Modeling Automated Highway Systems with VeriJ

Yan Zhang

Modeling Automated Highway Systems with VeriJ

ZHANG Yan
Université Pierre & Marie Curie, CNRS-UMR7606 (LIP6/MoVe)
Paris, France
yan.zhang@lip6.fr

Abstract

This paper presents VeriJ, a domain specific language (DSL) which consists of a subset of the Java language, with the aim to help engineers to easily build the state space encoding of a complex system. This language will allow to apply existing verification and control techniques based on a hierarchical variant of Decision Diagrams. The example of an automated highway system is used to present the basic constructs of the language.

1 Introduction

Modeling complex systems for verification purposes is usually a difficult task, and domain experts do not wish to handle low level models or complex specification formalisms. The area of automated highway systems with respect to microscopic traffic flow, is a significant example [10]. Several modeling and verification attempts have been made on such systems with Petri nets [4, 1, 3], transition systems [2], cellular automata models [7], behavioral models [6] or linear parameter varying models [11]. In these studies, system complexity and variable number of vehicles make modeling and state space generation a painful step in the verification process.

On the other hand, widely known general-purpose languages, like Java, can be used to build a system description using powerful and mature Integrated Development Environments (IDEs) such as Eclipse. There exists a powerful model checker, Java Path Finder (JPF) [5], for Java programs. JPF implements a backtrackable Java Virtual Machine (JVM) to provide non-deterministic choices and control over thread scheduling. But JPF does not scale up due to its internal state representation and to the large scope of Java.

To alleviate this problem, an approach based on limited programs to establish a formal specification could be proposed, inspired by Domain Specific Languages (DSL) [12, 8] that are dedicated to a particular domain and express a small set of adapted solutions within this restricted scope. Recent developments in this direction show that DSL, thanks to this limited scope, lead to more efficient solutions than is possible in more general programming languages (like JPF).

In this work, we propose to combine the advantages of the general-purpose language Java and domain specific languages to define VeriJ, a DSL which consists of a Java fragment dedicated to formal verification. This language will be used to create a model for specific domains. Automatic encoding of such a model into an efficient structure based on decision diagrams will allow domain experts to apply any verification technique available for this structure, for instance LTL or CTL model checking, or control synthesis with the Ramadge and Wonham framework [9]. Also, VeriJ provides an easy way to simulate the model. We first describe an example of highway system, then we present VeriJ through this example and conclude with future work.

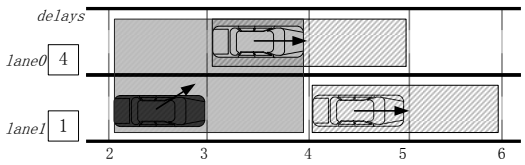

Figure 1: Highway

2 A Highway system

We consider a $1km$ highway section similar to the one in [2], with the aim to control vehicles and to avoid collisions. A small part of such a section is depicted in Fig. 1 (shadowed rectangles are explained later). This section is modeled as a discrete system that consists of a set of vehicles, moving on n lanes of length L, numbered from 0 to $n-1$. Moves of a vehicle include driving forward and changing lanes. And vehicles exiting the range of the section are deleted from the set.

To be consistent with our future purpose of verification and control, we consider a labeled transition system with at most one *uncontrollable* vehicle operated by the *environment* (dark colored in Fig. 1), the rest of the vehicles being *controllable* by the controller. With two (or more) uncontrollable vehicles, the environment could always produce a crash.

Each vehicle is defined as a tuple $(xpos, ypos, xspeed, yspeed, isControlled)$, where the value $xpos$ denotes relative position with respect to its predecessor in the horizontal direction and $ypos$ is the number of the lane. Integer value $xspeed$ denotes the horizontal speed, non-deterministically chosen within a fixed interval $[speed_{min}, speed_{max}]$, and $yspeed$ is the value of vertical speed, non-deterministically chosen in $\{-1,0,1\}$, allowing a vehicle to move in adjacent lanes (-1 or 1) or stay in the current lane (value 0). The boolean label *isControlled* indicates the status of the vehicle with respect to the controller. The speed of those vehicles can be modified and new vehicle arrivals are constrained by setting a lower bound d_{min} for a *delay* parameter, which represents the time elapsed between two successive arrivals on a given lane. Crashes are detected by estimating overlaps of *dangerZone*s, which are the areas covered from the current to the next step, shown as shadowed rectangles in Fig. 1. If there exists a crash, the configuration is regarded as *bad*, and given a specific state label.

