to an interlocking. In railway signalling, an interlocking is the part of a signal apparatus that prevents conflicting movements through an arrangement of tracks such as junctions or crossings.[4] An interlocking is designed so that it is impossible to display a signal to proceed unless the route to be used can be proven to be safe.

Figure 2 depicts the configuration of a point machine: Two tracks represent two possible positions, denoted as left and right. The lamps represent the position of the tracks after the movement. The main requirement is to move the tracks to the left or right position depending on the commands from the interlocking. Being a safety-critical system, properties such as: "When the track is set to right, the lamp should be lit" have to be proven to hold as soon as the command is given by the interlocking.

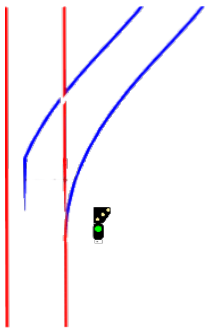


Figure 2 Point machine

EULYNX[5] develops the railway interlocking specification and requirements, involving various state machine diagrams for individual systems of the interlocking. These include level-crossing systems, interlocking systems, light signalling systems, and other auxiliary systems. Each of these systems interacts and communicates with each other through a communication

[4] https://projects.au.dk/into-cps/industry/railways-case-study/
[5] https://www.eulynx.eu/

interface, which must guarantee safe communication. To this end, the relevant behavioural part, i.e., the state machines of the SysML models must be verified against safety properties necessary to fulfil user requirements. To perform the verification, the state machines are transformed to Event-B code, so that Event-B can be used as a formal method. To implement this transformation, two questions must be clarified:

1. Which subsets of SysML and Event-B are sufficient for the required verification?

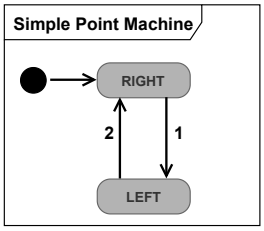
2. Which model elements are semantically interrelated, and how can they be consistently transformed?

Based on the list of state-machine diagram elements from the SysML 1.6 specification (Object Management Group 2019), we defined a supported subset of features sufficient to fulfill the requirements of the involved stakeholders at DB Netz AG, i.e., domain experts both in systems engineering and formal verification. In the following, we briefly introduce the two considered languages and identify respective subsets relevant for the transformation.

3.2. SysML: A Semi-Formal Language

SysML was developed from UML as a general purpose language for MBSE (Holt & Perry 2019). SysML can be regarded as both an extension and a restriction of UML including nine diagrams subdivided into structural (block definition diagram, internal block diagram, package diagram), behavioural (use case, activity, sequence, and state machine diagram), requirement and parametric diagrams. SysML is a primarily visual modelling language and aims to be easily understood by system engineers. It has gained popularity in different fields such as aerospace, defence, and medical industries (Holt & Perry 2019).

While five SysML diagrams are used in the context of the EULYNX initiative, we restrict ourselves to the transformation of *state machine diagrams* in the scope of this paper. In the following, we introduce the relevant concepts of SysML by modelling the point machine (cf. Fig. 2) as a state-machine diagram, depicted in Fig. 3.




```
1 when(move_point_left = TRUE) /
    left  := TRUE;
    right := FALSE;
2 when(move_point_right = TRUE) /
    left  := FALSE;
    right := TRUE;
```

Figure 3 SysML state machine for the point machine

3.2.1. State Machine and Region

There are two types of state machines: *Behavioural state machines* describe the behaviour of subsystems, whereas *protocol state machines* define valid interaction sequences, denoted as *protocols*. For defining the transformation to Event-B, the focus is set on behavioural state machines.[6] The point machine of Fig. 2 is depicted as a state machine in Fig. 3, identified by its name on the top left. A state machine can consist of multiple regions, while only one region is necessary for this example.

3.2.2. State

A state models a situation in the execution of a state machine, during which some invariant condition holds (Object Management Group 2015). States are also considered the fundamental building blocks of the state machine. In each active region, at most one state can be assumed at the same time. States can be subdivided into simple, composite, and submachine states; only simple states are relevant for the scope of this paper. Simple states do not have any sub-states, regions or internal transitions. Simple states can either be atomic or final states. While atomic states have no special meaning, final states, when active, indicate the completion of their parent region, i.e., the admissibility of a given input sequence. Figure 3 depicts two simple states (`LEFT` and `RIGHT`), which are both atomic.

3.2.3. Pseudo-State

The difference between a state and a pseudo-state is that the latter cannot be assumed by the state machine. Pseudo-states are typically used to connect multiple transitions into more complex paths. For example, a fork pseudo-state with a single incoming and multiple outgoing transitions can be regarded as a compound transition that leads to a set of orthogonal target states. Pseudo-states can be classified as initial states, junctions, choices, forks and joins, entry and exit points or history states. In the scope of this paper, we restrict ourselves to initial states, whereby an extension of the transformation towards other pseudo-state types is possible. In a region, there can be at most one initial state present. The initial state has only outgoing but no incoming transitions. The outgoing transitions of the initial state cannot trigger any event and have no guards. An initial state is depicted as a small solid filled circle (cf. Fig. 3).

3.2.4. Transition, Event, Effect, Trigger, and Guard

A *transition* in a state machine is a directed association between a source state and a target state, and can be expressed in the following form:

```
transition ::= [trigger][guard]['/'effect]
```

In Fig. 3, three transitions are depicted as arrows between the three states. An optional *trigger* can be used to specify an event that induces a state transition. An *event* is a notable occurrence at a point in time that causes a reaction of the state machine. These reactions lead to an execution step of the modelled behaviour. For example, a signal event may trigger a transition of a state machine. In Fig. 3, for instance, `move_point_left` triggers the transition from `RIGHT` to `LEFT`. An optional guard specifies additional constraints as a boolean expression. An optional *effect* is an action to be executed when its transition fires. While effects are sufficient for our current considerations, further action types such as *entry*, *exit*, and *do* actions can be added as an extension of the transformation. Considering the running

[6] To distinguish state machines from the eponymous diagram type of the UML, they are also referred to as SysML state machines.

example of Fig. 3, the variables `left` and `right` are set to `TRUE` and `FALSE` as an effect, respectively, when the transition from `RIGHT` to `LEFT` fires.

3.2.5. Port Ports are interfaces via which external entities can connect to and interact with the specified system. In the running example, the point machine receives the instruction whether to move to the right or to the left via ports (`move_point_left`, `move_point_right`). Ports are often used to trigger events.

3.3. Event-B: A Formal Language

Even though the behaviour of the point machine can be specified with a SysML state-machine as presented, it is not possible to verify safety-related properties. The chosen solution for formal verification in the context of the EULYNX initiative is to transform SysML state-machines into formal Event-B models. An overview of the relevant syntactic constructs of the Event-B language is therefore provided in the following.

Event-B is a formal method for system level modelling and analysis.[7] Its key features are the use of set theory as a modelling notation, defining a system at different levels of abstraction via refinements, and the verification of formal properties via theorems and invariants (Abrial & Hallerstede 2007). An Event-B model comprises two main components, a *Machine* and *Context*. For the transformation of SysML state machines, the former is sufficient.

