EMFeR: Model Checking for (EMF) Models*

Christoph Eickhoff
Kassel University
Kassel, Germany
christoph@uni-kassel.de

Martin Lange
Kassel University
Kassel, Germany
martin.lange@uni-kassel.de

Simon-Lennert Raesch
Kassel University
Kassel, Germany
raesch@uni-kassel.de

Albert Zündorf
Kassel University
Kassel, Germany
zuendorf@uni-kassel.de

ABSTRACT

For safety critical systems it is desirable to be able to proof system correctness. If your system is based e.g. on statecharts or finite automata you may use model checking techniques as provided e.g. by spin. If your system uses dynamic object models you may use tools like Alloy or graph based tools like Groove, Henshin, or SDMLib. Unfortunately, most of theses approaches use proprietary languages for the specification of models and model transformations. In order to verify system properties, one has to recode the system and its operations within the specific language of the used verification tool. This is tedious and error prone.

To overcome these limitations, this paper outlines our new EMFeR (EMF Engine for Reachability) tool. EMFeR provides complete testing and model checking capabilities for EMF based models and arbitrarily implemented transformations on such models. EMFeR computes reachability graphs based on EMF models and EMF model transformations. Basically, a reachability graph is a Linar Transition System where the states are formed by the different models and the transitions correspond to model transformations turning one state into another. Based on reachability graphs, EMFeR provides e.g. CTL (Computational Tree Logic) model checking and controller synthesis capabilities.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

model checking, controller synthesis, object models, EMF

ACM Reference Format:
Christoph Eickhoff, Martin Lange, Simon-Lennert Raesch, and Albert Zündorf. 2018. EMFeR: Model Checking for (EMF) Models. In *Proceedings of Models 2018.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.475/123_4

*Produces the permission block, and copyright information

Models 2018, October 2018, Copenhagen, Denmark

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

Let us assume, you have just build a new smart traffic light. Your smart traffic light has e.g. radar sensors to detect approaching cars and instead of switching periodically, it yields green on demand. Thus, when at rush hour times all the traffic goes in one direction, this direction gets green all the time. To implement this smart behavior you have used an EMF based object model that keeps track on car positions and traffic light states. Now, in order to deploy your smart traffic light in real world you need do get certified and thus you may need to proof system correctness.

Model Checking is a powerful formal method for the verification of e.g. liveness and safety features of parallel systems. There exists a number of powerful model checking tools like e.g. Spin [14]. For dynamic object models one may use formal tools like Alloy [15]. And the area of graph transformation tools provide reachability graph computation for similar purposes, cf. tools like Groove [12], Henshin [13], or SDMLib [19]. Unfortunately most of these approaches use proprietary languages for the specification of models and model transformations. In order to verify system properties, one has to recode the system and its operations within the specific language of the used verification tool. Thus, you basically specify or implement your smart traffic light, a second time. This is tedious and error prone. You will have to argue, that your implementation meets your model checking specification.

To overcome these limitations, this paper outlines our new EMFeR (EMF Engine for Reachability) tool [10]. EMFeR provides model checking capabilities for EMF [9] based models and arbitrarily implemented transformations on such models. Your model transformations may be implemented as plain Java, Xtend [21], using ATL [2], or any other approach. The model transformations are provided to EMFeR as Java 8 lambda expressions. Ideally, you may pass the actual method implementations that you want to use in your productive system to EMFeR in order to do an exhaustive testing of your system implementation. EMFeR applies your transformations to (clones of) a given model and to (clones of) the resulting models. Each time a new model is generated, EMFeR compares the new model with any previous model and checks whether a new model has been generated or whether an old model is reached again. To do this efficiently, EMFeR computes model certificates, i.e. hash keys, for each model, as proposed by the Groove system [12]. The process terminates when all possible models have been derived. The set of all generated models with links corresponding to the applied transformations forms a Labeled Transition System (LTS). In the

```
1  class RoadMap {
2     contains Road road
3     contains Car[] cars
4     contains Signal westernSignal
5     contains Signal easternSignal   }
6  class Road { contains Track[] tracks }
7  class Track {
8     String name
9     TravelDirection travelDirection
10    refers Track[] west opposite east
11    refers Track[] east opposite west
12    refers Signal signal opposite track
13    refers Car car opposite track   }
14 class Signal {
15    boolean pass
16    refers Track track opposite signal   }
17 class Car {
18    TravelDirection travelDirection
19    refers Track track opposite car   }
20 enum TravelDirection {UNDEFINED, EAST, WEST}
```

Listing 1: Road Work Class Model

```
1  class ReachabilityGraph {
2     contains ReachableState[] states
3              opposite parent
4     contains TrafoApplication[] trafoApplications
5              opposite parent }
6  class ReachableState {
7     long number
8     double metricValue
9     container ReachabilityGraph parent
10             opposite states
11    refers EObject root
12    refers TrafoApplication[] resultOf
13           opposite tgt
14    refers TrafoApplication[] trafoApplications
15           opposite src }
16 class TrafoApplication {
17    String description
18    container ReachabilityGraph parent
19             opposite trafoApplications
20    refers ReachableState src
21           opposite trafoApplications
22    refers ReachableState tgt
23           opposite resultOf }
```

Listing 2: Reachability Graph Class Model

context of graph transformations we call this LTS a reachability graph.

On the resulting reachability graph you may run CTL (Computational Tree Logic) [7] queries in order to model check e.g. safety and liveness features. You may also do any other model query in your favorite (EMF compatible) query language e.g. Java or OCL. You may also use the reachability graph for controller synthesis.

2 THE ROADWORK EXAMPLE

As running example for this paper we use a simple traffi system for a small one way roadwork area. This example stems from [11]. Listing 1 shows the class model of our example using XCore [20]. Listing 2 shows the class model of EMFeR's reachability graphs.

