Abstract—We present a novel approach for modeling cyber-physical systems for analysis and test purposes. Instead of creating a new expressive specification language with sophisticated semantics and complex compilers, we rely on a lightweight version of Back’s Action Systems, for which we provide a simple bounded model checker using the SMT solver Z3. In order to model industrial-sized embedded systems, we extend our simple specification language by using the powerful capabilities of the modern programming language Scala for creating Domain Specific Languages (DSL). This enables us to use the features of an expressive, object-oriented and functional general-purpose language without the need to increase the complexity of the model checker. We demonstrate how to model a railway interlocking system with a configurable track layout and sketch the application to model-based testing.

I. INTRODUCTION

When modeling computer systems for verification and validation (V&V) purposes one usually distinguishes among two approaches: analysis of rich general purpose specification languages and the use of specification languages with a reduced feature set, but with clear formal semantics. Especially for model-based testing and model checking the latter approach has dominated, e.g. in NuSMV [1], Prism [2], and JTorX [3]. In this work we describe a method to combine the above mentioned approaches by using the capabilities of the modern programming language Scala for creating a so-called embedded Domain Specific Language (DSL). Scala is both functional and object-oriented and it compiles down to Java bytecode.

We are not the first to use embedded DSLs for V&V purposes. Havelund has proposed the use of Scala DSLs for multiple purposes: as a means to provide an expressive language for runtime verification [4], [5], and as alternative to a specification language [6]. Miller et al. [7] use a Scala DSL for simulation. In contrast, we focus on model-based testing. Our DSL is a flexible and agile way to provide a front-end for a language, which is designed for test models.

Model-based testing tools often implement translations from high-level specification languages to simple transition systems. E.g. in fault-based testing tools, early work [8] dealt with a transformation from LOTOS specifications to labeled transition systems (LTS), recent tools perform a translation from UML to LTS via action systems [9].
A. Action Systems in Scala

Action systems are written as Scala code. We define their abstract syntax tree (AST) as class hierarchy and provide operators to make expressions more readable. In this section we give a detailed description of our syntax. Whenever there is an operator or another short-hand defined for a class we add a comment. Later on we will show how these operators are defined and can be used to enhance existing code. To define our AST we use case classes, a Scala feature that allows for pattern matching and creation of instances without the keyword new.

In Listing 2 we show the top-level element of the AST representing an action system: it consists of a list of initial assignments of all state variables, called initBlock (Line 1) and a list of actions. In Scala the member variables of a class can be written in brackets behind the class name. This generates public and immutable members as well as a default constructor that takes values for all fields. Type annotations are written behind the respective field, separated by a colon. Scala supports parametric polymorphism, the type parameters are written behind a type enclosed with square brackets.

An underscore as type parameter is treated as wildcard, it can be any type e.g. \( \text{List[Var[_]]} \) is a List ofVars of possibly different types. An action consists of a unique name (Line 4), a list of parameters, a guard that defines when the action is enabled, a body of multiple assignments to state variables, which are executed in parallel, and an ActionType to mark actions as Input, Output or Internal.

