Semi-Automatic Controller Design of Java-like Models

Yan Zhang[*] Béatrice Bérard

Lom Messan Hillah Yann Thierry-Mieg

{yan.zhang, beatrice.berard, lom-messan.hillah, yann.thierry-mieg}@lip6.fr

LIP6/MoVe, Université Pierre & Marie Curie, Paris, France

ABSTRACT

Controller synthesis consists in automatically generating a controller to restrict a hardware or software system so that it respects given requirements, for instance safety properties. Existing synthesis tools for discrete event systems mainly solve the problem for systems described in low-level formalisms.

Controller synthesis, however, is not used in most industrial engineering processes. Barriers to wider adoption are the complexity of formally expressing the system and its requirements, the state explosion induced by large systems, and the limited confidence in the result, due to the difficulty in understanding the generated code.

We propose an iterative, incremental, and semi-automatic approach to controller design, supporting the engineering process and mitigating state space explosion during synthesis. To provide a high-level environment, our approach is implemented in VeriJ, a Java-like language, and illustrated on a significant example taken from automated transport systems.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*CASE*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*

General Terms

Languages, Design, Verification

Keywords

controller synthesis, software engineering, automated transport systems, Java-like models

1. INTRODUCTION

Context. Controllability of discrete event systems (DES), introduced in the seminal work of Ramadge and Wonham [18], answers the following question: given a model M of a system and a specification, does there exist a controller $\mathcal{C}$ such that the composition of $\mathcal{C}$ and M satisfies the specification? If the answer is positive, synthesis attempts to automatically build a controller.

This problem has been largely studied for transition systems and several tools exist that take as input state machines or Petri nets. Although the theory has been available for more than twenty years, these tools are not widely integrated in industrial software engineering processes. The reasons are the following:

- Formally expressing the complete system and its requirements in low-level formal models is often very costly,
- Large systems produce a combinatorial explosion of the state space, which is an obstacle for scalability,
- The code generated for the controller has size linear to the state space, and reducing its size is NP-hard. Therefore it is difficult for system experts to interpret.

Hence industrial experts usually manually design controllers. Because manual solutions are error-prone, controllers then need to be extensively tested for correctness. The most widely used approach for this is simulation.

Contributions. We propose to combine the fully automatic synthesis with a user-centric design. Our main contribution is the definition of an iterative, incremental and semi-automatic approach for controller design. In addition to the usual engineering process (testing, simulation…), controllability checking is used to verify the correctness of user-defined strategies. Besides, we also use automatic (partial) controller synthesis in the design process. Iteratively, based on the feedback of the partial strategy, users can successively refine the system or the controller.

Designers can model their systems using VeriJ [19], a Java-like language. In this paper we introduce controller synthesis in the VeriJ framework. Since VeriJ is source compatible with Java, all operations in the engineering process can be directly run and tested within usual IDEs (integrated development environments) at any point of the process. By letting the system designer provide strategies and by targeting assistance for the designer rather than fully automatic synthesis we partly avoid scalability issues.

Outline. Section 2 briefly describes supervisory control theory and the usual process of industrial controller design. In Section 3 we present the techniques related to the semi-automatic controller design (SACD), while the instantiation with VeriJ is detailed in Section 4. We illustrate this approach in Section 5 on a significant example taken from automated transport systems and we discuss related work in Section 6.

[*]This work has been supported by a Ph. D. grant from the Chinese Scholarship Council.

2. BACKGROUND

2.1 Supervisory Control Theory

The controllability problem can be viewed as a two-player turn-based game, *environment* versus *controller*, where the controller tries to enforce the specification, against all possible environment moves. The problem is to find whether there exists a winning strategy for the *controller*. If the answer is positive, controller synthesis tries to automatically construct such a winning strategy. For safety objectives, it is well known that the game is determined (one of the two players has a winning strategy) and a memoryless strategy can be built in linear time w.r.t. the size of the model [6].

