A Closed-loop Model-based Design Approach Based On Automatic Verification and Transformation

Kun Zhang
Electrical and Computer Engineering
The University of Arizona Tucson, USA
dabiezu@email.arizona.edu

Jonathan Sprinkle
Electrical and Computer Engineering
The University of Arizona Tucson, USA
sprinkle@ece.arizona.edu

Abstract

Domain-specific modeling languages effectively constrain structural concepts, but constraints that are not easily captured with structural constraints are still important to fix at design time. In practice these kinds of constraints are implicitly left to be carried out by the domain modelers. This paper explores the process of incorporating system behavioral (not just structural) constraints into a DSML, and studies the way of generating feasible transformation solutions if those constraints fail, based on a transformation library constructed in advance. Our approach is to carry out the verification process through code generation, but utilize the results of verification as an input to a model transformation generator. The output transformation then operates on the original model. As a case study, we applied the approach to finite state machine (FSM) models that control a cyber-physical system.

Keywords Closed-loop Design, Constraints Verification, Model Transformation

1. Introduction

Domain-specific modeling languages (DSMLs) normally scale the systematic design to the level that modelers can manipulate well. A well-defined metamodel can ensure a valid model design in the sense of syntax correctness. However, design errors breaking the behavioral requirements can still happen. For example, modelers can correctly build distributed behavioral models (e.g., interactive finite state machines) without violating any modeling rules, but deadlock risks may still exist due to improper design of event perception timing. At this early stage (before testing the generated deployment code), model verification is preferred since the potential costs are relatively low.

Without the ability to specify behavioral constraints, the DSML cannot generate verification dynamically. When constraints change, verification or its generator needs to be redesigned. In order to overcome the shortcoming, this work incorporates behavioral constraints into the DSML.

Transformation is also required during the design phase, due to design changes or flaws detected in verification. The conventional way is to identify the source and target metamodels and then define the transformation action. Besides laboriousness, this static approach usually requires a span of knowledge of multiple domains. For example, modelers applying FSMs to a specific domain may need to know solutions to deadlock problems, although concurrency techniques have nothing to do with their own domain.

As shown in Figure 1, by (i) incorporating behavioral constraints into the DSML, (ii) automating the verification process, (iii) generating model transformations based on a transformation library constructed in advance, and (iv) running those model transformations automatically, we propose to close the loop of model based design. Thus, behavioral constraints are always interpreted into the verification code to automate the verification. Verification results are then fed into the transformation generator, and the generator outputs a transformation solution, which will be automatically performed on the original model. The loop will keep going on until all constraints are satisfied, and the closed-loop automation can correct models according to the given constraints.

We approach the process of closing the loop by exploring the process as applied to automating the distributed FSM modeling. The reason we select the domain of FSM is that modelers may compose various states for different applications (for discrete systems) and the distributed structures can satisfy concurrent design constraints. Specifically, we integrate behavioral constraints into the FSM DSML. By identifying all types of constraints, we have a list of all possible problems that can violate constraints. We denote these problems by problem nodes, and sort them in the order of priorities. For each node, transformation solutions are developed and then attached to the node. Thus, when a specific problem is detected, its respective transformations can be provided. Note that, the limitation of this approach is that the automation loop can only detect the pre-identified problems and correct them according to existing respective transformations. Problems out of the pre-defined scope can retain.

A similar attempt to create the closed-loop automation is proposed in [1], which incorporates the dynamic constraints

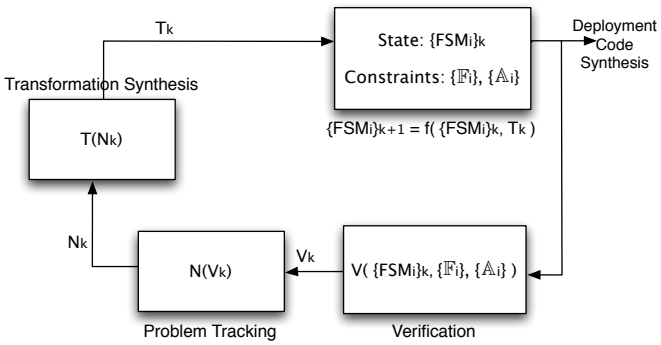

Figure 1. The closed-loop model-based design.

of continuous control systems (e.g., constraints on overshoot and settle time) into the DSML. In contrast, modelers can directly apply our work to systems that can be discretized and modeled as FSMs.