Expressions of our flat action system are represented as a generalized algebraic data type as shown in Listing 3. Kennedy et. al. showed how to encode generalized algebraic data types into an object-oriented language [17]. The Scala type checker ensures that only expressions that are well typed are representable by the AST. There are three different kinds

```
Listing 1: Calculator Example
1 def (result, storage) = (Var("result", IntT()), Var("storage", IntT()))
2 def param = Var("param", IntT())
3 def param = Var("param", IntT())
4 ActionSystem(List(result := 0, storage := 0), List(//)
5   Action("store", List(), TRUE,
6     List(), TRUE,
7     List(), TRUE,
8     List(), TRUE,
9     List(), TRUE,
10    Action("show", List(param), (result === param),
11       List(), Output),
12    Action("add", List(param), TRUE,
13       List(result := result + param))))
```

```
Listing 2: Action System
1 case class ActionSystem(initBlock: List[Assign[_]],
2   actions: List[Action])
3 case class Action(name: String, params: List[Var[_]],
4   guard: Exp[BoolT], body: List[Assign[_]],
5   actionType: ActionType)
6 case class Assign[T](variable: Var[T], // :=
7   expression: Exp[T])
```

```
Listing 3: Expression AST
1 case class IntT()
2 case class BoolT()
3 case class EnumT[E](values:List[String])
4 sealed abstract class Exp[T]
5 case class BoolLit(b:Boolean) // TRUE / FALSE
6 extends Exp[BoolT]
7 case class Not(b:Exp[BoolT]) // !
8 extends Exp[BoolT]
9 case class And(l:Exp[BoolT], r:Exp[BoolT]) // &&
10 extends Exp[BoolT]
11 case class Or(l:Exp[BoolT], r:Exp[BoolT]) // ||
12 extends Exp[BoolT]
13 case class Equals[l:T, r:Exp[T]] // ==
14 extends Exp[T]
15 ... 
16 case class Equals[l:T, r:Exp[T]] // ===
17 extends Exp[T]
18 case class EnumLit[E](c:String, ty:EnumT[E])
19 extends Exp[T]
20 case class Var[T](name:String, ty:T) extends Exp[T]
```

We conclude our work.
of expression types `IntT` (Line 1), `BooleanT` (Line 2) and `EnumT` (Line 3) which contains a type variable to distinguish different user defined enumerations. This type variable is only used at compile time to ensure that only enumerations of the same type can be compared or assigned to. It is a so-called phantom type variable [18].

```
Listing 4: Infix Operators
1 implicit class BoolExpOps[l : Exp[BooleanT]] {  
2   def unary_! = Not(l)  
3 implicit class ExpOps[T](l : Exp[T]) {  
4   def unary_! = Not(l)  
5   def == (r : Exp[T]) = Equals(l, r)  
6   def != (r : Exp[T]) = Not(Equals(l, r))  
7 }  
```

We use Scala’s implicit classes, to define the operators mentioned before. This feature can be used to add methods to existing classes. In Listing 4 we define four operators. Note that methods in Scala can have names that consist of special characters and not only of alpha-numerical characters, it is also possible to call methods without a dot and parentheses. E.g. the expression `result===param` (Listing 1 Line 10) is equivalent to `result.==(param)`, without the definition in Listing 4 Line 6 we would need to write `Equals(result, param)`, which would not feel as natural. One special case is the definition of prefix operators: if a method name starts with `unary_` followed by a non alpha-numerical character it can be used as a prefix operator. Because of the definition in Line 2 it is possible to write `TRUE.unary_!()`.

Operators can not only be used to define a synonym for a single tree node, but also to define shortcuts for more complex operations, e.g., in Line 7 `!=` is defined as the negation of `Equals`. This allows to provide an expressive language to the user without the need to change the representation of the syntax, it is also possible to provide different operators for different user preferences. It is not necessary that the abstract syntax is defined in the same project or even in a project one has control over, our techniques can be used to enhance any syntax tree defined in Scala or Java. We show how to further improve the creation of models by using object-orientation and higher-order functions in Section IV.

**B. Semantics**

A predicative semantics for action systems was given by Aichernig et al. [19]. We use a simplified version of action systems following the syntax of Listing 2. The semantics is as follows: while one or more actions are enabled, one is chosen non-deterministically and executed. An action is enabled in some state, if there exists a post-state that satisfies the semantic predicate. If no action is enabled, the action system terminates. In Definition 1 the formal semantics of an action system, is given as a function `[, ]`, that maps the syntax to predicates. These predicates describe the relation between the variables before `v = ⟨v_1, v_2, …⟩` and after the execution of an action `v’ = ⟨v_1’, v_2’, …⟩`.