Controllability. When the model is a Labelled Transition System (LTS) $\mathcal{T} = (S, s_0, \rightarrow)$ over a set of actions Act (with S the set of states, initial state s_0 and transition relation $\rightarrow \subseteq S \times Act \times S$), a safety specification requires avoidance of a subset S_{fail} of the set S_{reach} of reachable states. The controllability algorithm is a backward exploration, starting from S_{fail}: 1) any state from which the *environment* can reach a failure state in a single move is added to S_{fail} and 2) any state from which all *controller* moves lead to a failure state is added to S_{fail}. When a fixpoint is reached, either the initial state is a failure state, hence the system is not controllable, or there exists a winning strategy for the *controller*.

Synthesis. After the controllability test, the set S_{reach} is partitioned into states in S_{fail} and states in $S_{safe} = S_{reach} \setminus S_{fail}$. Let S_c be the subset of states where the *controller* can act, and Act_c be the possible controller actions. A strategy is a function $f : S_c \rightarrow 2^{Act_c}$ which maps each state in S_c to a subset of Act_c. A winning strategy selects actions ensuring that all system executions stay in S_{safe}.

Control in practice. If we consider controllability and synthesis tools as black boxes, controllability takes as input a model containing a system, a safety specification, and a (partially) defined controller. The output of the controllability check can be:
• system is *controlled*, there are no reachable failure states, hence any strategy of a controller is a winning strategy;
• system is *controllable*, there are reachable failure states, but there exist winning strategies for the controller;
• system is *uncontrollable*, the tool can exhibit a winning strategy for the environment.

Controller synthesis takes as input a controllable model produced by the previous check, and outputs a controlled model including a winning controller strategy.

2.2 Manual Steps in Controller Design

A controller design expert manipulates models as follows:
1) Build a model based on his expertise; the model includes the description of a system, a specification and a (partial) control strategy. It is usually defined by successive refinements.
2) Interpret diagnosis provided by tools to improve or correct the model.

Modeling is essential and cannot be fully automated. Correct interpretation of diagnosis also requires human insight. Most industrial controller design processes only use tests and simulations to provide feedback to the user. The strength of having a human design the controller is that the state space size is no longer a limiting issue. Moreover, human designed controllers can be reused for different parameters of a system. They can be small because they often include additional variables that allow a simple definition of the controller strategy.

Human design however is error-prone, so quality control tools must be provided. For instance, testing returns a diagnosis in case of errors that helps designers debug.

To formalize these notions, we consider quality control tools as black boxes, that take as input a model and can output:
• an error diagnosis, if problems are identified;
• otherwise, a metric M that defines a measure of confidence, such that higher values are better.

Differences between values of M can only be interpreted to compare variants of a given model. The maximum value of M (denoting 100% confidence) is usually a priori unknown. For instance, a metric for a test suite could be the number of successful tests. If a test fails, the user obtains a useful diagnosis. Testing increases confidence in the controller but cannot prove correctness.

3. SEMI-AUTOMATIC CONTROLLER DESIGN (SACD)

Fully automatic synthesis promises to automatically build a controller hence replacing the human designer. Synthesis tools however cannot cope with large state spaces, whereas human designers can often abstract parts of the system and guess a winning strategy.

To combine the strengths of both automatic synthesis and user-centric design, we first propose the concepts of partial controllability checking and partial synthesis, and then develop an iterative, incremental, semi-automatic approach.

3.1 Partial Control

We proceed by iterative refinement of a partially designed controller, progressively decreasing non-determinism in the model, which reduces the state space size. Hence controllability checking that was unfeasible at early stages of a controller design, may become possible at the final stages.

Partial controllability. To obtain useful results even when state space exploration is incomplete, we consider an additional possible output for the controllability check: *partially controllable* with confidence M. If the tool succeeds in complete exploration, it outputs a categorical answer (system is controlled, controllable or uncontrollable). Control theory tools can then be used in the controller design process just like other quality-control tools that provide a metric. Partial exploration with uncontrollable result produces valid counterexamples for the full system.

To instantiate this metric, we define M as a bound on the depth of a breadth-first exploration of the state space. Note that tools can usually dynamically detect if M is greater than the state space depth, indicating that the exploration is complete. They can also dynamically interrupt computation when memory limits are reached, and return the current depth.

With this definition, useful results can be obtained, even when exploration is incomplete. Partial controllability increases the user's confidence in the quality of the controller. If for some depth M, the system is not controllable, then a winning strategy for the environment in less than M steps can be obtained.