Figure 1 shows a simplified EMF object model for the start situation of our road work example. (EMFeR provides a simple HTML dump for object models based on Alchemy.js [1]. To allow simplifications, Figure 1 has been created, manually.) There are two lanes: the upper (northern) lane goes from right (east) to left (west) and consists of objects `n1` to `n7`. Above this lane there are two objects at the upper right corner of Figure 1. From right to left this are a car located at track object `n1` and going `WEST`. And next to it the `eastern` signal that is attached to track `n2` and currently shows green (`pass=true`). The middle row of Figure 1 shows on the left the `Road` object that contains all tracks and on the right the `RoadMap` object that contains the `Road`, all `Cars` and all `Signals`. The lower (southern) lane is represented by only four `Track` objects named `s1, s2` and `s6, s7` (numbering from left to right). The three middle objects of that lane are missing as they are blocked by road work. Instead, track `s2` is connected to track `n5` of the northern lane and track `n3` of the northern lane is continued by track `s6`. Each track has a `travelDirection` which equals to `WEST` for the northern tracks and to `EAST` for the southern tracks and to `UNDEFINED` for the tracks in the road work area. Finally, there are a car and a signal object in the lower left corner of Figure 1. The object structure for the initial situation is created using the standard `RoadworkFactory.eINSTANCE` generated by EMF.

Listing 3 shows how the reachability graph computation is invoked. Line 1 creates an `emfer` object. Line 2 adds the (root of) our start model as first state to EMFeR. EMFeR accepts simple transformations as Java lambda expressions with one (`root`) parameter. Via this `root` parameter EMFeR passes the EMF model that shall be transformed. Lines 3 to 4 add the simple `swapSignals` transformation to `emfer`. The transformation `swapSignals` operates our traffic lights, cf. Listing 5. In our example, transformation `swapSignals` is the (part of the) system implementation we want to verify, i.e. to test exhaustively. In contrast to graph transformations, EMFeR transformations are deterministic and produce only one new model state. However, sometimes one wants to apply a transformation on multiple model objects, one after the other. In our example there exist multiple `Car` objects, that may move, independently. For such cases, EMFeR accepts complex transformations that consist of a *path* and a *two-parameter* transformation. Lines 5 to 7 of Listing 3 add our `moveCar` transformation to emfer. Line 6 adds a lambda expression that computes the set of cars in the current model. At exploration time, EMFeR will call the second lambda expression (line 7) on each of these cars. In addition to the car that shall be moved, EMFeR also passes the root of the current model to the transformation in order to facilitate access to other model elements. Finally, line 8 starts the reachability graph computation.

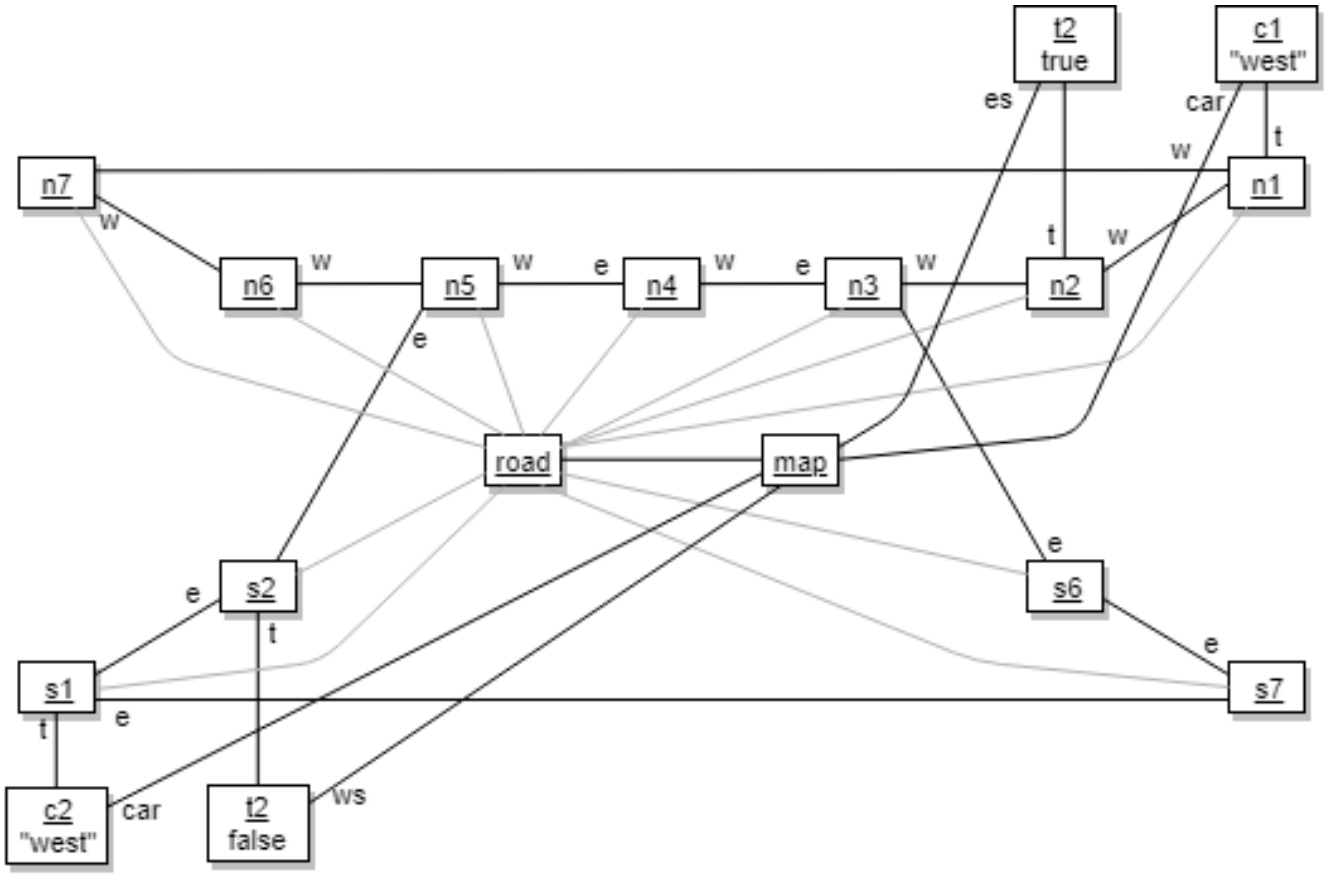

Figure 1: Start Situation (simplified)