**Definition 1 (Transition Relation):** Given an action system consisting of the action sequence `⟨Action_1, … Action_n⟩` the transition relation `T` is defined as follows:

\[
\begin{align*}
\langle Action_1, … Action_n \rangle &= \exists n \ (Action_1 \lor … \lor Action_n) \\
Action(n, p, g, Assign) &= \exists p : (g \land Assign \land tr = tr \land \langle n(p) \rangle) \\
\langle Assign(v_1, e_1), … Assign(v_k, e_k) \rangle &= \exists v_1 = e_1 \land … \\
&v_k = e_k \land v_{k+1} = v_{k+1} \land … \land v_n = v_n
\end{align*}
\]

The non-deterministic choice between actions is realized by a disjunction between their semantic predicates. The names and parameters of the actions form a trace `tr`, whenever an action is executed the trace is updated to `tr’` by appending the name and the selected parameters, of the executed action. The transition relation of an action is defined as the conjunction of its guard `g`, the list of assignments `Assign` and the addition of name `n` with parameters `p` to the trace `tr`. The parameters are added as local variables to the predicate. The guard `g` can be used directly in the predicate, because it is a condition. A list of assignments `Assign` is defined as a parallel assignment, i.e. all state variables are updated at the same time. We partition the variable set `Var` into the set of variables `{v_1, …, v_k}` that are updated in `Assign` and a set of variables `{v_k+1, …, v_n}` that keep their value. Every post state variable `v_i'` is set equal to the evaluation of the expression `e_i`, all unassigned variables `v_u'` are set equal to their value in the previous state `v_u`.

**Example 2:** The semantics of action `add` (Listing 1 Line 12) is:

\[
\exists param \ (True \land result_{i+1} = result_i + param \land storage_{i+1} = storage_i \land tr_{i+1} = tr_i \land \langle add(param) \rangle)
\]

**III. Test-Case Generation**

Our test-case generation approach is fault-based. We generate a set of so-called mutants by copying the original specification and injecting a fault in each copy. We then use a refinement check and a reachability analysis to find a trace, which reveals the difference between original specification and mutant. The reachability analysis is based on the bounded model-checking algorithm proposed by Biere et al. [20]. The obtained trace can be used to test, whether a system implements the original specification or a mutant.

**Definition 2 (Fault-Based Test-Case Generation):** Given the transition relations `T_{spec}` and `T_{mutant}` and a bound `k`, the formula

\[
T_{mutant}(u, p) \land \neg T_{spec}(u, p) \land I(s_0) \land \\
\bigwedge_{i=0}^{k-1} T_{spec}(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} s_i = u
\]

is satisfiable iff there exists a trace leading to a faulty behavior. A model of this formula contains this trace. The action system is translated to an SMT formula representing the step relation `T` according to the semantics given in
Definition 1. We want to find an unsafe state \( u \) where the transition relation \( T_{\text{mutant}} \) does not conform to the transition relation \( T_{\text{spec}} \). The formula \( T_{\text{mutant}}(u, p) \land \neg T_{\text{spec}}(u, p) \) finds a state \( u \) where a non conforming step is possible. It is unsatisfiable if the mutant conforms to the specification. In this case it is not possible to generate a test case. The rest of the formula states as symbolic model-checking problem whether an unsafe state is reachable within \( k \) steps. The state variables in \( s_0 \) are set equal to the values assigned in the \textit{init} block (Listing 2 Line 1) and the transition relation is unrolled \( k \) times: 
\[
I(s_0) \land T_{\text{spec}}(s_0, s_1) \land T_{\text{spec}}(s_1, s_2) \land \cdots \land T_{\text{spec}}(s_{k-1}, s_k).
\]
Then it is checked if one of the states is an unsafe state: 
\[
s_0 = u \lor s_1 = u \lor \cdots \lor s_k = u.
\]

In Listing 5 we show a part of the translation of our expression AST to an SMT formula. Note that for using the SMT solver Z3 in Scala, there are two different APIs available: the Java API [21] and native Scala bindings [22]. For our implementation we chose the Java API, since it is maintained more actively. We also note that our action system AST is already strongly typed, so we can safely construct SMT ASTs without having to check types again.