Partial synthesis. We propose two methods for partial synthesis. The first one is based on the previous metric: When a system is partially controllable with confidence M, synthesis based on the explored state space can be attempted. Such synthesis will build a control strategy allowing the controller to play at least $M' \leq M$ moves without loosing. Note that, this strategy may be a loosing strategy for $M' > M$.

Even partial strategies can eliminate possible failure states from explored states. Hence in general, we can reduce the size of the state space by iteratively implementing partial synthesis and increasing the metric M up to full exploration. If a solution can be found by this procedure, it is sound. However, if the procedure fails, there is no guarantee that no solution exists.

The second method, which can be combined with the first, consists in synthesizing non deterministic controllers. Such a partial strategy provides deterministic choices only for a subset of system states. In practice, the user provides a boolean predicate over states and asks for partial synthesis only in states satisfying this predicate. This predicate can describe for instance critical situations where correctness is essential.

3.2 SACD Process

Our iterative controller design process is shown in Fig. 1.

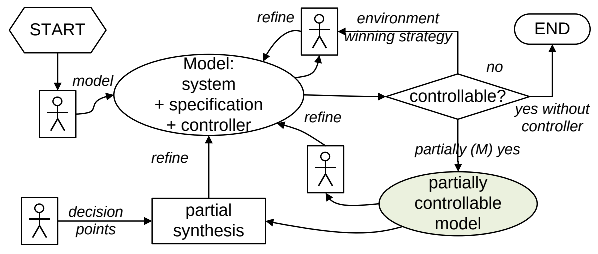

Figure 1: Semi-automatic Controller Design.

The first step consists in manually building a model, called initial state, containing a system, a specification, and a (typically non deterministic) controller. The user can refine the model with usual quality-control techniques, for instance, simulation and debugging (not represented in the figure).

At each step of the process, the user can first perform the controllability check on the model. If the test is unsuccessful, the user should interpret the diagnosis to correct the model. One way to do this, as already proposed in UPPAAL-Tiga [2], is to synthesize a counterexample and let the user interactively play against it.

For a partially controllable model (shaded node in the figure), the user can ask for automatic partial synthesis. Any modification of the model makes the process go to its initial state. If a controller is fully defined, the controllability check proves controller correctness.

4. INSTANTIATION OF SACD WITH VERIJ

We apply the SACD process on the Java-like modeling language VeriJ [19]. We first briefly present VeriJ and then show how the successive steps are instantiated in this framework.

4.1 VeriJ

VeriJ contains a core subset of Java and includes in addition several constructs dedicated to control (see Fig. 2) described below. Java implementations of these concepts are also provided to let VeriJ be source-compatible with Java.

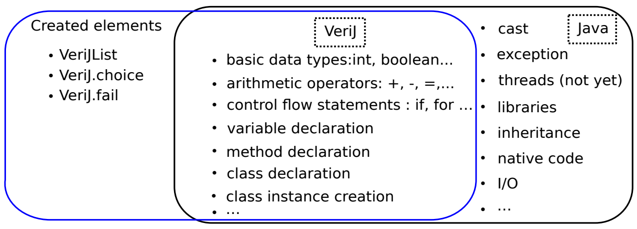

Figure 2: Relation between VeriJ and Java.

From the metamodel perspective, VeriJ comprises the follows:
• 55 metaclasses that are shared with the Java metamodel representing instructions, class declarations, etc. It is based on the Java metamodel provided by MoDisco[1] that includes 126 metaclasses.
• A metaclass `VeriJList` to support basic Java collections framework with the essential array operations (e.g. `get(index), add, set, remove`, etc.). This specific construct avoids the complexity of processing implementation of Java collections. A VeriJ Map could also be implemented in the future.
• A metaclass to model the `choice` operation, which will induce decision points corresponding to player moves. It carries as properties the player identifier (controller or environment), the label of the choice (for instance `Choose next move`), and the possible choices of actions (a list of integers representing moves).
• Another metaclass `fail` to label initial failure states.

VeriJ programs are analyzed using a transformation into LTS [19]. An LTS state is composed of the valuations of the program variables (heap + stack). Transitions correspond to instructions but, thanks to a dedicated abstraction, only states at decision points are retained in the LTS. More technical information about VeriJ and full code for the example are available[2].