2. Background

2.1 Design Verification

SPIN [2] (Simple Promela Interpreter) is the model checker used in this work, and it takes a Promela model as the input. Our verification code is written in Promela. Works translating a model or programming language (e.g., c language) into Promela code have already been done [3, 4].

SPIN has 4 modes: random, interactive, guided and verification simulations. Verification mode translates Promela into C code for the purpose of speeding up. Verification in SPIN checks all possible trails (organized in a tree structure) that are produced upon interleaving atomic concurrent instructions. When a verification process fails, it stops immediately and outputs the failed trail. We take this trail and the printed text as the verification results to trace problems.

2.2 Model Transformation

Model transformation takes a model as input, and then output an altered model according to the defined operation. The source and target metamodels specify the rules to which the involved models must conform. The GReAT (Graph Rewriting and Transformation) tool [5] for modeling transformation and generating transformation code (written in C language). The context of this work requires the source and target metamodels to be the same one, and the constraints model should not be changed by transformations.

Transformation itself requires testing to ensure correctness [7]. Works, such as [8] and [9], study the approach to automating the testing. Transformations can also be applied to other usages, like [10] proposed to transform the model to a general purpose formalism in order to reduce the costs of maintaining a DSML.

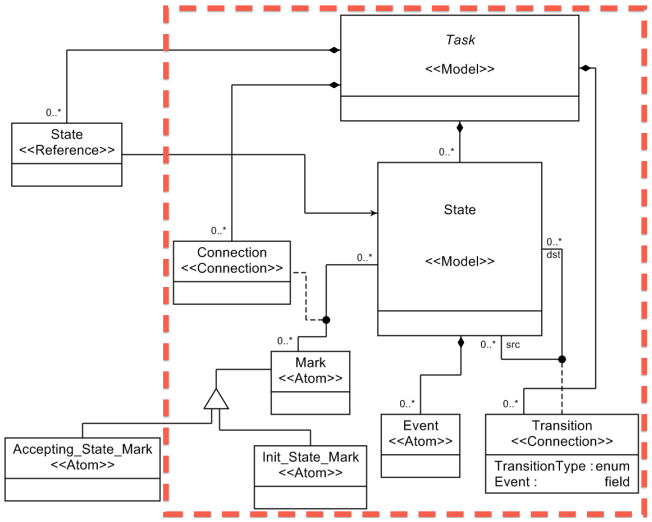

Figure 2. The metamodel of the FSM DSML. The dashed rectangle encloses the part required for FSM modeling, the rest is for modeling behavioral constraints.

2.3 State Machine

A deterministic FSM is a 5-tuple(A, S, s_0, δ, F), where A is the input alphabet, S is a finite set of states, $s_0 \in S$ is the initial state, $\delta : S \times A \rightarrow S$ is the state transition function, and F is the set of accepting states. S, s_0 and δ are defined structurally in the model. Behavioral requirements constrain on F and A.

A common error in distributed FSM is deadlock. Its solution must break the Coffman conditions (conditions that must hold simultaneously for causing a deadlock). The specific conditions that can happen to the FSM are exceptional reception timing (e.g., an event happens before the occurrence of receptors) and circular wait. Another usual error is the interactive behavior of the distributed FSMs may be inconsistent with the design intent. An activity model, given as a constraint, can guard the modeler's intent. A similar work in checking the consistency between FSMs and their collaborations is proposed in [4].