Listing 5: Translation of a Boolean Expression

```scala
1 def translate(exp: Exp[Boolean]) : BoolExpr = exp match {
2   case BoolLit(b) => mkBool(b)
3   case Not(b) => mkNot(translate(b))
4   case And(l, r) => mkAnd(translate(l), translate(r))
5   case Or(l, r) => mkOr(translate(l), translate(r))
6   case _ => throw new Exception("Invalid expression")
7 }
```

The Java API for Z3 provides a 

Listing 6: ModelClass

```scala
1 abstract class ModelClass(name : String) {
2   def vars : List[Var[_]] = init.map(_ variable)
3   def actions : List[Action]
4   def Member[T](valName: String, ty : T) = Var("name", ty)
5   def Param[T](valName: String, ty : T) = Var("name", ty)
6   def init : List[Assign[_]]
7   def [T](valName: String, ty : T) = 
8   
```

A. Object Orientation

The first concept we introduce is object orientation: we allow to group actions and state variables into classes, which can be instantiated multiple times. All classes that are used inside the test model have a common super class \texttt{ModelClass}. This class is provided by the DSL and is defined in Listing 6.

The parameter \texttt{name} (Line 1) is used for the name of the actual object that instantiates the model class. The function \texttt{init} (Line 2) is overridden by the actual implementation and is set to a list containing the initial value of each variable in the form of an assignment. The function \texttt{vars} (Line 3) is a shortcut to provide just a list of variables. The function \texttt{actions} (Line 4) is again overridden by the actual implementation with a list containing all actions that correspond to the model class. The functions \texttt{Member} and \texttt{Param} are used as constructors for member variables and parameters for actions respectively. Their purpose is to map variable identifiers of a model class down to a unique name in the flat action system by using the name of the object as a prefix.

Example 4: Assume we want to model a system consisting of two calculators using a common storage variable as illustrated in Listing 7. Instead of duplicating all actions by hand, we create a case class \texttt{MyCalc}. Since \texttt{storage} is global, we define it outside of the class and use directly the \texttt{Var} constructor (Line 1). We override the \texttt{actions} (Line 5) and \texttt{init} (Line 6). Since \texttt{result} is specific to an instance, we use \texttt{Member} as constructor (Line 7). Analogously we use the constructor \texttt{Param} in order to define \texttt{param}.

We define the action \texttt{store} directly with the \texttt{Action} constructor and have to concatenate the name of the object and the name of the action by hand (Line 10). Note that in guard and body all variables can be accessed naturally by their name without having to care about the scope. The instantiation of the classes is shown in Line 13. In Line 14 the \texttt{init} block is composed. This is done by gaining a list in the behavior of the SUT, the trace has to be extended. Aichernig et al. [23] have shown such an extension can be performed by an input-output conformance check. For the example trace above, an additional call of the \texttt{show} action (Listing 1 Line 10) would be necessary to propagate the faulty state to an observation.

IV. Programming Test Models

We now show methods that enable us to introduce new layers of abstraction. They allow us to write generic rules, which are resolved during run-time before test-case generation.

Example 5: Assume we want to model a system consisting of two calculators using a common storage variable as illustrated in Listing 7. Instead of duplicating all actions by hand, we create a case class \texttt{MyCalc}. Since \texttt{storage} is global, we define it outside of the class and use directly the \texttt{Var} constructor (Line 1). We override the \texttt{actions} (Line 5) and \texttt{init} (Line 6). Since \texttt{result} is specific to an instance, we use \texttt{Member} as constructor (Line 7). Analogously we use the constructor \texttt{Param} in order to define \texttt{param}.