3.3.1. Machine, Variables, Events In Event-B, a machine defines the behavioural properties of the model. Each machine is composed of variables (v), invariants ($I(v)$), and a collection of transitions denoted as events (cf. the schematic example[8] depicted in Fig. 4). There are further optional building blocks annotated with a $*$ in Fig. 4: A machine can refine another machine and be embedded into one or more contexts. Refined machines have an additional variant block containing an expression that is unique for the machine. Theorems $R(v)$ are additional properties that must be derivable from the set of invariants $I(v)$. For each theorem, a proof obligation is generated to prove that the theorem is derivable from the invariants, i.e., $I(v) \vdash R(v)$ (Hoang 2013). As the mandatory components are sufficient for describing the running example, we will now describe them in the remainder of this section.

The set of variables v defines the current state of a Event-B machine, and is used in several other language constructs. To model the point machine, six boolean variables are required (cf. Fig. 5): For the current position of the point machine (`RIGHT`, `LEFT`), for the next possible movement direction (`right`, `left`), and for the actual movement commands from outside the system (`move_point_right`, `move_point_left`).

3.3.2. Invariants Invariants $I(v)$ define constraints which have to hold at any time, i.e., in each possible state of the machine. They can involve one or more variables forming an expression in first order logic. Each invariant has a label,

```
MACHINE
  <machine_identifier>
REFINES *
  <machine_identifier>
SEES *
  <context_identifier>
  ...
VARIABLES
  <variable_identifier>
  ...
INVARIANTS
  <label> : <predicate>
  ...
THEOREMS *
  <label> : <predicate>
  ...
EVENTS
  <label>≙
  STATUS
    <status>
  WHEN
    <label> : <guard>
    ...
  THEN
    <label> : <action>
    ...
  END
  ...
VARIANT *
  <variant>
END
```

Figure 4 Event-B Machine Structure Example

followed by a colon and an expression. In Fig. 6, the invariants `typeof_RIGHT`, `typeof_LEFT`, `inv4`, `inv5`, `inv6` and `inv10` only define the data types of the respective variables; `inv3`, `inv7`, `inv8`, and `inv9` state that the involved variables must either be true or false, i.e., are never undefined. Finally, `distinct_states_in_position` ensures that the machine cannot be in state `RIGHT` and `LEFT` at the same time (position refers to the machine here). The predefined predicate *partition(S,x,y)* ensures that a set S can be partitioned into x and y, i.e. $x \cap y = \emptyset$ and $x \cup y = S$, so exactly one of the variables `RIGHT` and `LEFT` must be set to `TRUE`.

3.3.3. Events, Guards, Actions As a machine specifies the dynamic behaviour of an Event-B model, events are essential to trigger changes from one machine state to another. Events

```
MACHINE
  machine
VARIABLES
  RIGHT
  LEFT
  right
  left
  move_point_left
  move_point_right
  ...
END
```

Figure 5 Event-B: Machine with variables

[7] http://www.event-b.org/
[8] Adapted from http://deploy-eprints.ecs.soton.ac.uk/11/3/notation-1.5.pdf

```
INVARIANTS
  typeof_RIGHT : RIGHT ∈ BOOL
  typeof_LEFT : LEFT ∈ BOOL
  distinct_states_in_position : partition({TRUE},
    {RIGHT} ∩ {TRUE}, {LEFT} ∩ {TRUE})
  inv3 : (left=TRUE) ∨ (left=FALSE)
  inv4 : right ∈ BOOL
  inv5 : left ∈ BOOL
  inv6 : move_point_right ∈ BOOL
  inv7 : (right=TRUE) ∨ (right=FALSE)
  inv8 : (move_point_left=TRUE) ∨
    (move_point_left=FALSE)
  inv9 : (move_point_right=TRUE) ∨
    (move_point_right=FALSE)
  inv10 : move_point_left ∈ BOOL
```

Figure 6 Event-B: Invariants

involve a set of variables v and parameters x for these variables. An event e occurs in some state, if there exists some value for its parameter x such that the *guard* $G(x,v)$ holds in that state. In Fig. 7, there are two events that describe the transition from the state `RIGHT` to `LEFT` (`next_position`) and in the opposite direction (`current_position`). To move the point machine from `RIGHT` to `LEFT`, the machine must be in the state `RIGHT` (`isin_RIGHT`) and there must be a command to move it to the other state (`position_guards1`). These two conditions represent the guard of the event. For the `current_position` event, the opposite must hold. Actions describe how state variable values change when an event occurs. Similar to a guard, an action $Q(x,v)$ involves parameters x and variables v. As soon as the event occurs, the specified values are assigned to the respective variables. Each assignment is identified by a label (cf. Fig. 7). In case of the `next_position` event, the variables `RIGHT` and `LEFT` flip their values, as well as the variables for the next movement (`right` and `left`). As depicted in Fig. 7, a guard is a block following the keyword `WHEN`, the action is surrounded by the keywords `THEN` and `END`.

There are also events that do not have a guard, i.e., only consist of an action block. In our example, the initialisation event brings the point machine into the state `RIGHT` at the beginning, depicted in Fig. 8. Syntactically, the `WHEN` block is omitted, and the keyword `THEN` is replaced by `BEGIN`.

```
EVENTS
  ...
  next_position ≙
  STATUS
    ordinary
  WHEN
    isin_RIGHT : RIGHT = TRUE
    position_guards1 : move_point_left = TRUE
  THEN
    leave_RIGHT : RIGHT := FALSE
    position_actions1 : left := TRUE
    position_actions4 : right:= FALSE
    enter_LEFT : LEFT := TRUE
  END

  current_position ≙
  STATUS
    ordinary
  WHEN
    isin_LEFT : LEFT = TRUE
    position_guards2 : move_point_right = TRUE
  THEN
    leave_LEFT : LEFT := FALSE
    position_actions2 : right := TRUE
    position_actions3 : left := FALSE
    enter_RIGHT : RIGHT := TRUE
  END
```

Figure 7 Event-B: Move Events

```
EVENTS
  INITIALISATION ≙
  STATUS
    ordinary
  BEGIN
    init_RIGHT : RIGHT := TRUE
    init_LEFT : LEFT := FALSE
    act1 : move_point_left := TRUE
    act2 : right := FALSE
    act3 : left := FALSE
    act4 : move_point_right := TRUE
  END
  ...
```

Figure 8 Event-B: Initialisation Event

4. Transformation Design

According to the TGG approach, a transformation designer must specify a *consistency relation* between the involved languages (here the relevant subsets of SysML and Event-B). A TGG tool is then able to automatically derive various *operations* from the single consistency relation, including a forward transformation and backward synchronisation as required for our application. The TGG approach guarantees that all derived operations are *correct* in the sense that they always result in consistent triples according to the provided consistency relation.

After identifying relevant subsets of the involved modelling languages (cf. Sect. 3), the next step in the TGG transformation design process is to identify semantic similarities, i.e., the semantic overlap of both languages (cf. Sect. 4.1). This is formalised as a triple metamodel with correspondence link types connecting semantically overlapping types. Finally, based on the triple metamodel, a *rule-based* definition of the consistency relation completes the specification.