4.2 VeriJ in SACD

VeriJ is designed to benefit from the expressiveness and simplicity of Java to model complex systems and to carry the information required in supervisory control. Users can *1)* easily model the system, specification and controller in a Java-like program, *2)* use quality-control techniques and tools of mature IDEs, *3)* use partial controllability and synthesis as described above. All these properties make VeriJ a good choice to support the SACD process.

Modeling. The first step consists in modeling the system by:
• Building a set of VeriJ classes that represent relevant aspects of the system, based on the business domain metamodel. For instance, an application in finance might include the classes `Portfolio, Account, MarketData`, etc. Then users need to describe the behavior of these classes using standard Java syntax.
• Constructing the game structure using a possibly infinite loop containing player moves (see the `while (true)` in Fig. 5). Player moves are modeled as calls to a `choice` method. For instance, at each time step the market (playing the environment) might choose to emit actions like `OrderExecuted` or `OrderCanceled`. Users can define a (possibly partial) controller through calls to the `choice` operation, with player identifier "controller".
• Specifying the safety objectives as situations users wish to avoid. A call to the `fail` operation indicates an initial failure state.

Simulation and testing. At this stage, thanks to the Java-like syntax, in a mature IDE, users can test the model by running the program (possibly under a debugger) at any point. Calls to `choice` can be delegated to the standard Java `random()` to simply simulate the system, or connected to an interactive trace, where the user is prompted for values at each decision point. Execution traces (sequences of choices) can optionally be saved and replayed (typically under a debugger or as part of a test suite) for further analysis.

Consequently, modeling bugs common to many approaches (particularly when using formal notations) are more easily avoided or patched. Libraries of domain objects can also be easily reused to build other models from the same application domain.

User-defined controllers. The user can design a (partial) control strategy as VeriJ code and check the (partial) controllability of the resulting model. Non determinism is modeled as calls to the `choice` method with a range of values. This strategy is iteratively refined by restricting the range until the model is fully controlled.

[1]www.eclipse.org/MoDisco/
[2]http://pagesperso-systeme.lip6.fr/Yan.Zhang/VeriJ.html

Automatic controller synthesis. As defined in Section 2.1, a strategy is a mapping $f : S_c \to 2^{Act_c}$. Our strategies are defined on the subset of S_c containing only safe states that can lead to unsafe ones, and are non deterministic in other states. Our tool generates VeriJ code that tests program variables to determine the current LTS state. This code is injected as replacement of the calls to `choice`. However, it contains a (huge) boolean expression that is difficult to read and process even by tools. The size of this expression can sometimes be reduced (by heuristics like in [2], since the general problem is NP-hard) but in the worst case, it is linear in the state space size.

5. EXAMPLE: AUTOMATED TRANSPORT CONTROL

In this section, we illustrate our approach on a significant example from the area of automated transport systems. Some work has been devoted to the verification on variants of this system modeled with Petri net [5], transition systems [3], etc.

This example features a variable number of vehicles, stored in a list, as well as complex list operations such as reordering. Most of the previous formalisms require a static bound on number of instances and describing the transition relation is a difficult and error-prone step in the verification process. These difficulties are largely addressed by the high level of description offered by VeriJ.

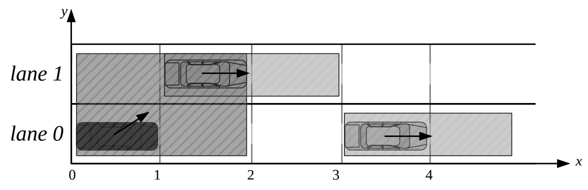

Figure 3: A highway section.

We consider a highway section similar to the one in [3], with the aim to control vehicles and avoid collisions. A small part of such a section is depicted in Fig. 3. 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. New vehicles can be dynamically inserted at left-most position and vehicles exiting the section are deleted from the list. Crashes are detected by estimating overlaps of danger zones (shadowed rectangles in the figure), which are computed based on current vehicle speeds.