We define the action \texttt{store} directly with the \texttt{Action} constructor and have to concatenate the name of the object and the name of the action by hand (Line 10). Note that in guard and body all variables can be accessed naturally by their name without having to care about the scope. The instantiation of the classes is shown in Line 13. In Line 14 the \texttt{init} block is composed. This is done by gaining a list
of all members of each instantiated object and concatenating an initial assignment for the global variable storage. The AST can then be constructed by using the constructor `ActionSystem` with the `initBlock` and the list of all actions of all objects (Line 16).

**Listing 7: Calculator Example OOP**

```scala
val allCalcs = List(MyCalc("Calc1"), MyCalc("Calc2"))
val ast = ActionSystem(initBlock, allCalcs.map(c => c.result := 0), Input)
```

**B. Enumerations**

As a second programming technique we show how to conveniently create enumeration types in the test models. In our code we need an object of the case class `EnumT[E]`, the set of literals and a phantom type for the type parameter E. In order to conveniently access literals we want to have a Scala value for each literal, evaluating to the actual AST element.

In Listing 8 we present a function `genEnum`. The function takes a String `prefix` which is used to create unique internal identifiers and an Integer `size` determining the number of produced enumeration literals. The function returns an `EnumT[E]` and a list of `EnumLit[E]`. Pattern matching can be used to bind all objects to Scala values.

**Listing 8: Enum Definition**

```scala
def genEnum[E](prefix: String, size: Int) :
  (EnumT[E], List[EnumLit[E]]) = {
    val ty = EnumT[E]()
    val tyList = (0 to size).map(prefix + _).toList
    (ty, tyList.map(Enamlit(_, ty)))
  }
```

**Example 5:**

Assume we want to introduce an enumeration type for the different arithmetic operations our calculator supports. Listing 9 shows, how this can be done by applying `genEnum`.

**Listing 9: Enum Definition Example**

```scala
case class Operator()
val (_OPERATOR, List(doAdd, doMult, doDiv)) = genEnum[Operator]("OPERATOR", 3)
```

**C. Higher-Order Functions**

Instead of isolated objects, models are often composed of collections that do not change at runtime, e.g., the track layout of an interlocking system. In this case they can be modeled as data structures in Scala and later translated down to a flat representation. This allows for the usage of higher-order functions to manipulate them while creating the model. We can use map or flatMap to write generic rules that are instantiated for every object in a list.

Another example is the resolution of quantifiers. In modeling we often need to express that a predicate holds for all, or at least one object in some set of objects. We can naturally express this with quantifiers, but even though quantifiers are available in SMT solvers, flattening them down to a chain of conjunctions or disjunctions leads to formulas that can be solved more efficiently. Listing 10 shows the implementation for a forall quantifier using the higher order function foldRight.

**Listing 10: Forall**

```scala
val allCalcs = List(MyCalc("Calc1"), MyCalc("Calc2"))
def initBlock = allCalcs.flatMap(Param("param", IntT()))
def result = Member("result", IntT())
def store = Action(name="_store", Nil, TRUE, List(storage := result), Input)
val ast = ActionSystem(initBlock, allCalcs.flatMap(_.action))
```

**Example 6:** We add a global action `reset`, to our calculator: it is enabled if the result of all calculators is greater than 5. If executed, it resets all results back to zero.

**Listing 11: Higher Order Functions Example**

```scala
Action("reset",List(),forall(allCalcs)(c => c.result>5), ([storage := result], Input))
```

**D. Guarded Assignments**

Now we introduce the support for conditional assignments, which is not directly available in our back-end language. Instead we simulate them by introducing additional parameters and moving the conditions into the guard.

In Listing 12 we define a case class `GuardedAssignment`, an infix operator `when`, and a case class `ActionBuilder`. A guarded assignment consists of an assignment and a condition under which it is executed. The operator `when` has on the left-hand side the assignment and on the right hand side the condition. Objects of the class hold a list of additional action parameters and a list of guarded assignments. `ActionBuilders` can be composed by building the union set of action parameters and concatenating the list of guarded assignments. Finally the function `toAction` can be called, which generates an action with the given name, the given guard (aGuard) and the given action type (aType).