Listing 4 shows pseudo code for EMFeR's reachability graph exploration operation. For each state (line 2) and each transformation (line 3), EMFeR first computes the handle objects, on which the `trafo` shall be applied (line 4). For complex transformation, line 4 uses the path lambda provided e.g. in Listing 3, line 6. For simple transformations, we just use the root as handle. Now, for each handle (Listing 4, line 5) we first clone the current model (line 6) and then we apply the current transformation to the cloned model passing the clones of root and handle as parameters. The transformation may modify the clone. This may result in a totally new model state. In this case we add the new model state to our reachability graph and connect it to the current model via an `TrafoApplication` link that carries the name of the transformation and the used handle as `description`. It may also happen, that the new state is isomorphic[1] to an old state, that has been created earlier. In this case we just add a `TrafoApplication` link between the current state and that old state. New states will also be considered in line 2 of our algorithm, i.e. we will apply all transformation on all handles of the new states, again. Thus, EMFeR's explore operation computes the set of all states that can be created by applying all trafos on all handles on all states derived from (and including) the start state, iteratively. Figure 2 shows an outline of the reachability graph generated by our example in Listing 3. Each node in Figure 2 represents / contains an EMF object model modeling the corresponding state. EMFeR provides a simple dump for reachability grahs where you may click on a node in the reachabilty graph view in order to show the contained object model.

Listing 5 shows our `swapSignals` transformation. This transformation has been implemented in plain Java. Basically, `swapSignals` checks that the road work area of our street (tracks with undefined travel direction) is clear of cars (lines 3 to 7). Next, there shall be a car waiting on red (line 19 to 20) and there shall be no car just in front of a green light (line 17 to 18). Under these conditions, the red signal becomes green and the green signal becomes red, lines 21 to 22. Method `swapSignals` is an example for an operation that may be used to actually operate our trafic signals in the final system. Thus the task at hand is, to proof that `swapSignals` works save and e.g. fair.

Listing 6 shows our `moveCar` transformation. A car may not move if it sees a red signal (line 6) or if the road is blocked by another car (line 25). Cars move in their travel direction (line 9 to 14) and on leaving the road work area they stay in their lane (line 20 to 23). If possible, line 26 moves the car to its new position. The

[1] We consider two EMF models as isomorphic if there is a bidirectional mapping between their objects that respects all attribute values and all references. Although EMF uses `ELists` for to-many references, we do NOT consider the order of references.

```
EMFeR emfer = new EMFeR()
    .withStart(roadMap)
    .withTrafo("swap Signals",
        root -> swapSignals(root))
    .withTrafo("move car",
        root -> ((RoadMap) root).getCars(),
        (root, car) -> moveCar(root, car));
int size = emfer.explore();
ReachableState startState =
    emfer.getReachabilityGraph().getStates()
        .get(0);
AlwaysGlobally alwaysGlobally =
    new AlwaysGlobally();
ExistFinally existFinally =
    new ExistFinally();
ExistGlobally existGlobally =
    new ExistGlobally();
boolean noDeadLock = alwaysGlobally
    .test(startState, s->!isCarDeadLock(s));
boolean unfair = existFinally
    .test(startState, s-> existGlobally
        .test(s, s2->isEastCarWaitsAtRed(s2)));
ArrayList<TrafoApplication> examplePath =
    existGlobally.getExamplePath();
System.out.println(examplePath);
```

Listing 3: Calling Emfer

`moveCar` operation is used to simulate car behavior. Thus `moveCar` would usually not become a part of the final system. Still, any proof for the `swapSignals` operations holds only, iff operation `moveCar` simulates the car behavior, "appropriately". In reality, there may be erroneous car behavior like driving on the wrong lane or having an engine crash in the road work area that may cause system failures not covered by our simulation. Similarly, the simulation may exhibit behavior you would not see in reality, e.g. cars waiting at green traffic lights.

3 REACHABILITY GRAPH EXPLORATION

EMFeR's algorithm for the exploration of reachability graphs is outlined in Listing 4. However there are a number of issues to be discussed in more detail.

First, when EMFeR has generated a new model it uses model certificates, i.e. hash keys, to efficiently identify possibly isomorphic old models. This follows the ideas of Arendt Rensink and his Groove system [12]. Thus, identifying isomorphic old states is reasonably fast. The main efficiency problem of our reachability graphs is the memory consumption.

In general, the reachability graph exploration might not terminate or the reachability graph may become very large. In our road work example the used transformations do not create new objects but just change links between existing objects or change some boolean attributes. Thus, in our road work example there is only a finite number of reachable states (56 states to be precise). However,

```
EMFeR::explore() {
   for each state {
      for each trafo {
         compute handles for trafo
         for each handle {
            clone current state
            apply trafo on clone
            if (new state)
               add trafo edge and
               new state to graph
            if (old state)
               add trafo edge to graph
         }
      }
   }
   return number of states
}
```

Listing 4: EMFeR explore

```
private void swapSignals(EObject root) {
   RoadMap roadMap = (RoadMap) root;
   for (Car c : roadMap.getCars()){
      if (c.getTrack().getTravelDirection()
          == UNDEFINED) {
          return; }
   }
   boolean carIsWaiting = false;
   // no car about to enter and one car
   // waiting at red light
   Signal west = roadMap.getWesternSignal()
   Signal east = roadMap.getEasternSignal();
   boolean carAtWest =
      west.getTrack().getCar() != null;
   boolean carAtEast =
      east.getTrack().getCar() != null;
   if (west.isPass() && carAtWest) return;
   if (east.isPass() && carAtEast) return;
   if (   west.isPass() && carAtEast
       || east.isPass() && carAtWest) {
      east.setPass( ! east.isPass());
      west.setPass( ! west.isPass());  } }
```