4.1. Semantic Similarities and Triple Metamodel

Our aim is to identify similarities in the semantics of the two languages such that SysML models can be transformed into Event-B models in a semantics preserving manner. Doing this requires domain knowledge from experts familiar with both languages. Figure 9 depicts the relevant excerpt of the SysML metamodel for state machines to the left, the Event-B metamodel to the right, and the mapping in form of correspondences (denoted as diamonds), which are connected to the respective metamodel elements. To improve readability, only multiplicities different from `1` for the source and `0..*` for the target of an association are depicted.

The `Statemachine` class defines the primary behaviour of

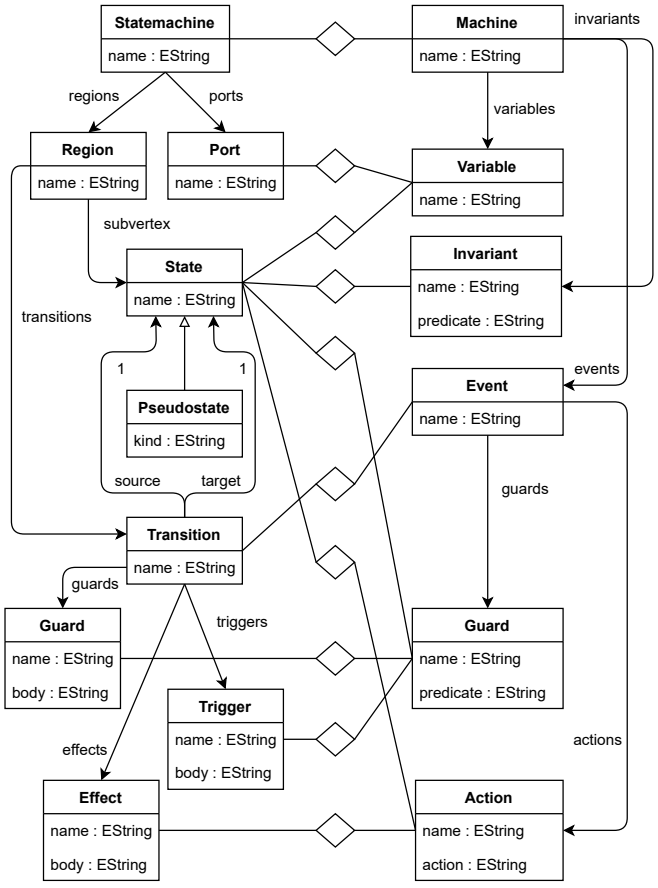


Figure 9 Triple Metamodel

the modelled system and consists of `Regions` and `Ports`. The Event-B `Machine`, which consists of `Variables`, `Invariants` and `Events`, has the same purpose, so these classes correspond to each other. In the SysML metamodel, a `Region` consists of `States` and `Transitions` but has no direct correspondence in the Event-B metamodel. The `States` of a SysML state machine correspond to `Variables` in the Event-B model. As they can also be involved in the definition of `Invariants`, `Guards`, and `Actions`, correspondence links to these language constructs are also required. `Invariants` in Event-B specify the properties that a `Variable` must satisfy before and after each `Event`. For example, the data type and value ranges of variables that correspond to SysML `States` are specified by `Invariants`. A `Transition` in SysML is annotated with `Triggers`, `Guards` and `Effects`, and it represents the directed relation between a source and a target `State`. Activating a `Transition` is similar to the occurrence of an `Event` in Event-B, thus these elements are connected via a correspondence. An `Event` incorporates `Guards` to restrict its occurrences and `Actions` that take place during the `Event`. `Guards` and `Actions` in Event-B thus correspond to `Triggers`, `Guards`, and `Effects` in SysML as depicted in Fig. 9. `Ports` in SysML state machines are used for communication with external components. They are used to trigger `Events` and are therefore represented as `Variables` in the Event-B model. In Sect. 3.2, we made a distinction between `States` and `Pseudostates`, with the latter being a subclass of the former in the SysML metamodel. `Pseudostates` do not have a direct correspondence in Event-B and do not need to be translated. Their adjacent `Transitions`, however, correspond to `Events` (cf. Fig. 8), such that the semantics of `Pseudostates` has an indirect influence on the Event-B model. Such details cannot be expressed solely using the triple metamodel, however, as it is mainly used for typing model elements and defining mappings between nodes of particular types. Instead, TGG rules are used to fully specify the desired consistency relation between the languages, which is presented next.

4.2. A Consistency Relation as a Triple Graph Grammar

Consistency management is a key challenge in the field of Model-Driven Engineering (MDE), as domain experts of different fields of expertise work on different models concurrently. TGGs are a well-known approach to consistency management with the unique advantage of being declarative enough to address multiple consistency management operations with the same specification, while still achieving an acceptable level of scalability for realistic application scenarios (Anjorin, Buchmann, et al. 2020). TGGs were introduced by Schürr (Schürr 1994) as a technique for bidirectional model transformation. As a BX language, TGGs can be viewed as a practical implementation of the delta-lens framework, based on the mature theory of algebraic graph transformation (Anjorin 2016). From a single declarative specification, rules for different operations including forward and backward transformation, consistency checks, and model synchronisation, can be automatically derived. In the application scenario under consideration, a *forward transformation* (initial transformation from SysML to Event-B) and a *backward synchronisation* (propagating updates/fixes from Event-B back to SysML) are of special interest. In addition to potentially[9] reducing execution times, an incremental synchronisation of updates is an elegant means of retaining existing and unchanged structure that does not have a corresponding construct in the updated model (e.g. `Regions` in SysML).

A TGG defines a consistency relation between two languages by generating a language of admissible, i.e. *consistent*, triple graphs. The triple graphs consist of connected source, target, and correspondence models, typed over a triple metamodel (cf. Fig. 9), also denoted as a *TGG schema*. All model triples that can be generated by a finite sequence of *TGG rules* starting from an empty triple graph, form the language of the respective TGG. A given model triple is *consistent* with respect to a TGG if it is a member of the language of the TGG.

To specify a suitable consistency relation of SysML state machines and Event-B models, a TGG was defined for the application scenario in close cooperation with domain and transformation experts at DB Netz AG. 14 rules were defined in total, from which we now present a selected rule to demonstrate how the required transformation and synchronisation operations are automatically derived. The complete set of TGG rules in their visual representation is available online[10].

In Fig. 10, the rule `DefaultTargetStateToEnterAction`

[9] Some BX languages require longer execution times for incremental modi (Anjorin, Buchmann, et al. 2020)

[10] https://bit.ly/3aFuKSH

is depicted. It allocates an existing `State` to a `Transition` in SysML, and creates a corresponding `Action` in the Event-B model. The new `Action` is connected to the SysML `State` via a correspondence link. To be able to apply this rule, several elements must already exist: Besides the `State`, there must be a `Transition` in the same `Region` of the SysML `Statemachine`. This `Transition` must correspond to an `Event` of an Event-B `Machine` that in turn corresponds to the correct SysML `Statemachine`; the new `Action` is added to the `Event`. The elements required to exist before applying the rule, also referred to as *context elements*, are coloured black, whereas elements created by applying the rule are green and have a ++ mark-up.