3. Problem Statement

As shown in Figure 1, we denote a set of FSMs by $\{\text{FSM}_i\}_k$, where i is the integer index, and k is the discrete time step. After performing the transformation T_k on $\{\text{FSM}_i\}_k$, the set of FSMs becomes:

$$\{\text{FSM}_i\}_{k+1} = f(\{\text{FSM}_i\}_k, \text{T}_k) \quad (1)$$

For a specific domain under some context, it is possible to identify the full set of types of behavioral constraints. Herein, let behavioral constraints be $\{\mathbb{F}_i\}$ and $\{\mathbb{A}_i\}$, as denoted in section 2.3. The system described in (1) keeps

evolving until the constraints are satisfied:

$$\begin{aligned} \{\mathrm{F}_i\}_k &\subseteq \{\mathbb{F}_i\} \\ \{\mathrm{A}_i\}_k &\subseteq \{\mathbb{A}_i\} \\ \mathrm{F}_i \in \mathrm{FSM}_i, \mathrm{A}_i \in \mathrm{FSM}_i, \exists k \in \mathbb{I}^+ & \end{aligned} \tag{2}$$

The verification engine outputs the verification result V_k, which provides adequate information for tracking a single problem node N_k:

$$\begin{aligned} \mathrm{V}_k &= V(\{\mathrm{FSM}_i\}_k, \{\mathbb{F}_i\}, \{\mathbb{A}_i\}) \\ \mathrm{N}_k &= N(\mathrm{V}_k) \end{aligned} \tag{3}$$

Problem node represents the indexing of problems that can possibly violate constraints. The set of problems is obtained by analyzing the set of types of constraints. Sometimes priorities exist among these problem nodes (e.g., some should be addressed before others), thus, a well-organized structure for maintaining the nodes is required.

The specific problems in FSM domain are deadlock and behavioral inconsistency as stated in Sec 2.3. Problem nodes include exceptional reception timing, mutual exclusion, circular wait, etc. We will take an example of behavioral inconsistency to illustrate the idea.

For each node, a bundle of transformation solutions can be prepared in advance of running the transformation generator. Nodes keep references to their solutions in software implementation. Thus, the transformation is obtained by:

$$\mathrm{T}_k = T(\mathrm{N}_k) \tag{4}$$

Hence, a closed-loop design process is constructed. The purpose of our work is to figure out the process in (1) $\sim$ (4).

There are 3 chanllages: (i) incorporating the domain behavioral constraints ($\{\mathbb{F}_i\}$ and $\{\mathbb{A}_i\}$) into the DSML; (ii) translating the constraints model into the verification code to automate the verification (shown in (3)); and (iii) automatically proposing a set of transformation solutions for a given verification result (shown in (4)).

4. Approach

4.1 Overview of the Closed-loop Automation

The big picture of our implementation of the idea in Figure 1 is shown in Figure 3. The model block represents the model of FSMs, which is implemented in GME (the generic modeling environment) [11]. GME can invoke the interpreter software to generate the Promela code and then the interpreter will call the Spin. We have an external java code to glue the rest part (Step 3 $\sim$ Step 5 in the figure) up.

4.2 Incorporation of Constraints in DSML

We study the process of integrating behavioral constraints into the FSM DSML. Two aspects of constraints, as stated in section 2.3, will be taken into consideration: deadlock and interactively behavioral consistency. The metamodel for generating the FSM modeling language is shown within the dashed rectangle in Figure 2. The rest part ('State Reference' and 'Accepting State Mark') provides models for behavioral constraints.

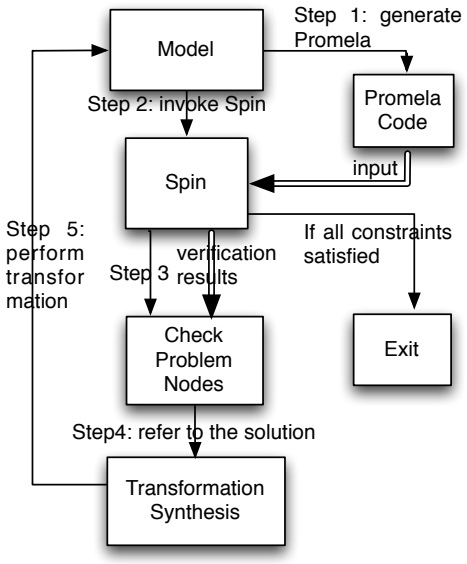

Figure 3. The overall implementation of the automation loop.