Listing 5: Controlling the traffic signals

if one employs e.g. a transformation that creates new cars or that extends the road or just a counter for the number of car moves done, the number of possible states would become infinite or just larger than we can handle. Currently, EMFeR holds the whole reachability graph within main memory (i.e. 8GB on Albert's laptop). For our road work example this are about 15 model objects per state (plus EList objects for to-many references). Depending on the size of your

```
1  private void moveCar(EObject root,
2                       EObject handle) {
3     RoadMap roadMap = (RoadMap) root;
4     Car car = (Car) handle;
5     Signal signal = car.getTrack().getSignal();
6     if (signal != null && ! signal.isPass())
7        return;
8     EList<Track> targets;
9     if (car.getTravelDirection()
10        == TravelDirection.EAST) {
11       targets = car.getTrack().getEast();
12    } else {
13       targets = car.getTrack().getWest();
14    }
15    Track newPos = null;
16    if (targets.size() == 0) return;
17    if (targets.size() == 1) {
18       newPos = targets.get(0);
19    } else {
20       for (Track t : targets) {
21          if (t.getTravelDirection()
22             == car.getTravelDirection())
23          {  newPos = t; }
24    }  }
25    if (newPos.getCar() != null) return;
26    car.setTrack(newPos);
27 }
```

Listing 6: Move Car Simulation

main memory, EMFeR may handle up to some million reachable states.

To avoid OutOfMemory exceptions, EMFeR has a customizable limit for the maximal number of reachable states it creates. This limit defaults to 300000. It may be adapted according to the sizes of the employed models and according to the memory space available. When the limit is reached, EMFeR just terminates the exploration and delivers a partial reachability graph. If you are lucky, the partial reachability graph already contains the states you are looking for. In our road work example this might be a dead lock state, where two cars on the single road work lane block each other.

Per default, EMFeR does a breadth first exploration of the reachability graph. This means, new states are managed in a fifo queue for further exploration. In [6] we extended SDMLib's reachability graph exploration algorithm with a metric computation provided as Java lambda expression. EMFeR has adopted this idea. The metric computation computes a metric value for each new state and then the queue of new states is sorted according to this metric value (minimal value first). Thereby, the metric computation steers the exploration strategy similar to an A* algorithm. This results in a hill climbing strategy for our reachability graphs where the most promising states are expanded, first. Actually, the result is a kind of taboo search as states that have been considered will not be expanded again. Thus, our exploration strategy will backtrack out of local optima (if there is still memory space left).

In the special case of Computational Tree Logic CTL [7] proof obligations, the checking of CTL operators and the expansion of the reachability graph may be inter wined. This would allow to stop the exploration as soon as a counter example (for always operators) or a positive example (for exist operators) has been found.

The space limitation problem exists also for traditional model checkers like Spin [14]: depending on your formulas and the available memory space, there is an upper bound for the number of boolean variables Spin can handle. Systems for dynamic object structures like Alloy [15] usually follow a breadth first strategy, too. To deal with the space limitation problem, traditional model checkers employ very efficient encodings for states. To improve EMFeR we implemented a lazy cloning strategy, where we clone only model elements that are modified and share unmodified model parts within multiple states. In our road work example the road and its tracks are not directly modified by our example transformations. Thus, the road and tracks object could be reused in all states and we may clone only the root object, the cars, and the signals. Thus each new state would clone 5 model objects and 12 model objects would be shared. However, in our example the bidirectional links between cars and tracks and between signals and tracks, cf. Listing 1 prevent the cloning of cars without cloning their current tracks. Similarly, the `contains` association prevents the sharing of kid objects between clones of the parent object. Thus, we could not share the same `Road` object within multiple `RoadMap` objects. To enable lazy cloning, we have to adapt our model by using unidirectional references instead of bidirectional references and by using unidirectional references instead of contains associations. According to these changes we need to modify our transformation operations where they use backward references, e.g. a track does no longer know whether it hosts a car. However, with these modifications for an application of the `moveCar` transformation our lazy cloning approach clones only the moved car and the root object, i.e. we share 15 out of 17 objects and the new reachable state needs only 2 new model objects (plus the object for the reachable state and an object for the `TrafoApplication`. Similarly, the `swapSignals` operation clones only the two modified signals and the root object, i.e. only 3 new model objects for the new state. In our example this reduces the number of model objects in the total reachability graph down to some 15%. In addition, omitting `opposite` references safes some memory space, too. On the down side, as we need to adapt our model and our model transformations, we either do not longer model check the actual operation code or the actual operation code needs to stick to the adapted model. As our adapted model does not longer use EMF `contains` associations, the usual EMF persistence mechanisms would no longer work. However, to some extend these are EMF specific problems and you could go for e.g. SDMLib based model implementations that avoid many of these problems. Generally, we plan to extend EMFeR to apply for other model implementations.

Another approach dealing with too large state spaces are abstraction techniques. In traditional model checking, integer attributes might e.g. be abstracted to attributes with values less than, equal, or greater than zero. Such techniques may be applied to EMFeR, too, but it would require to modify the underlying EMF models and the