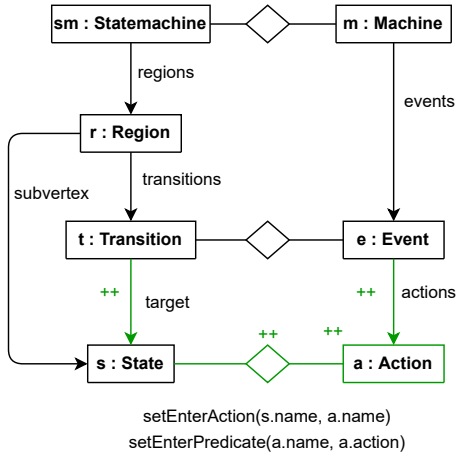


Figure 10 Declarative TGG rule

Finally, there are two *attribute conditions* attached to the rule, represented in a textual concrete syntax at the bottom of Fig. 10. These attribute conditions must evaluate to `true` after rule application, i.e., can express conditions between attributes of existing elements as an additional precondition for rule application, or include attributes of newly created elements representing a postcondition over attribute values. While there is a library of standard attribute conditions, it is also possible to implement new conditions (Anjorin et al. 2012) such as `setEnterAction` and `setEnterPredicate` for a specific application scenario.

To use TGG rules for model transformations, they have to be *operationalised* for the specific transformation scenario. In Fig. 11, the rule shown in Fig. 10 is operationalised for a forward transformation, i.e. in this case the transformation from SysML to Event-B. As the source model is provided as input in this scenario, all context elements of the declarative rule must be already *translated*, which is indicated by a respective marker (☑). As the "created" edge from `t` to `s` already exists, it is marked as translated by applying this operational rule (☐ → ☑). Compared to the declarative rule (Fig. 10), the target and correspondence model elements in the operational rule are unchanged as they are constructed analogously during the forward transformation process. In this manner, all declarative rules are systematically operationalised to derive corresponding forward rules. The forward rules are then used by an algorithm to realise a forward transformation. Conceptually, all that has to be done is to search for an applicable forward rule, apply it, and continue the process until no forward rule is applicable, i.e., all input elements are marked as translated. To ensure that this process always results in a consistent triple (correctness), does not fail if such a consistent result exists (completeness), and is reasonably efficient years of research have been invested in developing translation algorithms for TGGs. The interested reader is referred to e.g., Schürr et al. (Schürr & Klar 2008) for further details.

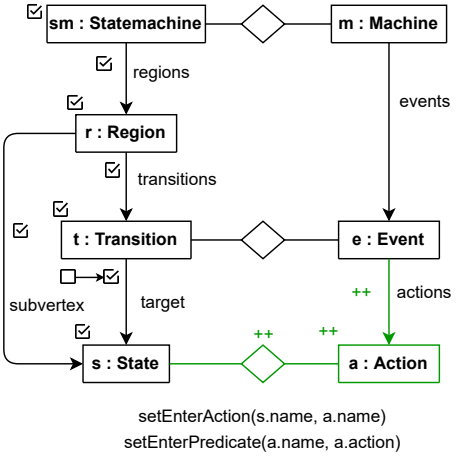


Figure 11 Forward Transformation Rule

We now provide some more details concerning the two involved attribute conditions, depicted in Fig. 12 and 13 in an abbreviated form. Attribute conditions specify the consistency relation between attributes in different models (cf. (Anjorin et al. 2012; Lambers et al. 2012; Guerra et al. 2009)). The attribute condition `setEnterAction` relates the values of the SysML `State` name and the Event-B `Action` label. These values are stored in the variables `v0` and `v1`, respectively. The *binding state* of the involved attributes can be obtained (Line 5), and is a sequence of `B` (for bound) or `F` (for free) for every attribute. If the `State` name is bound and the `Action` label free, i.e., case `BF`, the label is set by adding a prefix (Line 9), satisfying the condition (Line 10). If both attributes are bound, i.e., case `BB`, the condition is satisfied if their values correspond as expected (Line 14-15). All other cases can be handled analogously.

Two important points should be noted: (i) Such an attribute condition is implemented once and can be combined freely with other attribute conditions in any order in multiple TGG rules. (ii) As part of the operationalisation process, the correct order of conditions and choice of binding states is computed. This ensures that a compact library of attribute conditions can be implemented and reused flexibly. For example, `setEnterAction` could be implemented as `addPrefix("enter_", s.name, a.name)` with a generic library attribute condition `addPrefix` (Anjorin et al. 2012).

The second attribute condition `setEnterPredicate` (Fig. 13) operates on the label (`name`) and the actual instruction (`action`) of an `Action`. This means that it is used to ensure *intra-model* consistency as it only refers to the target model. Let us consider the case of `BB`, i.e., both attributes are bound, which

```
1  public class setEnterAction extends AttrCond {
2    public void solve() {
3      var v0 = variables.get(0);
4      var v1 = variables.get(1);
5      var bindingState = getBindingState(v0, v1);
6
7      switch(bindingState){
8        case "BF": {
9          v1.bindTo("enter_" + v0.getValue());
10         setSatisfied(true);
11         return;
12       }
13       case "BB": {
14         setSatisfied(v1.getValue().equals(
15           "enter_" + v0.getValue()));
16         return;
17       }
18       case "FB": {...}
19       case "FF": {...}
20     }
21   }
22 }
```

Figure 12 Attribute Condition `setEnterAction`

is especially relevant for backward transformation and synchronisation (the Event-B model is provided as input). On Line 10, the condition checks if the label starts with `"enter_"` as for example in `"enter_LEFT : LEFT := TRUE"` from Fig. 7. If this is the case, then the expected instruction should be, e.g., `"LEFT := TRUE"` for label `"enter_LEFT"`. This check is implemented on Line 12 – 14. For all other cases, the condition is not satisfied and therefore blocks the application of the corresponding rule.

```
1  public class setEnterPredicate extends AttrCond {
2    public void solve() {
3      var v0 = variables.get(0);
4      var v1 = variables.get(1);
5      var bindingState = getBindingState(v0, v1);
6
7      switch(bindingSate){
8        case "BB": {
9          var name = v0.getValue();
10         if(name.startsWith("enter_")){
11           var expectedAction =
12             name.replace("enter_", "") + " := TRUE";
13           setSatisfied(v1.getValue().equals(
14             expectedAction);
15         } else {
16           setSatisfied(false);
17         }
18         return;
19       }
20       case "FF" {...}
21       ...
22     }
23   }
24 }
```

Figure 13 Attribute Condition `setEnterPredicate`

The presented triple metamodel and rules only cover the basic language constructs, and can be extended to handle, e.g., composed states and include them in the scope of the transformation, which is left to future work due to space limitations.

5. Implementation

This section introduces the tool-chain used to implement the transformation, and sketches the work-flow established for the existing tools at DB Netz AG.