Several standard constructs are needed to implement the description above: *for each* is needed to update delays for each lane or move each vehicle in the set, *exists* is used for predicates indicating a collision, and *non-deterministic choice* has to be applied in adapting speed, adding new vehicles and setting the uncontrolled vehicle. We now show how to build a DSL interface for highway systems in Java.

3 A DSL interface for the highway system

As mentioned in section 1, VeriJ chooses a restricted part of Java. For instance, VeriJ includes basic data types, arithmetic operators, assignments, decision and control statements, public or private access modifiers, construction of classes with instantiation, and standard instructions from logic. VeriJ does not support the features such as libraries or native code.

The VeriJ specification of the highway system described above is illustrated in Fig. 2. It contains four classes: **Highway**, **Delays**, **Vehicle** and **VehiclesSet**. We introduce the different classes and emphasize several difficult operations with corresponding VeriJ code for explanation.

Highway. This class has a *vehicles* typed **VehiclesSet** and a *delays* typed **Delays**. The main operation consists of a *transition* from the current state to the next state. This transition contains five steps shown in the code below.

```
void transition() {
        vehicles.moveEachVehicle(); //apply speed
        vehicles.reorderVehicles(); //to be described in class VehiclesSet
        playEnvironment();
        playController();
        delays.updateDelay();
}
```

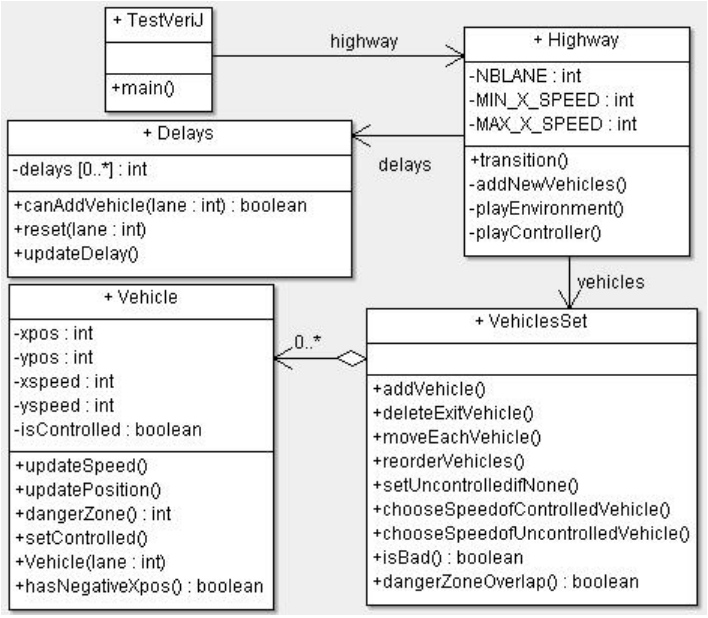

Figure 2: Class Diagram describing the VeriJ model of Highway Systems

In this function, *playController* only control the speed of controlled vehicles, the other actions are all performed by *playEnvironment*, described as below.

```
private void playEnvironment() {
        addNewVehicles();
        vehicles.deleteExitVehicle();
        vehicles.setUncontrolledIfNone();
        vehicles.chooseSpeedOfUncontrolledVehicle();
}
private void playController() {
        vehicles.chooseSpeedOfControlledVehicles();
}
```

The synchronization of *delays* and *vehicles* lies in *addNewVehicles()* with the non-deterministic operation *NDChoice()*, a random boolean generator. It allows to specify free choice semantics of a system and will be used to build the different target states of the model.

```
private void addNewVehicles() {
    for(int lane=0; lane<NBLane;lane++){
        if (delays.canAddVehicle(lane) && VeriJ.NDChoice("add vehicle?")) {
            vehicles.addVehicle(lane);
            delays.reset(lane);
        }
```

```
    }
}
```

Delays. This class uses a Java data type "array" to store "integer" data *delay* on each lane. For an integer counter $j \in \{0,1,...,n-1\}$, a vehicle may be added on j^{th} lane if $delays[j] > d_{min}$. At the moment a vehicle enters this lane, $delays[j]$ is reset to 0.