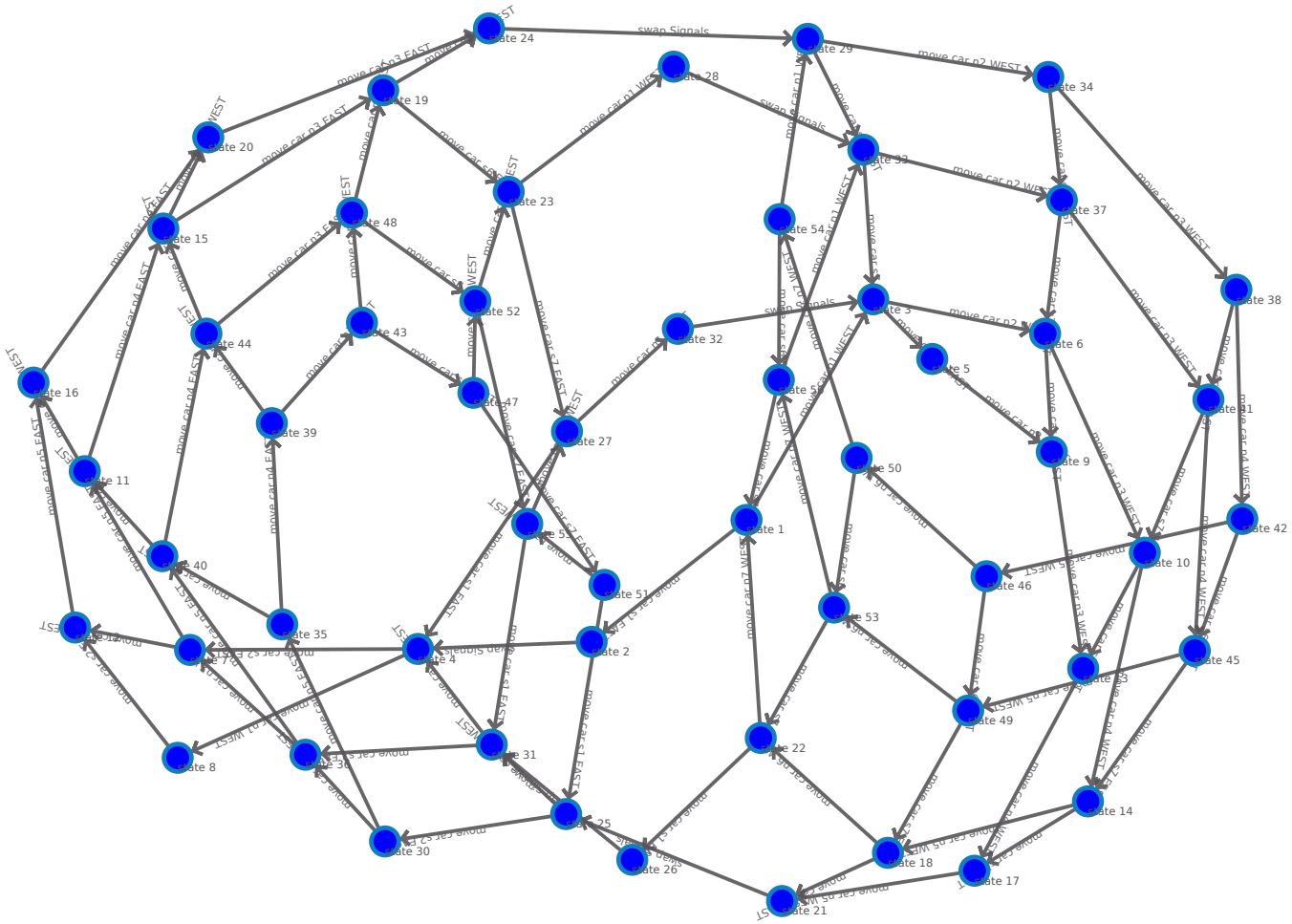

Figure 2: Roadwork reachability graph

model transformations to deal with the abstraction, explicitly. This kind of abstraction would thus require a certain recoding of your system which contradicts the original goals of EMFeR to model check your productive EMF system without recoding it within the language of some traditional model checker.

For graphs there exist abstraction mechanism where multiple objects are folded into a single object that then represents a complex substructure, cf. e.g. [3]. In our example we may e.g. interpret the signal track of each lane as folded objects that represent an arbitrary long list of tracks. Then we add simulation transformations that create new cars on the signal track of each lane and that remove cars when they reach the end of the road work area. This simulates an arbitrary large number of cars traveling on an arbitrary long road. Altogether, this abstraction simulates an infinite road with an infinite number of cars within a finite model with a finite reachability graph. Still, proofs done on this finite model may hold for the infinite road, too. The conditions under which this holds have been studied by Barbara König [3], very well. Basically, the abstraction has to guarantee that an abstract state still allows all transformations to execute that are executable on any of the original states represented by the abstract state. And the resulting abstract state must contain the resulting concrete states derived by this step.

4 REACHABILITY GRAPH ANALYSIS

Once a reachability graph has been computed, we may run all kinds of analysis and query operations on it. We may e.g. search for all states that contain a forbidden situation (two cars in the road work area traveling in opposite direction) or all transformation edges that connect a valid state with an invalid state. This gives insight on which model transformations may need enhanced preconditions in order to avoid invalid states. You may also search for shortest paths that lead from the start state to some final state where you assign e.g. specific costs to each model transformation. For such queries you may use plain Java or OCL or graph queries or any other appropriate query language. After all, our reachability graph is a simple (EMF) model, again.

For comparability reasons, EMFeR also supports Computational Tree Logic CTL [7] to analyze reachability graphs. EMFeR provides 8 CTL operators for all combinations of *always* and *exist* quantifiers

```
[2 --move car n1 WEST--> 4,
4 --move car n2 WEST--> 7,
7 --move car n3 WEST--> 11,
11 --move car n4 WEST--> 15,
15 --move car n5 WEST--> 19,
19 --move car n6 WEST--> 23,
23 --move car n7 WEST--> 2]
```

Listing 7: Car traveling east waits at red light, forever

with *finally*, *globally*, *next*, and *until* operators. Listing 3, lines 18 to 19 show the usage of the `AlwaysGlobally` operator. The operator is parameterized with a predicate to be tested on all states reachable from the start state. In EMFeR, we provide predicates as boolean Java lambda expressions that may be implemented in plain Java or e.g. using OCL or some other (your favorite) query language. Listing 3 line 20 to 22 shows how EMFeR CTL operators may be nested to form more complex queries. Line 20 to 22 search for an unfair situation, i.e. the east car waits at red light forever. Unfortunately, line 20 to 22 detect that transformation `swapSignals` from Listing 5 is unfair. Thus, the `ExistGlobally` operator of line 22 succeeds in finding a circle in our reachability graph where only the car traveling west moves and the other car waits for ever. On success, the *exist* operators generate an `example` path. (The *always* operators produce `counterExample` paths, on failure.) Listing 3, Line 25 prints the unfair example path. Listing 7 shows the output for our unfair example. In state 2, the car traveling east is waiting at its red light. The other car has not yet moved. From then on, only the car traveling west moves until it has done a full circle reaching state 2, again. When the circle is closed, the car traveling west may just do new circles for ever. In this case the car traveling east will starve to death at the red light.