5.1 Modeling the Highway System with VeriJ

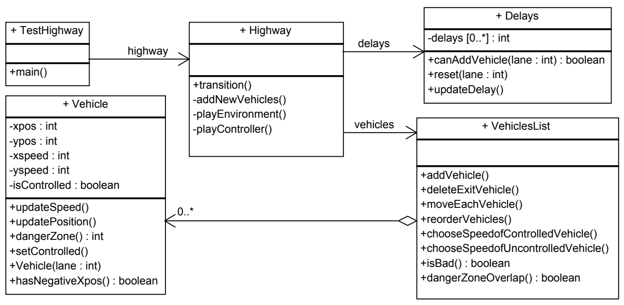

Figure 4: Class diagram for the highway system.

To analyze this system, we first build a set of VeriJ classes, represented in Fig. 4:

- a `Vehicle` has a forward and lateral speed (noted *xspeed* and *yspeed*) as well as coordinates with respect to x and y axes. It also has a boolean property "isControlled" that indicates whether the vehicle is currently controlled.
- The `VehicleList` is a holder for a list of Vehicles, that supports global operations on the list (like adding, deleting and reordering).
- `Delays` represents for each lane the time elapsed after a new inserted vehicle on that lane.
- The `Highway` represents the full system and contains the scaling parameters of the model and the main actions.
- `TestHighway` instantiates the `Highway` and contains the engine of the system.
- `Constants` contains the number of lanes, maximum and minimum *xspeed*, minimum delay d_{min}, etc.

We then define the behavior of the objects, as operations of their corresponding VeriJ classes. Fig. 5 shows code extracts from the example. The main simulation loop is defined in the `main` method. The `transition` method of class Highway first updates the system state, to express that a time step has elapsed. This consists in updating delays, car positions according to their current speed, and sorting the list of cars by distance to the entrance of the highway. Then both players will choose vehicle speeds for the next time step. The controller can set the speed of controlled vehicles (see `playController`). The environment adds new vehicles, deletes exiting vehicles, and can arbitrarily choose one vehicle as out of control and update its speed (see `playEnvironment`).

The operations concerning vehicles are described in class VehicleList and include adding or removing cars, as well as more complex operations such as reordering the list of cars according to their position along the x axis, defining the danger zones and identifying crashes. For each vehicle, the method `updateSpeed` chooses a speed, calling `choice` to perform a non-deterministic choice.

We can simulate the system by simply running the code as a Java program and connect it to a graphical interface for simulation.

Defining scenarios. We studied several characteristic scenarios of the highway:

•[Scenario 1] All vehicles are controllable, the controller updates all vehicle speeds. As mentioned above, the environment *1)* adds at most one vehicle at the entrance of the highway, with random choices for speed and lane, when the time elapsed is greater than d_{min}, and *2)* deletes the vehicles that exit the highway;

•[Scenario 2] Based on scenario 1, there is an immobilized vehicle at a predefined position of the highway;

•[Scenario 3] Based on scenario 1, environment can arbitrarily choose the speed of uncontrolled vehicles (dark colored in Fig. 3). At most one vehicle is uncontrolled, since the environment could trivially cause two uncontrolled vehicles to crash.

•[Scenario 4] Based on scenario 3, the elapsed time of every lane is reset whenever a new vehicle is added on the highway. With scenario 3, the system is reported uncontrollable for more than one lane: two cars are introduced simultaneously, then one becomes uncontrollable and crashes into its neighbor at the next time step. Keeping a minimal safety distance between cars is essential to controllability.

These scenarios share most of their code and defining variations is relatively easy in a programming environment. For example, in scenario 1, we removed the last two method calls from `playEnvironment()`. For scenario 2, we added a method `addUnmovedVehicle()` in class Highway and invoke it in the main function before the simulation loop.

5.2 Controller Design

```
// class TestHighway:
public static void main(...) {
 Highway hw = new Highway();
 while (true)
      hw.transition(); }
// class Highway:
private void playController() {
 vehicles.chooseSpeedOfControlledVehicles(); }
private void playEnvironment() {
 addNewVehicles();
 vehicles.deleteExitVehicle();
 vehicles.setUncontrolledIfNone();
 vehicles.chooseSpeedOfUncontrolledVehicle(); }
```

```
public void transition() {
 delays.updateDelay();
 vehicles.moveCars();
 vehicles.reorderCars();
 playEnvironment();
 playController(); }...
// class Vehicle:
public void updateSpeed(int playerID, int pos) {
 ...
 this.xspeed = VeriJ.choice(minXspeed,maxXspeed,
  playerID,Constants.CHOOSE_X_SPEED);
 this.yspeed = VeriJ.choice(minYspeed,maxYspeed,
  playerID,Constants.CHOOSE_Y_SPEED);}
```

Figure 5: Code samples from the highway system.