Operations in this class are thus *reset* and *updateDelay*, where a loop statement is used to update time *for each* lane. Predicate *canAddVehicle* allows synchronization with the vehicle set.

Vehicle. Vehicles using relative (instead of absolute) positions (see Section 2) for the horizontal coordinate will lead to a more compact representation of the state space and make the collision detection easier. Let i denote the i^{th} vehicle in the list, a horizontal move from time k to time $k+1$ is obtained by:

$$xpos_i(k+1) = xpos_i(k) + xspeed_i(k) - xspeed_{i-1}(k) \tag{1}$$

Hence, class **Vehicle** has operations *updatePosition* according to the formula above, *updateSpeed*, where non-deterministic *xspeed* is given in terms of a random value, and *setControlled* to set a vehicle as a uncontrolled one. It also contains a *dangerZone* defined by horizontal and vertical speeds.

The tricky point for this class will be the constructor. When class **Vehicle** is instantiated by calling *Vehicle(int lane)*, a new vehicle is added on the lane corresponding to the parameter.

```
public Vehicle(int lane) {
    xpos = 0;
    ypos = lane;
    xspeed = VeriJ.random(MIN_X_SPEED, MAX_X_SPEED);
    yspeed = 0;
    isControlled = true;
}
```

VehiclesSet. A vehicle set is a list of vehicles, described by a "list" with type **Vehicle**. The size of this list is variable to allow adding new vehicles and deleting exiting vehicles.

Vehicles are added to the list one by one from the beginning. Thus, the list has an order. Since vehicles have different speeds, it is possible that the relative position $xpos_i$ becomes negative in the update operation, when overtaking occurs. In this case, the list must be reordered. It also suggests a risk of collision if *dangerZone*s overlap with each other. Predicate *isBad* returns a boolean value labeling a configuration where a collision occurs. If there is no uncontrolled vehicle in the section, a nondeterministic operation may be applied to set (at most) one. Class **VehiclesSet** includes operations *addVehicle*, *deleteExitVehicle*, *moveEachVehicle*, *reorderVehicles*, *chooseSpeed*, *setUncontrolledifNone* and a predicate *isBad*.

The operation *reorderVehicles* is a difficult point in this class. It must carry out two actions, changing order in the list and assigning correct values to relative position *xpos*. To take into account the case where a high-speed vehicle overtakes more than one vehicle, we introduce the predecessor and successor for each vehicle with negative *xpos*. A for loop with index is used to apply reorder to each vehicle.

```
public void reorderVehicles() {
    for(i=0; i<vehicles.size(); i++){
        Vehicle vehicle = vehicles.get(i);
        if (vehicle.hasNegativeXpos()){
            //set predecessor
            Vehicle vpred = vehicles.get(i-1);
```

```
            //record current negative xpos
            int currentpos = vehicle.xpos;
            //update xpos of current vehicle
            vehicle.xpos = vehicle.xpos + vpred.xpos;
            //update xpos of predecessor
            vpred.xpos = - currentpos;
            //set successor
            Vehicle vsucc = vehicles.get(i+1);
            //update xpos of successor
            vsucc.xpos = vehicle.xpos + currentpos;
            //swap the order of current vehicle and predecessor
            vehicles.set(i, vpred);
            vehicles.set(i-1, vehicle);
        }
    }
}
```

The configuration label *bad* is determined by the answer to ”does a vehicle set contains at least an item with the crash property?”. This labeling is performed by the *exists* predicate, which returns true if predicate *dangerZoneOverlap()* returns true for at least one item in the iterable object *vehicles*.

```
public boolean isBad(){
return VeriJ.exists(dangerZoneOverlap(), vehicles);
}
```

The syntax of VeriJ was explained on the highway system and will then be used to automatically generate a model for state space exploration. In the same way, VeriJ can be used on other engineering specifications to handle formal verification.

4 Simulation

In this section, we apply VeriJ to execute a highway system, running on top of a host JVM. Three implementation modes are designed for particular purposes.