The fairness problem of our current example system is easily solved, if the `swapSignals` transformations gets executed once the car traveling west leaves the road work area. Like Groove [12], EMFeR allows to force a set of transformations to be executed, if possible, by assigning priorities to them. Doing so, our example system becomes fair for two cars. If we add a second car on the upper lane, these two cars might take turns in blocking the road work area and thus they may block the signals from swapping, forever. To address this, we need to enable our signals to show red on both sides, in order to drain the road work area and then to give yield to the opposite direction. Instead of creating the more sophisticated behavior manually, we will use controller synthesis techniques provided by EMFeR in the next section.

5 CONTROLLER SYNTHESIS

The basic idea for controller synthesis with EMFeR is to provide EMFeR with all the basic operations, cf. Listing 8 lines 2 to 19. Next, we generate all possible situations, i.e. all possible reachable states, line 21. Line 22 to 23 construct a synthetic controller parameterized with the EMFeR reachability graph. Lines 24 to 27 tell the synthetic controller which transformations it may use to control the execution. All other transformations simulate the environment, in our case the car related operations. Next, line 28 to 30

```
1  EMFeR emfer = new EMFeR()
2  .withTrafo("move car", root->getCars(root),
3             (root,car)->moveCar(root, car))
4  .withTrafo("create car going east",
5             root -> createCarGoingEast(root))
6  .withTrafo("remove car going east",
7             root -> removeCarGoingEast(root))
8  .withTrafo("create car going west",
9             root -> createCarGoingWest(root))
10 .withTrafo("remove car going west",
11            root -> removeCarGoingWest(root))
12 .withTrafo("signal green green",
13            root -> signalGreenGreen(root))
14 .withTrafo("signal red green",
15            root -> signalRedGreen(root))
16 .withTrafo("signal green red",
17            root -> signalGreenRed(root))
18 .withTrafo("signal red red",
19            root -> signalRedRed(root))
20 .withStart(roadMap);
21 int size = emfer.explore();
22 SyntheticControl syntheticControl
23    = new SyntheticControl(emfer)
24 .withTrafo("signal green green")
25 .withTrafo("signal red green")
26 .withTrafo("signal green red")
27 .withTrafo("signal red red")
28 .withMetric( root -> dangerMetric(root))
29 .withMetric( root -> redWaitCosts(root))
30 .applyMetric();
31 EMFeR emfer2 = new EMFeR()
32 .withTrafo("move car", root -> getCars(root),
33      (root, car) -> moveCar(root, car), 1)
34 .withTrafo("create car going east",
35      root -> createCarGoingEast(root), 1)
36 .withTrafo("remove car going east",
37      root -> removeCarGoingEast(root), 1)
38 .withTrafo("create car going west",
39      root -> createCarGoingWest(root), 1)
40 .withTrafo("remove car going west",
41      root -> removeCarGoingWest(root), 1)
42 .withTrafo("synthetic control",
43      root -> syntheticControl.run(root), 0)
44 .withStart(roadMap);
45 int size2 = emfer2.explore();
```

Listing 8: Creating a synthetic controller

provide and apply two cost functions. Basically, the `dangerMetric` assigns `Integer.MAX_VALUE` to all reachable states where two cars are entering the road work area in opposite directions. In addition, `redWaitCosts` punishes states where cars are waiting at red lights.

```
1  public void run(EObject root) {
2     ReachableState currentState
3        = findMatchingOldState(root);
4     if (currentState == null)
5        return; // Unknown state.
6     double bestMetricValue
7        = currentState.getMetricValue();
8     TrafoApplication bestTrafoApp = null;
9     for (TrafoApplication trafoApp
10       : currentState.getTrafoApplications()) {
11       if (null != emfer.getPathTrafosList(
12                   trafoApp.getDescription())) {
13          if (trafoApp.getTgt().getMetricValue()
14                < bestMetricValue) {
15             bestTrafoApp = trafoApp;
16             bestMetricValue = trafoApp
17                .getTgt().getMetricValue();
18    } } }
19    if (bestTrafoApp != null) {
20       PathTrafo pathTrafo
21          = emfer.getPathTrafosList(
22             bestTrafoApp.getDescription());
23       pathTrafo.trafo.run(root, root);
24    }
25 }
```

Listing 9: synthetiControl.run()

Now our synthetic controller works as follows, cf. Listing 9: 1st the controller identifies the reachable state that matches the current situation, cf. line 2 to 3. 2nd the controller loops through all operations it may apply (line 9 to 12) and checks which operation would improve the situation (most) (line 13 to 14). If it is possible to improve the situation (line 19), the controller executes the best step (line 23). Otherwise, the controller does nothing (and the cars may move).