Performances (made on a 2.66 GHz, 4GB RAM Linux PC) are shown in Tables 1 and 2. The column "depth" gives the maximal depth reached in the breadth-first exploration. The columns "controllability" show the result of controllability check (*py*: partially yes, $y(w/o)$: yes without controller) with $|A|$ the size of set A. The columns "synthesis" present the performance of automatic synthesis when possible, with NA when Not Available and − when uncontrollable, and $|code|$ giving a measure of the generated controller, expressed in number of boolean connectives in the tests.

We studied the scenarios above (identified in column *sc*), with different values of the parameters d_{min}, n, L. This produces 6 cases, identified by C1 to C6 in column *id*.

Controllability and controller synthesis. From the results in Table 1 we can see that, in scenario 3, the system is reported uncontrollable when the highway contains 2 lanes (C4). After studying some crash scenarios, we deduced that the problem is related to simultaneous insertion of an uncontrolled and controlled vehicle in adjacent lanes, where the uncontrolled vehicle can at the next time step crash into its neighbor. We thus design a scenario 4 that does not allow this behavior. This process corresponds to the upper "refine" step of Fig. 1.

Full automatic controller synthesis. The synthesized controller code for controllable cases is specific to each case, which prevents any reuse of the controller code. For simple instances (C1 and C5), the size of the generated code is still manageable, but for larger instances such as cases C2, C3 and C6, the generated code is huge, complex and unusable in practice.

Semi-automatic controller design. We now manually define several control strategies. Table 2 gives the controllability and synthesis results when using the combination of user-defined controller ($S1$ to $S7$) and automatic synthesis. These strategies are mostly non deterministic but restrict the system by some choices that intuitively seem "safe". A bad choice will yield an uncontrollable system, a good choice will reduce the search while preserving controllability. Each time we successfully built a controllable system, we could add more constraints to the controller code.

Strategy $S1$ is described in Fig.6. In this code, the frontmost car chooses the maximum possible value `maxXspeed` for *xspeed* and 0 for *yspeed*. Intuitively, since no vehicles are in front, it cannot crash in front, and cannot be caught up by another car. Other cars are assigned arbitrary speeds. Hence this controller is still non-deterministic.

We iteratively added strategies $S2$ to $S5$ to the strategy $S1$.

• Strategy $S2$ limits maximum *xspeed*: when f is in the same lane as c, then c should adopt a speed inferior to the distance between them;

• Strategy $S3$ requires c to change lane if f is uncontrolled and is in the same lane as c, but not to change lane if f is uncontrolled and is in a different lane from c;

```
// size: number of cars,
// posCar: index of current car
// actionLabel: which speed (x or y) to be updated
if (posCar == size - 1){
 if (actionLabel == Constants.CHOOSE_X_SPEED)
  return maxXspeed;
 else if (actionLabel == Constants.CHOOSE_Y_SPEED)
  return 0; }
```

Figure 6: Deterministic part of strategy S1

• Strategy $S4$ sets the $c.xspeed$ to $f.xspeed$ if f is controlled and in the same lane;

• Strategy $S5$ forbids lane changes, when f is controlled and is far from c (it cannot be reached in one time step).

A bad choice in one of these strategies would lead to an uncontrollable result and would be reverted. In C3, the search space reduces as the controller becomes more restrictive, so does the size of supervisor code. After combination of strategies $S1$ to $S5$, we apply automatic synthesis for the remaining decisions, which validates these strategies.

Similarly, we iteratively design a controller for C6. We start with the controller elaborated for C3, directly reusing strategies $S1$ to $S5$. This yields a controllable system, with a state space that can now be fully explored. However, since the system is more complex than C3, we need further constraints and define additional strategies $S6$ and $S7$. We defined $S6$ that accelerates or slows down c so that it tries to match $f.xspeed$ when f is controlled, but slows down c to its minimum speed if it is following an uncontrolled vehicle. Constraint $S7$ tries to bring a vehicle *xspeed* to the middle of its possible speed range, if this does not violate other constraints. The controller including $S1$ to $S7$ is completed by synthesis to reach a solution for C6.