5.1. PTC Integrity Modeler

For creating SysML models, the PTC Integrity Modeler[11][12] is used at DB Netz AG. The PTC Integrity Modeler provides a development environment that allows different systems engineering teams to work in a collaborative setting, from the conceptual level to the delivery and maintenance of the system. It helps define an unambiguous single model definition of the system, including requirements, functions, as well as hardware and software components. Besides SysML, several other OMG standards are supported, including UML, Unified Profile for DoDAF/MODAF (UPDM), and Open Verification Methodology (OVM). For creating SysML state machines, a visual editor is used. It is also possible to simulate modelled behaviour to detect errors in early stages of development. The tool provides interfaces for synchronisation with other modelling tools (e.g. Simulink[13], Doors[14]). Code in different general purpose languages (e.g. C, C++, Java and Ada) can be generated from the models (Zolotas et al. 2020). For our tool integration solution, the model export to Eclipse Modelling Framework (EMF)-compatible XML Metadata Interchange (XMI) is relevant.

5.2. RODIN Platform

As an Integrated Development Environment (IDE) for formal modelling with Event-B, the *RODIN* tool (Abrial et al. 2010) is used at DB Netz AG. It is provided as an open-source Eclipse plug-in that supports the construction and verification of Event-B models. Besides basic support for formal modelling, RODIN provides feedback for the developer at design-time. Event-B development (modelling and programming), and formal verification are decoupled into distinct phases to ease, for example, tracing the origin of a failed proof obligation. As verification techniques, both model checking and theorem proving are supported. Model checking can be used as a pre-filter, before theorem provers are applied to proof obligations (Abrial et al. 2010). In addition to their textual representation, formal models can be visualised and simulated to make the models more comprehensible for the developer. These features are integrated via a range of plug-ins for the RODIN platform. Various analysis tools, such as theorem provers[15], model checkers[16], step-wise simulation[17] and translation tools such as for UML-B[18] have been developed as extensions for the RODIN platform. The UML-B plug-in, for instance, helps to diagrammatically visualise the formal model, and thereby aids construction and

[11] https://www.mathworks.com/products/connections/product_detail/ptc-integrity-modeler.html
[12] https://www.ptc.com/en/products/windchill/modeler
[13] https://www.mathworks.com/products/simulink.html
[14] http://www-03.ibm.com/software/products/en/ratidoor
[15] http://www.b4free.com/index.html
[16] http://www.stups.uni-duesseldorf.de/ProB/overview.php
[17] http://www.brama.fr/indexen.html
[18] users.ecs.soton.ac.uk/cfs/umlb.html

validation. Event-B machines are stored and imported as XMB files, which are syntactically similar to XMI files. XMB files can be visualised with the Rose Structured Editor from different viewpoints.

5.3. eMoflon::IBeX

To bridge the modelling environments for SysML and Event-B (cf. Fig. 14), an additional tool is required to perform the transformation and synchronisation steps in both directions. In our approach, the TGG-based model management tool eMoflon::IBeX[19] is used to address this task. Although only forward transformation and backward synchronisation are of primary interest for our application, eMoflon::IBeX also supports other consistency management operations including consistency checking and (concurrent) model synchronisation (Weidmann et al. 2019). Similar to RODIN, eMoflon::IBeX is an open-source Eclipse plug-in. Several external components can be attached via defined interfaces, including incremental graph pattern matchers and Integer Linear Programming (ILP) solvers, leading to a modular software architecture. Both triple metamodels and TGG rules are specified in a textual concrete syntax, complemented with a simultaneous read-only visualisation using PlantUML[20]. Additional attribute conditions from an extensible library of conditions implemented in Java (cf. Sect. 4.2) can be specified in a simple textual concrete syntax (Weidmann et al. 2019). Metamodels for source, target, and correspondence models as well as the respective models are all EMF compatible and can be persisted in any EMF compatible format including the default XMI. The common use of EMF as a modelling standard substantially eases the establishment of a tool-chain to connect PTC Integrity Modeler, eMoflon::IBeX, and RODIN.

5.4. From PTC Integrity Modeler to RODIN and Back

The setup for the established tool integration is depicted in Fig. 14. eMoflon::IBeX is used as a bridge between the PTC Integrity Modeler and RODIN. After specifying the consistency relation between SysML state machines and Event-B as triple metamodel and a TGG, eMoflon::IBeX is used to operationalise the TGG as required for the scenario.

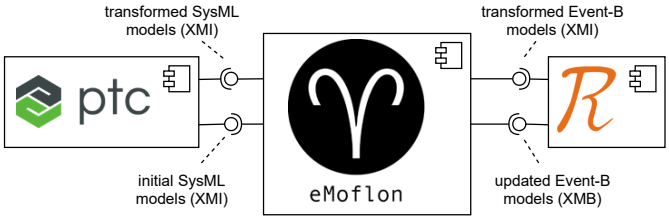


Figure 14 Overview of the Tool Integration Setup

The behaviour modelling of the system is done using SysML with the PTC Integrity Modeler. The resulting model is exported as an XMI file from which a relevant part (state machines) is extracted and passed on as the input model for eMoflon::IBeX. The forward transformation is executed by eMoflon::IBeX, using the derived forward rules and a generic forward transformation algorithm. The output of this forward transformation is a correspondence and target model that extend the source model to a consistent triple. A transformation protocol is also generated containing information about which rules were executed in which order to create which elements. While only the target model is strictly required for the tool integration, the correspondence model and transformation protocol are useful for debugging and understanding the transformation, e.g. with MDE debuggers for TGGs (Giese et al. 2014; Weidmann et al. 2020). The next step is to convert[21] the generated target model from XMI to XMB and to import it into RODIN. After this RODIN is used to automatically generate Event-B code from the XMB file. The Event-B code can now be used to perform formal verification and check if all safety requirements are fulfilled. Gained insights are integrated directly in the Event-B code, resulting in a new version. To reflect these changes to the Event-B code back to SysML, the backward synchronisation with eMoflon::IBeX requires (i) the old triple of source, correspondence, and target models, and (ii) a *target delta* representing the changes applied to the target model, which should be backward propagated incrementally to the existing source model. Currently, this target delta (also a model) must be created manually based on a text diff between the initial and final Event-B code. This is certainly a step that could be improved in the future by automatically transforming a diff on Event-B code to a target delta that eMoflon::IBeX directly understands.

The current state of the implementation should be regarded as a proof-of-concept prototype as (i) the level of automation can still be improved, and (ii) only the most important parts of the two behavioural models are covered. We are convinced, however, that the scope can be extended to cover the remaining parts of the state machine specification and even further SysML models without fundamentally changing the overall work-flow. Using XMI as a uniform data exchange format seems promising as it is supported by all three tools, but has a number of drawbacks including its missing ability to represent diffs or delta structures (Zolotas et al. 2020). Further tool integration could replace XMI by a more suitable standard for data exchange. To assess strengths, weaknesses, and the potential of our approach, the results of a qualitative evaluation are presented in Sect. 6.