- Random Mode: This mode randomly generates a value for each non-deterministic situation;
- Keyboard Mode: It allows engineers to set up their own scenario for the system by assigning the non-deterministic choice an expected value from keyboard; a scaled number of vehicles can be added on the highway section;
- Trace Mode: In this mode, a recorder is built to keep the values set in the other two modes to offer a means to observe the system.

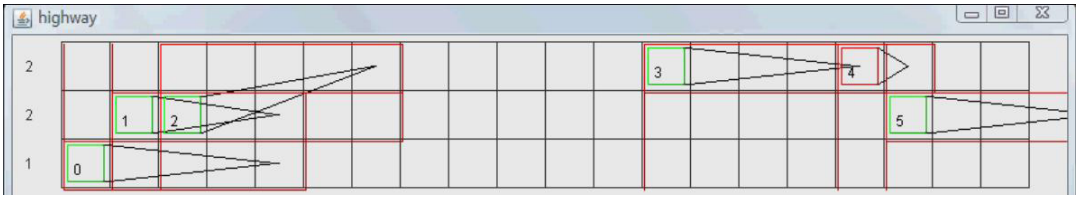

Figure 3: Highway Execution

Fig. 3 shows a configuration of an execution in a random mode. In this diagram, each grid represents a position one vehicle may occupy, with the correct order for vehicles and *vehicle*4 labeled as uncontrollable. Overlaps exist between *dangerZone*s of *vehicle*1 and *vehicle*2 and between *vehicle*3 and *vehicle*4, hence producing a bad configuration.

This example shows how VeriJ benefits from standard Java execution, while also offering a nice basis to build a complete representation for formal analysis.

5 Conclusion

This paper presents VeriJ, a DSL for modeling, simulation and verification of complex systems. The next step will be to translate this fragment in a framework based on Decision Diagrams, allowing to provide a compact encoding of the state space for a system described in VeriJ. Verification techniques (like LTL or CTL model checking) and control synthesis will then be directly applied on the translation.

References

[1] A. Aitouche and S. Hayat. Multiagent model using coloured petri nets for the regulation traffic of an automated highway. *Journal of Intelligent Transportation Systems, IEEE*, 1:37–42, 2003.

[2] B. Bérard, S. Haddad, L. Hillah, F. Kordon, and Y. Thierry-Mieg. Collision Avoidance in Intelligent Transport Systems: towards an Application of Control Theory. In *Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08)*, pages 346–351, Göteborg, Sweden, May 2008. IEEE Press.

[3] F. Bonnefoi, L. Hillah, F. Kordon, and G. Frémont. An approach to model variations of a scenario: Application to intelligent transport systems. In *Fourth International Workshop on Modelling of Objects, Components and Agents (MOCA 'O6)*, pages 65–86, Universität Hamburg, 2006.

[4] I. Demongodin. Modeling and analysis of transportation networks using batches petri nets with controllable batch speed. In *PETRI NETS '09: Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, pages 204–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] T. Gvero, M. Gligoric, S. Lauterburg, M. d'Amorim, D. Marinov, and S. Khurshid. State extensions for Java PathFinder. In *Proceedings of the 30th international conference on Software Engineering (ICSE'08)*, pages 863–866, New York, NY, USA, 2008. ACM.

[6] H. Jochen. New Behavioral Model for Microscopic Freeway Traffic-Flow Simulation. *Transportation Research Record: Journal of the Transportation Research Board*, 2088:10–17, 2008.

[7] W. Knospe, L. Santen, A. Schadschneider, and M. Schreckenberg. Towards a realistic microscopic description of highway traffic. *Journal of Physics A: Mathematical and General*, 33(48):L477–L485, 2000.

[8] F. Maraninchi and Y. Rémond. Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems. *Science of Computer Programming*, 46:219–254, 2003.

[9] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete-Event Processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[10] S. E. Shladover. Modelling and control issues for automated highway systems. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 215(4):335–343, 2001.

[11] L. Tamás, K. Balázs, V. István, and J. Bokora. Parameter-dependent modeling of freeway traffic flow. *Transportation Research*, 18(4):471–488, Aug. 2010.

[12] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, LNCS, pages 291–373. Springer, 2008.