**Listing 12: Guarded Assignment**

```scala
implicit class AssignOps[T](assignment : Assign[T]) {
  def when(guard : Exp[Boolean]) = GuardedAssignment(assignment, guard) 
}
case class GuardedAssignment[T](assignment : Assign[T], guard : Exp[Boolean])
case class ActionBuilder[inputParams : List[Var[_]], assignments : List[GuardedAssignment[_]]] {
  def toAction(name: String, aGuard : Exp[Boolean], aType : ActionType) = {
    ...
  }
}
```

In Listing 12 we define a case class `GuardedAssignment`, an infix operator `when`, and a case class `ActionBuilder`. A guarded assignment consists of an assignment and a condition under which it is executed. The operator `when` has on the left-hand side the assignment and on the right hand side the condition. Objects of the class hold a list of additional action parameters and a list of guarded assignments. `ActionBuilders` can be composed by building the union set of action parameters and concatenating the list of guarded assignments. Finally the function `toAction` can be called, which generates an action with the given name, the given guard (aGuard) and the given action type (aType).

**Example 7:** Consider the calculator example. Assume we want to add two arithmetic operations division and multiplication and compose all of them to one action `compute`, which takes both the operand and the operator as parameter. Listing 13 demonstrates, how such a composed action can be written using the `ActionBuilder`.
Listing 13: Action Builder

```
1 ActionBuilder{
2 List(input, operator), // action parameters
3 (result := result + input) when operator === doAdd,
4 (result := result / input) when operator === doDiv,
5 (result := result * input) when operator === doMult
6 }, toAction("compute",
7 (operator === doDiv) ==> (input !== 0), Input)
8 }
```

In Listing 14 we show, how it would look like if the action would be written by hand. The trick is to introduce additional parameters and use them as temporary variables. For every state variable (result) that is updated in a guarded assignment of the action builder, we create such a parameter (result1). We then use the guard in order to restrict the value of the parameter to the value we actually want to assign to the state variable. This is done by transforming the guarded assignment to an implication and replacing the assignment with an equality statement. Additionally we introduce a skip assignment to an implication and replacing the assignment to the state variable. This is done by transforming the guarded value of the parameter to the value we actually want to assign.

Listing 14: Action Builder desugared

```
1 Action{
2 "compute",
3 List(input, operator, result),
4 (operator === doDiv) ==> (result === result / input) &
5 (operator === doAdd) ==> (result === result + input) &
6 (operator === doDiv) ==> (result === result / input) &
7 (operator === doMult) ==> (result === result * input) &
8 !(operator === doAdd) ||
9 operator === doDiv ||
10 operator === doMult ||
11 List(result := result), Input)
```

V. CASE STUDY: INTERLOCKING SYSTEM

A. Description

The objective of our case study is the modeling of a railway interlocking system. Such a system monitors elements of a railway station and allows the operator to establish safe thus non-conflicting train routes, on which a train can pass. A train route starts and ends at a signal and determines the way consisting of tracks and switches. The set of allowed train routes is pre-defined by the manufacturer.

The operator issues commands to the system like requesting or cancelling a train route, or the manual movement of a switch. When a train route is active, the interlocking system monitors the movement of a train using the route. So-called track supervision elements tell the system, whether a track or switch is occupied. If a train has passed all elements of a train route in the right order and stands within the so-called goal area, the train route is dissolved automatically.

B. Modeling

Figure 1 shows the class hierarchy of our test model as UML diagram. The associations visualize fields (members). The base class of the test model is ModelClass, which is imported from our DSL. TrainRoute is used to model the behavior of train routes and store the layout of the train station. The latter is done in layout, which lists the track elements that are part of the train route beginning at the start signal and ending at the goal signal in the order, in which a train passes them. In positions we store the desired position of each switch occurring in the train route. The type of state is _TrainRouteState, which has the following literals: _IDLE – the train route is not established, _ADMISSIBILITY_CHECK – the system is determining whether the train route is admissible, _SET_UP – the system is moving all switches into the desired position, _SIGNALCLEARING – the system switches the start signal to a go position, and _SUPERVISION – the train route is established and the tracks are monitored for passing trains.