6. Evaluation

We now provide a qualitative evaluation of our implemented solution based on three small but representative test cases provided by DB Netz AG. After running the transformation and synchronisation chain for the three test cases, we then conducted a semi-structured interview with three SysML and Event-B modelling experts at DB Netz AG. In particular, we aimed to investigate the following research questions:

RQ1 Feasibility: Is it possible to transform representative examples of SysML state machines into Event-B (Sect. 6.1)?

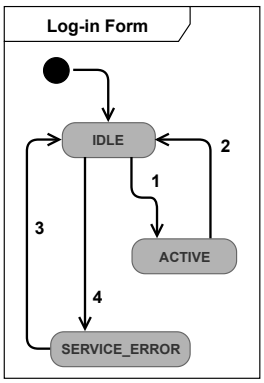
[19] emoflon.org
[20] plantuml.com/
[21] In most cases this just involves changing the extension of the file from ".xmi" to ".xmb".

RQ2 How is the applicability, extensibility and usability of the solution perceived by relevant practitioners (Sect. 6.2)?

6.1. Representative Test Cases

This section briefly introduces the three test cases used for our evaluation. To validated the transformation results, formal modelling experts at DB Netz compared them to the expected Event-B models and conducted simulations using the RODIN platform. Although the test cases are rather simple and contain only language elements that were presented in Sect. 3.2, they differ in the structure of transitions and states, and combine different triggers and actions with each other. In particular, they can be used to investigate whether the rule-based transformation produces a correct result that complies with the modelling experts' expectations.

6.1.1. Log-In Form State Machine The first test case consists of a *state machine* designed for a simple log-in process. It consist of three states: In the `IDLE` state, the log-in form is ready to accept requests, while the `ACTIVE` state indicates that a user has logged into the system. The `SERVICE_ERROR` state represents an error state for the system. It is possible to switch between `IDLE` and `ACTIVE`, as well as between `IDLE` and `SERVICE_ERROR`, whereas a direct transition from `ACTIVE` to `SERVICE_ERROR` is not possible. The transition 1 has a trigger (in round brackets) and a guard (in square brackets), whereas all other transitions have only one trigger. Figure 15 depicts the state machine for the log-in form.

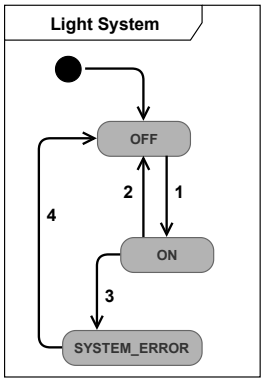



```
1 when(Insert_Card = TRUE)
    [Finish = FALSE] /
    Card_Inserted := TRUE;
    Service_Fixed := FALSE;
    Finish := FALSE;
2 when(Insert_Card = FALSE) /
    Card_Inserted := FALSE;
    Finish := TRUE;
3 when(Check_Service = FALSE) /
    Service_Error := FALSE;
    Service_Fixed := TRUE;
4 when(Check_Service = TRUE) /
    Finish := FALSE;
    Service_Error := TRUE;
    Service_Fixed := FALSE;
```

Figure 15 Test Case 1: Log-In Form State Machine

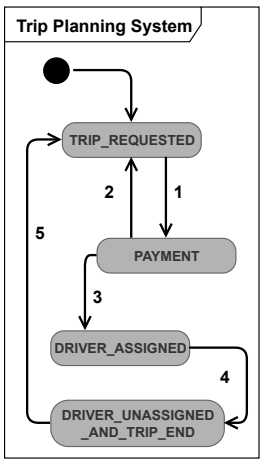
6.1.2. Light System This state machine represents a basic `Light System`. Similar to the log-in form test case, there are three states, of which one is an error state. In this example, however, there is a transition from the state `ON` to the state `SYSTEM_ERROR`, forming a cycle between all three states. Figure 16 depicts the state machine for light system.

6.1.3. Trip Planner System This test case is slightly more complex than the previous two as there is a fourth state and a fifth transition to be transformed. The system models a trip planner, e.g. for a cab ride. As soon as a user requests a trip, they are asked to pay a certain amount of money (1). The user can either confirm the payment and choose a driver (3) or go back to modify the requested trip (2). Transition 4 represents the completion of the trip. With the last step (5), the driver is unassigned and the state machine goes back to the state `TRIP_REQUESTED`. Figure 17 shows the state machine for the trip planning system.




```
1 when(switch_on = TRUE) /
    light_on := TRUE;
    light_off := FALSE;
    shut_system := FALSE;
2 when(switch_on = FALSE) /
    light_on := FALSE;
    light_off := TRUE;
3 when(switch_error = TRUE) /
    light_on := FALSE;
    light_error := TRUE;
    shut_system := FALSE;
4 when(switch_error = FALSE) /
    light_error := FALSE;
    shut_system := TRUE;
```

Figure 16 Test Case 2: Light System State Machine




```
1 when(payment = TRUE) /
    payment_requested := TRUE;
    payment_failed := FALSE;
2 when(payment = FALSE) /
    payment_requested := FALSE;
    payment_failed := TRUE;
3 when(payment_success = TRUE) /
    payment_successful := TRUE;
4 when(driver_assigned = TRUE) /
    driver_cancelled := FALSE;
    trip_completed := TRUE;
    payment_successful := FALSE;
5 when(driver_assigned = FALSE) /
    driver_cancelled := TRUE;
    trip_completed := FALSE;
```

Figure 17 Test Case 3: Trip Planning System State Machine

Using our implemented solution, Event-B machines could be generated from the SysML input models. The generated Event-B machines were all syntactically correct and could be displayed in the RODIN platform. According to domain experts, the transformation works as expected for the three test cases.

6.2. Interviews with Modelling Experts and a Project Manager at DB Netz AG

To investigate RQ2, we conducted three semi-structured interviews presented in the following, which were based on the implemented test cases. We were able to obtain interviews with (i) a modelling expert from the SysML semi-formal modelling side, and (ii) a modelling expert from the Event-B formal modelling side in order to get technical feedback, as well as with (iii) a project manager in order to get feedback on the complete approach and aspects related to strategic plans for future developments. The interviews presented in this section are a summary of the complete interviews provided online[22].

[22] https://bit.ly/3aFuKSH

6.2.1. Applicability The modelling experts expect the rule-based transformation to save time and lower the error rate compared to the current manual process. Even for project members with only basic knowledge about formal modelling, it should now be possible to generate and verify the formal model produced from a SysML state machine thanks to the fully automated procedure. Furthermore, based on expert reviews of the TGG, a certification of the entire process is now a possibility.

In order to use the approach in practice, the considered subsets of the SysML and Event-B metamodels must be sufficiently large. For a first proof-of-concept version, the currently supported subsets are sufficient but must be extended in the future. This extension depends on requirements, however, as it is important to only cover features that are actually useful for the formal verification process.

An advantage of using a BX language for the transformation is that several other consistency management operations can be derived from the same specification, i.e., from the developed TGG. While forward transformation and backward synchronisation are currently most relevant to reflect fixes in the formal model back to the semi-formal model, also forward synchronisation could be relevant for reflecting changes in the SysML model at later stages of the engineering process. Other supported operations, such as consistency checking, are certainly perceived as being potentially useful for future workflows in the context of the EULYNX project.