To test our synthetic controller, Listing 8 lines 31 to 45 run EMFeR again, now using the basic car operations and the synthetic controller. In our example, the first reachability graph with all the basic operations has 30720 reachable states. Using the synthetic controller, `emfer2` has only 2497 reachable states. All other situations are avoided by the synthetic controller as they have higher costs. We could actually create a second synthetic controller that is based on this smaller reachability graph. This would reduce the size of our synthetic controller but the reduced controller could only handle the 'good' situations that it allows. If some error occurs (power outage, irregular car movement, ...) the reduced controller would not know how to react, while the full controller covers much more situations. To be honest, to develop cost functions that result in a save and *fair* synthetic controller was quite tricky. Achieving safety is quite simple. However, achieving fairness was rather painful. Assume there are always new cars showing up traveling west while some car is waiting to travel east. To be fair, the controller will finally switch to 'red red' to drain the road work area in order to give yield to the other direction. However, once the 'red red' road work area has drained, the controller has to decide which direction should go next. To be fair, the controller needs to know, which was the previous traveling direction. To provide this information, we had to extend our model with a `lastTravelDirection` attribute. In addition, the `signalXY` operations have been extended to update the `lastTravelDirection` attribute, appropriately. Finally, the `redWaitCosts` assign higher costs to the car waiting longer (wanting to go the other direction). Overall, this simple approach for controller synthesis works fine, if the local information provided by a single reachable state suffices to come up with a reasonable cost model. However, if you need to consider some lengthy path through your reachability graph in order to come up with a good decision, a simple metric does not suffice.

To overcome these limitations, EMFeR provides a game-theoretic solution of the controller synthesis problem. In this 2-player game the *controller* (the traffic lights) has to choose actions (which light to turn green or red) against arbitrary moves of an *environment* (the cars) such that the system controlled in this way is safe and fair. The existence of such a controller is equivalent to the existence of a winning strategy in a model checking game [18] for a formula of the modal μ-calculus [16]. Here, game moves alternate strictly, and the traffic lights begin. Then the controller's choice of one action in this game can be expressed using a CTL existNext operation on top of an alwaysNext. In the modal μ-calculus this is written $\Diamond\Box S$ expressing the set of states from which a step can be made (by the controller) such that after any following steps (made by the cars) a state in set S is reached.

Consider the following system of recursive equations for sets P_{ew}, P, P_e, P_w of states.

$$
\begin{aligned}
P_{ew} &=_\nu \Diamond\Box P \\
P &=_\mu \Diamond\Box P \cup (\neg \texttt{isCarAt}(s, \texttt{EAST}) \cap P_e) \cup (\neg \texttt{isCarAt}(s, \texttt{WEST}) \cap P_w) \\
P_e &=_\mu \Diamond\Box P_e \cup (\neg \texttt{isCarAt}(s, \texttt{WEST}) \cap P_{ew}) \\
P_w &=_\mu \Diamond\Box P_w \cup (\neg \texttt{isCarAt}(s, \texttt{EAST}) \cap P_{ew})
\end{aligned}
$$

where s refers to the current state of evaluation, the index μ/ν indicates whether we define the *least*, resp. *greatest* solution of the corresponding equation, and equations listed first take precedence over those listed later. Hence, this system of equations represents a modal μ-calculus formula with a ν-μ pattern of fixpoint alternation which is typically used to describe fairness properties.

To understand the intuitive meaning of this recursive definition of states, it is best to consider the three μ-equations first, bearing in mind that least fixpoints describe reachability properties. For a given set P_{ew}, we have that P_x, $x \in \{e, w\}$ abbreviating `EAST` and `WEST` describes the set of states from which the controller can enforce a visit to a state in which the opposite entrance of the road works is empty. Then P describes those states from which a situation is reachable in which one side is empty and then later the other side will be empty too. Now the first equation enforces all states of the set P_{ew} to be followed by a state with property P. This means, we require that both sides of the road work are cleared again and again.

There are various algorithms for model checking the μ-calculus [4, 8, 17]. For efficiency reasons we employ a top-down strategy.

```
1  EMFeRGame game = new EMFeRGame()
2  .withOpponentTrafo("move car",
3        root->getCars(root),
4        (root, car) -> moveCar(root, car))
5  .withOpponentTrafo("create east",
6        root->createCarGoingEast(root))
7  .withOpponentTrafo("remove east",
8        root->removeCarGoingEast(root))
9  .withOpponentTrafo("create west",
10       root->createCarGoingWest(root))
11 .withOpponentTrafo("remove west",
12       root->removeCarGoingWest(root))
13 .withMyTrafo("swap green green",
14       root -> swapGreenGreen(root))
15 .withMyTrafo("swap green red",
16       root -> swapGreenRed(root))
17 .withMyTrafo("swap red green",
18       root -> swapRedGreen(root))
19 .withMyTrafo("swap red red",
20       root -> swapRedRed(root))
21 .withMuCondition("noLeftCar",
22       root -> ! isCarAt(root, WEST))
23 .withMuCondition("noRightCar",
24       root -> ! isCarAt(root, EAST))
25 .withGeneralCondition("safe",
26       root -> ! isCarDeadLock(root))
27 .withStart(roadMap);
28
29 int size = game.explore();
```

Listing 10: synthetiControl.run()

During the construction of our reachability graph, we check the basic conditions `isCarAt`. Whenever a condition `isCarAt`(s, x) holds for some state s and $x \in \{\texttt{WEST}, \texttt{EAST}\}$, we add a flag x to s. These flags are forwarded to subsequent states. Thus these flags indicate that on the path leading to the current state one such condition has been fulfilled. As soon as we reach a state where both conditions have been flagged, we know that there is a path that fulfills all our requirements. Once both have been fulfilled, for successor states we reset all flags and the game starts anew, which corresponds to the computation of a greatest fixpoint P'. Finally, for each state in our reachability graph we compute the shortest path to a state that fulfills all conditions. Now we are able to generate a game based controller that in each step chooses the alternative that has the shortest distance to a winning state.

With the help of this game-theoretic interpretation of controller synthesis, in our example the simple conditions $\neg$`isCarAt`$(s, \texttt{WEST})$ and $\neg$`isCarAt`$(s, \texttt{EAST})$ suffice to achieve a game based signal controller that guarantees fairness for all cars, cf. Listing 10 lines 21 to 24. Generally, the game controller and its μ conditions allow to specify simple liveness conditions, i.e. a set of states / conditions that does not need to hold in every state but that shall be achieved again and again. In contrast to a simple `alwaysFinally` condition, the game controller enforces the conditions for all possible behaviors of the simulated environment.