RouteElement serves as base class for all elements that can be part of the train route. A Signal can be the start and the goal of a train route. If it does not show stop, a train may pass. The operator can manually lock a signal (flag lock), which hinders a train route that starts there to be admissible.

The abstract class TrackElement serves as base class for both straight Tracks and Switches. The parameter occupied models the information of the track supervision element whether the element is occupied. A state variable usage models the state of an element within a train route. Possible literals are: _UNUSED – the element is not part of an established train route, _NOT_YET_FREE – the element is occupied and is in a train route that is admissible and about to be established, _FREE – the element is not occupied and part of a train route that is established or about to be established, _OCCUPIED – the element is occupied, _HAD_BEEN_OCCUPIED – the element has been occupied since the route has been established, and _DISSOLVED – a train has passed and all previous elements have been dissolved.

Switch defines three state variables: interlocked – a train route containing the switch is established, locked – the switch is manually locked by the operator and position of type _Direction stores the current position of the switch. Possible literals for the latter are: _UNKNOWN, _LEFT, and _RIGHT.

Listing 15: Sample operator command

```
1 def requestTrainRouteAction =
2 Action{
3 name="request",
4 List(), // no parameters
5 state == _IDLE && activeTrainRoute == _NONE,
6 List(state := _ADMISSIBILITY_CHECK,
7 activeTrainRoute := trainRouteID(this)),
8 Input }
```

In Listing 15 we show an operator command for requesting a train route. This input action has no parameter, since the given train route is already part of the name of the action (Line 3). Within the guard (Line 5), we check, whether the current state is _IDLE and no other train route is performing an admissibility check (activeTrainRoute). The body sets the state of the train route to _ADMISSIBILITY_CHECK and the active train route to a self reference (Line 7).
In the following, we show how to model the processing of the inputs from the track supervision elements. We decided to do this in one action, so that it can be performed in one step of the transition relation. The rules, how the usage state of a track element changes, depend on the position of a track element within the layout of a train route. We define the rules within `TrainRoute`, using `ActionBuilder` and compose them into a global input action.

Some rules do not only depend on a single track element, but on a track element, all elements before it, and the element behind it. We define a list `partitioning`, that consists of one triple per track element, except the last one, to express these rules. The triples are composed of a list of previous elements, the current element and the next element:

```scala
val partitioning = (layout.initz, layout, layout.tail).zipped.toList
```

Listing 16: Sample rules for usage change

In Lines 7–12 we present a second, more complex rule that is instantiated by using `partitioning` from above. This allows for expressing the quantification in Line 11, which states that the rule only applies if one of the elements, which is located before the considered track, is in another usage state than dissolved. The pattern matching is used to give human readable names (before, current, next) to the elements of the partitioning. In Listing 17 we show how to compose the rules for the usage state.

Listing 17: Composing the axle counter input action

```scala
def axleCounterInputAction = allTrainRoutes.foldMap(_.axleCounterInputBuilder).toAction("axleCounterInput", TRUE, Input)
```

### C. Test-Case Generation

Now that we have shown the modeling of an interlocking system for a generic train station, we sketch the generation of a test case for a specific instantiation. Figure 2 shows a simple train station, consisting of ten straight tracks, two switches, and eight signals. We have defined eight train routes, one for each pair of consecutive signals, naming it after the start and the goal signal, e.g. the train route from signal $s_{1}$ to $s_{3}$ is named tr$_{13}$. We show how to manually construct a mutant, to demonstrate test-case generation.