Considering the goals of the EULYNX project, the automated approach supports the synergetic combination of formal and semi-formal methods. A primary goal in EULYNX was to establish a well-understandable semi-formal language such as SysML to be used by all project partners, as well as to offer separate, complementary formal verification facilities. With the automated approach, this goal can be achieved to enable a uniform validation and verification process that incorporates safety requirements.

6.2.2. Extensibility As the EULYNX project is constantly evolving, it is important to have a solution that can easily be adapted and extended. For example, a recent change was the replacement of flow ports by proxy ports, so the supported SysML language subset would have to be extended accordingly. As the EULYNX specification also involves interconnected state machines, the current subsets also have to be extended to handle such state machines as well.

According to the interviewees, the automated approach can be easily extended and adapted to the expectations of the formal modelling experts for verification. The current transformation results provide a solid basis for the further development of the tool-chain by providing reliable results. The only thing lacking at the moment is “conformity”, which means that the translated models are not yet certified. If the complete TGG-based tool chain could be certified, then the overall validation and verification approach could be provided to other project members using different target formal modelling languages.

Regarding the future role of the automated transformation in research projects, a medium-term goal is to convince the project managers of EULYNX and RCA[23] of the importance of an automated transformation approach. A short-term goal is to apply the approach to EULYNX models, which can form the concrete basis to convince other project members.

6.2.3. Usability As the complete transformation process still requires some user interaction, usability aspects of the tool-chain cannot be ignored. The most important manual steps are including *safety requirements* and *safety invariants*. An additional manual step is to add further invariants of two types: *State invariants* have to hold for a single state, whereas *global invariants* apply to the complete model. While state invariants can already be generated automatically, global invariants must be specified and added manually by the formal modelling experts. This is because global invariants do not have a corresponding SysML construct and can only be added to the semi-formal model as an informal annotation.

Despite these manual tasks, the use of an automated transformation still leads to a substantial reduction of effort. For complex models such as several state machines with refinements communicating with each other, the automation promises to be especially beneficial in this regard. Moreover, assuring the correctness of the resulting Event-B machine for the manual process is still an unresolved issue, as errors can occur when humans perform the transformation manually. With the automated process, only safety invariants have to be added, and the rest of the Event-B code can be generated automatically, increasing time efficiency possibly by around 70-80%.

The main obstacle with the automated approach is that an average engineer might have reservations about processes they do not understand in full detail. To increase acceptance, it would be very helpful to have a User Interface (UI) that supports the engineer while conducting the automated transformation. The UI should support the user to initiate the transformation process via a single click of a button, and it should present the translated model as well as the corresponding verification results. It should also provide a means of visualising and editing the formal model if necessary, and should ideally be easy to use for inexperienced users without any expert knowledge on formal modelling. The UI should provide the domain experts in both semi-formal modelling and formal modelling with a simple work-flow to run the simulation and verify user requirements without having to fully understand the underlying details.

Another aspect relevant for future maintenance of the system is the required knowledge for refining the consistency relation definition, i.e., for adding or modifying TGG rules. Different levels of expertise are conceivable: To maintain the set of rules and modify the tool-chain, knowledge about both the semi-formal and formal language are required, and probably also expert knowledge in MDE. All other modelling experts should understand the overall process, but should not require in-depth knowledge to apply it. To achieve this, the solution should be well-documented based on the triple metamodel as a central and formal artefact.

[23] The RCA initiative is driven by several EULYNX project members and strives for improving command, control and signalling systems using MBSE techniques: https://eulynx.eu/index.php/news/61-rca-gamma-published

6.3. Summary and Threats to Validity

For all three test cases presented in Sect. 6.1, the transformation yields correct results according to the modelling experts. As the SysML state machines involve all syntactic constructs presented in Sect. 3.2 in different arrangements, the rule-based approach appears to work as expected for this language subset (RQ1). From the interviews, we conclude that the implementation forms a solid basis for automating the transformation from SysML to Event-B, which can be extended in the future. An extension to support the full expressive power of the formal and semi-formal modelling languages is, however, necessary for practical usage. Furthermore, the handling of the tool-chain still requires advanced knowledge about all three tools and underlying concepts. Usage should, therefore, be simplified by offering a suitable UI to support inexperienced users (RQ2).

While the provided test cases incorporate more states, transitions, and actions than the running example (cf. Sect. 3.1), the models are still rather small and simple. Larger, interconnected state machines would be necessary to determine corner cases for which further rules might be necessary. Regarding the assessment of the approach, it is important to note that the interviewees were already strongly in favour of automating the transformation, such that chances might be overstated and risks underestimated. Finally, only three people were interviewed, all employed at the same company and working on the same project at the time of implementation. It is questionable, therefore, if our results can be directly transferred to other industries or application contexts. Regarding maintainability, it is important to note that at least one person with advanced knowledge about TGGs must be involved in the project to add and adapt rules whenever this is necessary. For certifying the process, i.e., validating the correctness of the transformation itself, each new rule must be considered. As the TGG-based consistency relation definition is purely syntactic, the certification process should also involve simulation and different testing strategies to complement the formal verification.

7. Conclusion and Future Work

This paper proposes an approach for an automated model transformation between SysML and Event-B models based on TGGs. The approach is demonstrated with a prototypical implementation that is able to connect the modelling tools PTC Integrity Modeler for SysML, and the RODIN platform for Event-B via the TGG-based consistency management tool eMoflon::IBeX. We limited the scope of the transformation to minimal subsets of both languages covering most of the relevant language constructs, and identified semantic similarities to create a triple metamodel as a basis for typing the consistency relation. As a final step, we defined a suitable TGG consisting of 14 rules and required attribute conditions. To validate our approach, we provided a qualitative evaluation based on three representative examples provided by DB Netz AG and semi-structured interviews conducted with domain experts. Our interviewees stated that the prototype is indeed a good basis for substantially reducing manual efforts and for eventually certifying the transformation process.

Future steps include extending the transformation of SysML state machines to cover more language constructs, thereby increasing the practical applicability of the approach. This extension involves further tests with real-world models from the EULYNX project. Depending on the results, other project partners can be convinced to use BX techniques in a similar fashion to establish model transformations between their respective modelling languages. Furthermore, developing a suitable UI to support inexperienced users would improve the usability of the current approach.

Acknowledgments

This work was partially supported by the North Rhine Westphalian Ministry of Economic Affairs, Innovation, Digitalisation and Energy (MWIDE) through the Pro-LowCode project (005-2011-0022). We would also like to thank our anonymous reviewers for their helpful comments and suggestions.

References

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., & Stevens, P. (2016). Introduction to bidirectional transformations. In J. Gibbons & P. Stevens (Eds.), *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures* (Vol. 9715, pp. 1–28). Springer.

Abrial, J., Butler, M. J., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, *12*(6), 447–466.

Abrial, J., & Hallerstede, S. (2007). Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Informaticae*, *77*(1-2), 1–28.