Cases C3 and C6 show that the SACD approach helps design readable controllers with proof of correctness. For large parameter values, this method is better suited than fully automatic synthesis where the generated code is huge and difficult to interpret.

6. RELATED WORK

Most of the work, devoted to practical aspects and implementation of controller synthesis, uses inputs specified as Petri nets or automata, and specifications described as temporal logic formulas, with a focus on performance rather than ease of use [4, 8].

To tackle the modeling of complex systems, some tools support the use of scripts to generate graphs. For example, the luafaudes script is integrated to the graphical interface DESTool and linked to the back-end engine libFAUDES [16], an C++ software library for DES synthesis. Supremica [1], provides parameterization and instantiation for modeling. It has a graphical front end and supports

Table 1: Controllability checking (no controller), and synthesis

cases					depth	controllability				synthesis		
id	sc	d_{min}	L	n		$time(s)$	$\|S_{reach}\|$	$\|S_{fail}\|$	$isCtr$	$time(s)$	$\|S_{reach}\|$	$\|code\|$
C1	**1**	2	10	1	Full(41)	4.0	3939	70	*yes*	4.5	3101	503
C2		2	20	1	61	547	727077	6652	*py*	NA	NA	57568
C3	**2**	4	10	2	Full(50)	1047	366615	16334	*yes*	NA	NA	82496
C4	**3**	3	10	2	30	9.1	12883	144	*no*	-	-	-
C5	**4**	3	10	2	Full(62)	20.2	32213	18	*yes*	22.3	32172	134
C6		3	15	2	56	221.1	652582	700	*py*	NA	NA	11420

Table 2: System under user-defined controller

cases		depth	controllability				synthesis		
id	strategy		$time(s)$	$\|S_{reach}\|$	$\|S_{fail}\|$	$isCtr$	$time(s)$	$\|S_{reach}\|$	$\|code\|$
C1	S1	Full(39)	1.4	396	3	true	1.4	368	60
C2	S1	Full(127)	606.8	459026	10640	yes	NA	NA	165822
C3	S1	Full(43)	160.8	64055	5330	yes	NA	NA	55060
	S1-S2	Full(43)	89.9	40938	2034	yes	NA	NA	33015
	S1-S3	Full(39)	21.8	8906	507	yes	NA	NA	6127
	S1-S4	Full(39)	14.9	6333	319	yes	NA	NA	3662
	S1-S5	Full(36)	7.9	3161	214	yes	7.7	1194	1999
C6	S1-S5	Full(106)	601.3	585521	1609	yes	NA	NA	25752
	S1-S6	Full(85)	479.7	467216	441	yes	NA	NA	10714
	S1-S7	Full(81)	229.9	275626	128	yes	1753	253918	5592

multiple modeling styles.

However, representing dynamic structures (like the list of vehicles and its related operations) is only possible by assuming a higher bound on size and using a fixed size array. This kind of process is error-prone and hinders scalability in the analysis.

For this reason, some other tools provide a modeling language to complement or replace the graph model. This is the case of SMV (Symbolic Model Verifier) or SPIN, which are foremost model checking tools but can be used for synthesis ([9, 13]). UPPAAL-Tiga [2] is a synthesis tool for systems modeled as timed game automata integrated with a C-like language. Our SACD process would be applicable using these tools, though from the modeling point of view, VeriJ was more adapted to our case study than timed automata.

On the other hand, industry has shown major interests in verifying controllers. Formal methods have shown to improve controllers quality of X-ray equipment in [11]. A control algorithm is manually designed [15], to avoid collision among multiple unmanned aerial vehicles (UAVs). Although simulations show good performances, a verification is needed in the safety critical case.

Java Path Finder (JPF) [12] performs model-checking of full Java programs. In a notable industrial experiment of JPF [14], it was used in conjunction with Simulink to detect errors in a given supervisory controller on a UAV. However, JPF by itself does not provide controller synthesis. Because VeriJ is source compatible with Java, JPF can also be used to analyze VeriJ models.