Figure 4 is a model example. It shows a set of two distributed (concurrent) state machines. Any execution trail, such as (State 1 $\rightarrow$ State 2 $\rightarrow$ State A) that gets stuck at State A, is valid without specifying constraints. The 'Accepting State Mark' will invalidate this trail due to the occurrence of deadlock, and only 2 trails, (State 1 $\rightarrow$ State A $\rightarrow$ State 2 $\rightarrow$ State B) and (State A $\rightarrow$ State 1 $\rightarrow$ State 2 $\rightarrow$ State B), can pass the constraint.

Suppose the modeler's intent is to let State 1 happen before State A. However, the interleaving between FSMs produces two feasible trails as shown above, and one breaks the intent. We then introduce the activity modeling in Figure 5 to constrain the interaction. The 3^{rd} column in the figure describes the temporal order between State 1 and State A without specifying relationships among other states, since they are implicitly constrained by 'Accepting State Mark'. By adding this constraint, only the behavior (State 1 $\rightarrow$ State A $\rightarrow$ State 2 $\rightarrow$ State B) will be allowed.

4.3 Verification Synthesis

The idea of constructing verification is to generate the Promela code according to the FSMs and the constraints. The FSMs model leads to the code framework. Constraints will be interpreted into logic statements containing assertions or printing clauses. The code is then taken into the Spin checker running in the verification mode.

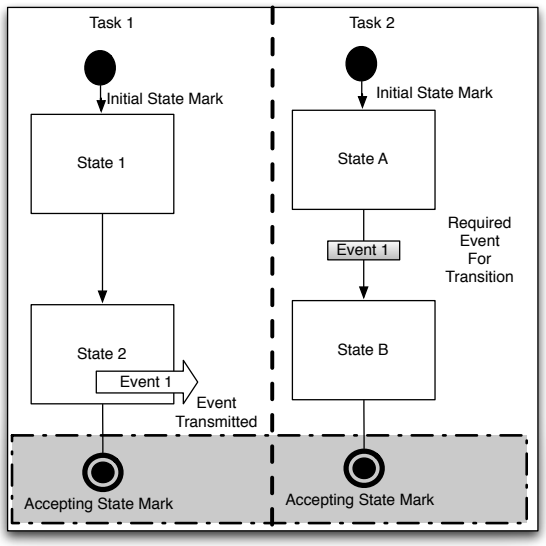

Figure 4. A model example showing two concurrent state machines. The squares represent states. The arrow within a square denotes a transmitted event during the execution of the container state. The small shadowy rectangle is the event required for firing the attached transition. The dashed gray box contains the set of accepting states, which are added if constraints on F are required.

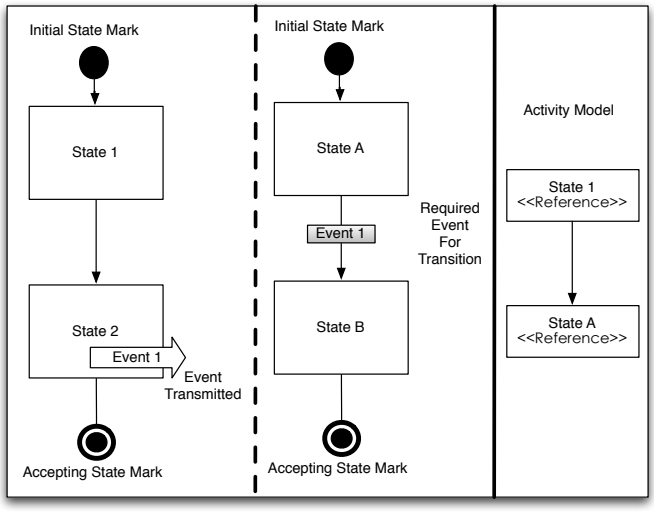

Figure 5. A model example showing two concurrent state machines. The 3^{rd} column is the activity model constraint.

The generated Promela code must be consistent with the generated application code (e.g., the way they handle events). As shown in [6], FSMs are used to generate Java code for applications. For example, in the application code, events are discarded immediately if no transition may accept them. The generated verification must handle events in the same way.