Amendola, A., Becchi, A., Cavada, R., Cimatti, A., Griggio, A., Scaglione, G., ... Tessi, M. (2020). A model-based approach to the design, verification and deployment of railway interlocking system. In T. Margaria & B. Steffen (Eds.), *ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III* (Vol. 12478, pp. 240–254). Springer.

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2010). On challenges of model transformation from UML to alloy. *Softw. Syst. Model.*, *9*(1), 69–86.

Ando, T., Yatsu, H., Kong, W., Hisazumi, K., & Fukuda, A. (2014). Translation rules of sysml state machine diagrams into csp# toward formal model checking. *Int. J. Web Inf. Syst.*, *10*(2), 151–169.

Anjorin, A. (2016). An introduction to triple graph grammars as an implementation of the delta-lens framework. In J. Gibbons & P. Stevens (Eds.), *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures* (Vol. 9715, pp. 29–72). Springer.

Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H., Eramo, R., ... Zündorf, A. (2020). Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Softw. Syst. Model.*, *19*(3), 647–691.

Anjorin, A., Varró, G., & Schürr, A. (2012). Complex attribute manipulation in TGGs with constraint-based programming techniques. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, *49*.

Anjorin, A., Weidmann, N., Oppermann, R., Fritsche, L., & Schürr, A. (2020). Automating test schedule generation with domain-specific languages: a configurable, model-driven approach. In E. Syriani, H. A. Sahraoui, J. de Lara, & S. Abrahão (Eds.), *MoDELS 2020, Virtual Event, Canada, 18-23 October, 2020, Proceedings* (pp. 320–331). ACM.

Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., & Diguet, J. (2014). Synchronization of models of rich languages with triple graph grammars: An experience report. In D. D. Ruscio & D. Varró (Eds.), *ICMT@STAF 2014, York, UK, July 21-22, 2014. Proceedings* (Vol. 8568, pp. 106–121). Springer.

Butler, M., & Hallerstede, S. (2007). The rodin formal modelling tool. In *Facs 2007 christmas workshop: Formal methods in industry, proceedings* (pp. 1–5).

Caltais, G., Leitner-Fischer, F., Leue, S., & Weiser, J. (2016). Sysml to nusmv model transformation via object-orientation. In C. Berger, M. R. Mousavi, & R. Wisniewski (Eds.), *CyPhy 2016, Pittsburgh, PA, USA, October 6, 2016, Revised Selected Papers* (Vol. 10107, pp. 31–45). Springer.

Caltais, G., Leue, S., & Singh, H. (2020). Correctness of an ATL model transformation from sysml state machine diagrams to promela. In S. Hammoudi, L. F. Pires, & B. Selic (Eds.), *MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, Proceedings* (pp. 360–372). SCITEPRESS.

Freund, E. (2012). IEEE standard for system and software verification and validation (IEEE Std 1012-2012). *Software Quality Professional*, *15*(1), 43.

Giese, H., Hildebrandt, S., & Lambers, L. (2014). Bridging the gap between formal semantics and implementation of triple graph grammars - ensuring conformance of relational model transformation specifications and implementations. *Softw. Syst. Model.*, *13*(1), 273–299.

Giese, H., Hildebrandt, S., & Neumann, S. (2010). Model synchronization at work: Keeping SysML and AUTOSAR models consistent. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, & B. Westfechtel (Eds.), *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday* (Vol. 5765, pp. 555–579). Springer.

Guerra, E., de Lara, J., & Orejas, F. (2009). Pattern-based model-to-model transformation: Handling attribute conditions. In R. F. Paige (Ed.), *ICMT@TOOLS 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings* (Vol. 5563, pp. 83–99). Springer.

Hermann, F., Gottmann, S., Nachtigall, N., Ehrig, H., Braatz, B., Morelli, G., … Ermel, C. (2014). Triple graph grammars in the large for translating satellite procedures. In D. D. Ruscio & D. Varró (Eds.), *ICMT@STAF 2014, York, UK, July 21-22, 2014. Proceedings* (Vol. 8568, pp. 122–137). Springer.

Hoang, T. S. (2013). An introduction to the event-b modelling method. *Industrial Deployment of System Engineering Methods*, 211–236.

Holt, J., & Perry, S. (2019). *SysML for systems engineering: A model-based approach*. Institution of Engineering and Technology.

Huang, E., McGinnis, L. F., & Mitchell, S. W. (2020). Verifying SysML activity diagrams using formal transformation to petri nets. *Systems Engineering*, *23*(1), 118–135.

Ko, H., Zan, T., & Hu, Z. (2016). BiGUL: a formally verified core language for putback-based bidirectional programming. In M. Erwig & T. Rompf (Eds.), *PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, Proceedings* (pp. 61–72). ACM.

Lambers, L., Hildebrandt, S., Giese, H., & Orejas, F. (2012). Attribute handling for bidirectional model transformations: The triple graph grammar case. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, *49*.

Object Management Group. (2015). *OMG unified modeling language TM (OMG UML), version 2.5* (No. March). https://www.omg.org/spec/UML/2.5/PDF.

Object Management Group. (2019, November). *OMG systems modeling language (OMG SysML), version 1.6.* https://www.omg.org/spec/SysML/1.6/PDF.

Pais, R., Barros, J. P., & Gomes, L. (2014). From SysML state machines to petri nets using ATL transformations. In *Doctoral conference on computing, electrical and industrial systems* (pp. 227–236).

Schürr, A. (1994). Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, & G. Tinhofer (Eds.), *WG 1994, Herrsching, Germany, June 16-18, 1994, Proceedings* (Vol. 903, pp. 151–163). Springer.

Schürr, A., & Klar, F. (2008). 15 years of triple graph grammars. In H. Ehrig, R. Heckel, G. Rozenberg, & G. Taentzer (Eds.), *ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings* (Vol. 5214, pp. 411–425). Springer.

Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, *20*(5), 42–45.

Snook, C. F., & Butler, M. J. (2006). UML-B: formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, *15*(1), 92–122.

Stevens, P. (2017). Bidirectional transformations in the large. In *MODELS 2017, Austin, TX, USA, September 17-22, 2017, Proceedings* (pp. 1–11). IEEE Computer Society.

Weidmann, N., Anjorin, A., & Cheney, J. (2020). VICToRy: Visual interactive consistency management in tolerant rule-based systems. In B. Hoffmann & M. Minas (Eds.), *GCM@STAF 2020, Online-Workshop, 24th June 2020, Proceedings* (Vol. 330, pp. 1–12).

Weidmann, N., Anjorin, A., Fritsche, L., Varró, G., Schürr, A., & Leblebici, E. (2019). Incremental bidirectional model transformation with eMoflon::IBeX. In J. Cheney & H. Ko (Eds.), *Bx@PLW 2019, Philadelphia, PA, USA, June 4, 2019, Proceedings* (Vol. 2355, pp. 45–55). CEUR-WS.org.

Zolotas, A., Rodriguez, H. H., Hutchesson, S., Piña, B. S., Grigg, A., Li, M., … Paige, R. F. (2020). Bridging proprietary modelling and open-source model management tools: the case of PTC integrity modeller and epsilon. *Softw. Syst. Model.*, *19*(1), 17–38.