The EMFeRGame creates 67577 reachable states which is roughly 2.2 times the size of the metric based reachability graph. This is a result of the additional flags that indicate which μ condition holds in which area of the reachability graph. Using EMFeR to test the game based controller results in 18631 reachable states. This is caused by the two flags for our two μ conditions and by the much simpler conditions that allow much more situations to occur. For example, the metric based controller never allows both signals to show green while the game based controller switches to red very late, e.g. just when one car has entered the road work area and there is another car at the opposite signal.

6 CONCLUSION

EMFeR provides reachability graph computation for (EMF) based models based on model transformations provided as simple Java lambdas. Thus, you can do complete testing and model checking and controller synthesis on your genuine model using your genuine model transformations. Ideally, you may test your productive code, exhaustively. There is no need to recode your problem e.g. in Promela or in any other proprietary language used by current model checkers. EMFeR provides full CTL logic and in addition some special modal μ calculus operations. It should be easy to add more μ calculus operations to EMFeR on demand.

Our roadwork example results in fairly large reachability graphs with some 18,000 or 37,000 states for a street with 11 tracks and 2 signals. Actually, a synthetic controller with 2,500 or 18,000 states is quite big for the control of just 2 signals. The reason is that the generated controller observes all car movements on the whole road. We could shrink our road to e.g. 3 tracks, one for each entry to the road work area and one for the roadwork area itself. On each track there might be a car or not resulting in 2^3 possible states and there are two signals which are either red or green giving 2^2 states. This would thus result in less than 33 reachable states. The metric based controller would add a `lastTravelDirection` bit resulting in up to 64 states. The game based controller employs two mu conditions that result in two flags passed along the game states, i.e. up to $2^{3+2+2} = 128$ states.

In the other direction, we could enlarge our road with additional tracks and additional signals for e.g. a crossing. We have run cases with some ten million states, i.e. about 2^{23} states. Each additional track would add a factor of two to the upper bound of states, thus we could go to a street with about 20 tracks or a crossing with 10 incoming lanes, roughly double the size of our current example. There is hope, that EMFeR performance can be further enhanced by techniques developed e.g. in the context of the Spin model checker. Still the size of manageable object models is quite limited. For larger problems model abstractions are necessary. Systematic approaches to model abstraction like the counter example guided abstraction refinement method [5] are future work.

However, EMFeR allows to do full testing and model checking on dynamic object models with model transformations implemented in your favorite transformation language or in the language used for the actual implementation. The key contribution are efficient

hashing mechanisms for models and isomorphism checking. These techniques basically rely on reflective access to the models as provided by EMF's `EClass`. We need to be able to ask a model element for its attributes and to read and write those attributes. Using java.lang.reflect we could achieve this reflective access for general Java objects. Thus, our current work is to generalize EMFeR for other modeling frameworks and for POJO models.

REFERENCES

[1] Alchemy.js 2018. Alchemy.js - A graph visualization application for the web. http://graphalchemist.github.io/Alchemy/.

[2] ATL 2018. ATL Transformation Language. http://www.eclipse.org/atl/.

[3] Paolo Baldan, Andrea Corradini, and Barbara König. 2004. Verifying finite-state graph grammars: an unfolding-based approach. In *International Conference on Concurrency Theory*. Springer, 83–98.

[4] G. Bhat and R. Cleaveland. 1996. Efficient Local Model-Checking for Fragments of the Modal μ-Calculus. In *Proc. 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS'96 (LNCS)*, T. Margaria and B. Steffen (Eds.), Vol. 1055. Springer, 107–126.

[5] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

[6] Christoph Eickhoff, Lennert Raesch, and Albert Zündorf. 2016. The SDMLib Solution to the Class Responsibility Assignment Case for TTC2016.. In *TTC@ STAF*. 27–32.

[7] E. A. Emerson and E. M. Clarke. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 2, 3 (1982), 241–266.

[8] E. A. Emerson and C. L. Lei. 1986. Efficient Model Checking in Fragments of the Propositional μ–Calculus. In *Symposion on Logic in Computer Science*. IEEE, Washington, D.C., USA, 267–278.

[9] emf 2018. Eclipse Modeling Framework. https://www.eclipse.org/modeling/emf/.

[10] emferWebSite 2018. EMFeR Github Site. https://github.com/fujaba/EMFeR.

[11] Joel Greenyer, Daniel Gritzner, Timo Gutjahr, Tim Duente, Stefan Dulle, Falk-David Deppe, Nils Glade, Marius Hilbich, Florian Koenig, Jannis Luennemann, et al. 2015. Scenarios@ run. time-Distributed Execution of Specifications on IoT-connected Robots.. In *MoDELS@ Run. time*. 71–80.

[12] grooveWebSite 2018. Groove Web Site. http://groove.cs.utwente.nl/.

[13] HenshinWebSite 2018. Henshin Web Site. https://www.eclipse.org/henshin/.

[14] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[15] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290. https://doi.org/10.1145/505145.505149

[16] D. Kozen. 1983. Results on the Propositional μ-calculus. *TCS* 27 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6

[17] P. Stevens and C. Stirling. 1998. Practical Model-Checking using Games. In *Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98 (LNCS)*, B. Steffen (Ed.), Vol. 1384. Springer, 85–101.

[18] C. Stirling. 1995. Local Model Checking Games. In *Proc. 6th Conf. on Concurrency Theory, CONCUR'95 (LNCS)*, Vol. 962. Springer, 1–11. https://doi.org/10.1007/3-540-60218-6_1

[19] wwwSDMLib 2018. Story Driven Modeling Library. http://sdmlib.org/.

[20] Xcore 2018. Xcore. https://wiki.eclipse.org/Xcore.

[21] xtend 2018. Xtend. https://www.eclipse.org/xtend/.