Listing 18: Rule of action builder to be mutated.

```scala
val actionPart1 = partitioning.filter { case (_, _, next) => isInGoalArea(next).b }
```

Example.: Consider the rule in Listing 18. A mutation operator that replaces an enumeration literal in an assignment with another literal would be applicable. The assignment `next.usage := _DISSOLVED` would then be changed e.g. to `next.usage := _FREE`. Let us assume the fault has been
introduced in the train route $tr_{13}$. Performing the refinement check results in a trace, in which the train route $tr_{13}$ is established and a train passes all but the last track elements and finally stands on the track $TCH$ as illustrated in Figure 2.

Additional to the manual creation of mutants, our tool supports the automatic generation of mutants with mutation operators. The tool traverses the AST and creates copies of the model, with one changed element in each copy. We used the following mutation operators: change one enumeration literal to another one, negate a Boolean expression, replace a Boolean expression with $TRUE$ or $FALSE$ and remove a variable assignment. When applied to the example in Figure 2 we obtain 2816 mutants.

The test-case generation was performed up to a trace length of 15 and resulted in 1818 killed mutants. Our tool generates one test case per mutant without using information from previously killed mutants. Hence, 1818 test-cases were generated of which 293 turned out to be equivalent, resulting in a total number of 1525 test-cases. One obvious improvement would be to reduce the number of test cases, by reusing existing test cases to kill new mutants. This would lead to a reduced test suite that is cheaper to run on the SUT. The whole test-case generation process took about four hours on a standard notebook with an Intel i7 2.7GHz dual core processor and 16GB RAM. Note that the test-case generation was split into two parallel worker tasks each processing half of the model mutants.

VI. CONCLUSION AND FUTURE WORK

In this work we have shown, how to create specification models with the use of the modern, object-oriented and functional programming language Scala. The used specification language is a simple version of Back’s action systems. It only consists of guards, which are boolean expressions over a set of variables and bodies with parallel assignments to these variables. Thus it can be easily mapped to a transition relation for SMT-solving, but it is not limited to it and even simulation via execution is possible.

We have shown how to model a railway interlocking system using our approach. Analyzing such systems, e.g. for automated test-case generation or verification, can be a very complex task and often reveals poor scalability in verification tools for industrial case studies. Aichernig et. al [24] mention synchronous modeling as key factor for increasing the scalability. The reason for this lies in the processing of inputs from the track supervision elements to the interlocking system. If each input is modeled as discrete event, the number of possible interleavings explodes for combinatorial reasons.

In asynchronous modeling this challenge is usually faced by implementing so-called partial-order-reduction techniques. The downside of a synchronous modeling approach is that multiplexing inputs from different sources that operate on different sampling rates may lead to a counter-intuitive testing interface. In the case of the interlocking system the two input sources are the commands of the operator and the occupation information from the track supervision elements.

In our approach we chose a combination of synchronous and asynchronous modeling. Commands from the operator are treated as discrete events, while all inputs from the track supervision elements are processed in a single action.

Another conclusion that can be drawn from our approach is that not all complex data structures, which are used to express the behavior of the model, have to be part of the test model that is being analyzed by the model checker. In our railway interlocking case study, we used Scala lists to encode the layout of the train station. We wrote generic rules, which referred to those data structures, but when compiling it down to the AST that is processed by the back-end, only Boolean and enumeration type variables are left. Hence, we translate all Scala types of our DSL model to these two kinds of basic types. Even though SMT solving can handle more complex data structures in general, for scalability reasons it is still beneficial to limit the domain of variables.

We note that our DSL still targets power users and in future we can extend the sugaring for the railway domain so that it becomes more user friendly. Since our back-end language is very similar to Event-B, we also plan an export to Event-B, which would also benefit from the additional abstraction layers we have shown.

Acknowledgment: We are grateful to the anonymous reviewers for their valuable feedback. The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 332830 and from the Austrian Research Promotion Agency (FFG) under grant agreement N° 883898 for the implementation of the project CRYSTAL, Critical System Engineering Acceleration.

REFERENCES