Support of programming languages as input for tools considerably widens the application scope of traditional control theory. For instance, the work [7, 17] have investigated the use of control theory to avoid common synchronization problems in concurrent Java software. In [10], European Research Project FastFIX even proposes to use synthesis in the context of software maintenance. However, support for more expressive languages yield an unsustainable computational cost, whereas the partial approaches we presented in this paper try to avoid this problem by focusing on user-assistance rather than full problem solving.

7. CONCLUSION

We propose a semi-automatic controller design approach, based on the concept of partial controllability and partial synthesis, that helps tackle scalability issues. The use of formal techniques provides proofs of correctness for the designed controllers. Full verification of the controllers designed during the highway case study was possible.

Thanks to the use of a Java-compatible syntax, VeriJ is well adapted to controller design and is easy to use for software engineers who can benefit from powerful Java IDE and various tools for code analysis.

8. REFERENCES

[1] K. Akesson, M. Fabian, H. Flordal, and R. Malik. Supremica - an integrated environment for verification, synthesis and simulation of des. In *8th Int'l WODES*, pages 384–385. IEEE, 2006.

[2] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *CAV*, pages 121–125. Springer, 2007.

[3] B. Bérard, S. Haddad, L. Hillah, F. Kordon, and Y. Thierry-Mieg. Collision avoidance in intelligent transport systems. In *9th Int. WODES*, pages 346–351. IEEE Press, 2008.

[4] K. Bollue, M. Slaats, E. Abrahám, W. Thomas, and D. Abel. Synthesis of behavioral controllers for des: Increasing efficiency. In *11th Int'l WODES*, pages 27–34. IFAC/Elsevier, 2010.

[5] I. Demongodin. Modeling and Analysis of Transportation Networks Using Batches Petri Nets with Controllable Batch Speed. In *Proc. 30th Int. Conf. on Applications and Theory of Petri Nets*, pages 204–222. Springer, 2009.

[6] L. Doyen and J.-F. Raskin. Games with imperfect information: Theory and algorithms. In *Lect. in Game Theory for Computer Scientists*, pages 185–212. 2011.

[7] C. Dragert, J. Dingel, and K. Rudie. Generation of concurrency control code using discrete-event systems theory. In *16th SIGSOFT Int. Symp. on Foundations of Softw. Eng.*, pages 146–157. ACM, 2008.

[8] B. Finkbeiner and H. Peter. Template-based controller synthesis for timed systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 392–406, 2012.

[9] M. M. Gallardo, P. Merino, L. Panizo, and A. Linares. A practical use of model checking for synthesis: generating a dam controller for flood management. *Softw. Pract. Exper.*, 41(11):1329–1347, 2011.

[10] B. Gaudin and A. Bagnato. Software maintenance through supervisory control. In *34th IEEE Softw. Eng. Workshop (SEW)*, pages 97–105. IEEE, 2011.

[11] J. Groote, A. Osaiweran, and J. Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *27th IEEE Int. Conf. on Softw. Maintenance (ICSM)*, pages 467–472. IEEE, 2011.

[12] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[13] M. Hendriks, B. Van Den Nieuwelaar, and F. Vaandrager. Model checker aided design of a controller for a wafer scanner. *Int. Journal on Softw. Tools for Tech. Transfer*, 8(6):633–647, 2006.

[14] F. Lerda, J. Kapinski, H. Maka, E. Clarke, and B. Krogh. Model checking in-the-loop: Finding counterexamples by systematic simulation. In *American Control Conference*, pages 2734–2740. IEEE, 2008.

[15] J. Manathara and D. Ghose. Reactive collision avoidance of multiple realistic UAVs. *Aircraft Engineering and Aerospace Technology*, 83(6):388–396, 2011.

[16] T. Moor, K. Schmidt, and S. Perk. Applied supervisory control for a flexible manufacturing system. In *11th Int'l WODES*, pages 253–258. IFAC/Elsevier, 2010.

[17] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *International Conference on Object Oriented Programming Systems Languages and Applications*, pages 83–98. ACM, 2011.

[18] 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.

[19] Y. Zhang, B. Bérard, L. Hillah, F. Kordon, and Y. Thierry-Mieg. Modeling complex systems with VeriJ. In *Proc. 5th Int. Conf. Verification and Evaluation of Computer and Communication Sys. (VECOS)*, pages 34–45, 2011.