Each state of FSM is translated into a single Promela process 'proctype'. The initial states of all tasks and the event dispatch process will start at the very beginning. Each state has a counter variable associated with it, showing the times the state has run. Constraints introduced by the activity model are translated into a comparison statement in Promela. For example, the statement would result in a false assertion if the comparison detects that the 'State A' happens before the 'State 1'. Then, the next state following the current state will be invoked automatically if the transition is a completion transition. Otherwise, only when the transition is waiting for a specified event and the event is dispatched at this time, the transition can be fired. This dispatch process also checks if all task processes are active, and reports a timeout message if all processes are frozen.

Then SPIN will run the Promela code in verification mode. If a constraint violation detected, SPIN stops and generates the trail log. We then feed the Promela code with the log into SPIN to recur the violation with details printed out. Based on the details, we can infer the problem node that causes the violation.

4.4 Transformation Solutions

Based on the above 2 types of constraints, we list the possible problems in the order of processing priorities: (i) the event required for a transition does not exist in the model; (ii) the event required for a transition will not happen after the receptor is ready to receive; (iii) a circular wait exists; (iv) a specific behavioral trail breaks the activity model constraint. These problems are the problem nodes in section 3, and each node maintains references to respective transformation solutions.

Every iteration in the automation loop (in Figure 1) checks all problem nodes. If one node violates the constraints, the loop performs the node's transformation solution first, and then runs the next iteration.

The trail (State 1 $\rightarrow$ State 2 $\rightarrow$ State A) will cause deadlock since Event 1 is released before the transition between State A and B can receive it, and the (ii) node will be responsible. A solution example, shown in Figure 6, is pointed by the (ii) node. The solution is implemented in the GReAt, and then be translated into executable code. In this figure, the models locating outside the dashed rectangle represent the State A, B and the transition. This is an identification of what part of the design will be altered. The dash rectangle contains the models that will be added to the original design. A transitory State 3 is connected via a transition from State A and a transition to State B. The transformation result

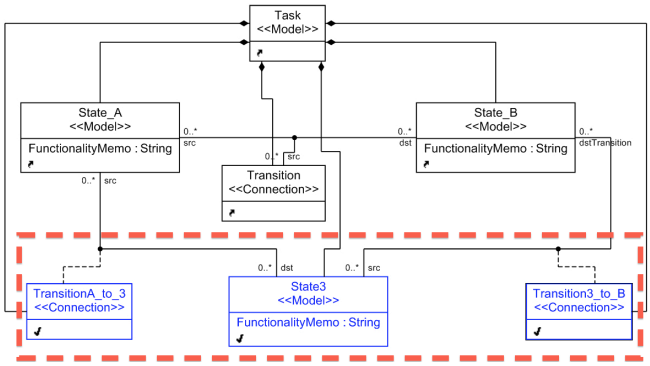

Figure 6. A transformation solution example for deadlock problem. Out side the dashed rectangle is the identification of the existing model and relationship in modeler's design, the rectangle contained part is the new model and relationship that will be add to the design.

is shown in Figure 7. The dashed block avoids the possible deadlock by allowing transition to State B via the State 3 if the event has happened. The State A and State B in Figure 6 is not necessarily the corresponding states in Figure 7, they are actually identified by the models' unique IDs.

After running the automation loop, the final result is shown in Figure 8. A transitory state is added at the very beginning in Task2, and the transition to State A will happen only after the State 1 has been executed. State B will never be reached unless State 2 is executed and the Event 1 has been transmitted. Thus, the automation loop produces the model that sticks to the trail (State 1 $\rightarrow$ State A $\rightarrow$ State 2 $\rightarrow$ State B) as desired.

5. Application Example

A good example to apply our research is a model-based design work for configurable sensor network in river environment [6]. The domain background is that, a group of drifters, equipped with propellers and sensors, are released into the river for the purpose of real-time water quality monitoring. They also report their own conditions (e.g, position, tilt). They communicate via cellular network with a server, which is responsible for synchronizing data among the database and the drifters.

The approach in [6] is to design the DSML utilizing FSM concepts as language structure and using domain concepts as events. We have further developed the DSML and the generated Java code also has been well improved for the purpose of efficiently concurrent performance.

The design using FSMs on the server side and the drifter side are shown in Figure 9 and Figure 10. Each side has four complex state machines, and interacts not only with other state machines within its own side, but also with with machines in the other side via networking. The complexity explodes when the number of drifters gets larger. The number

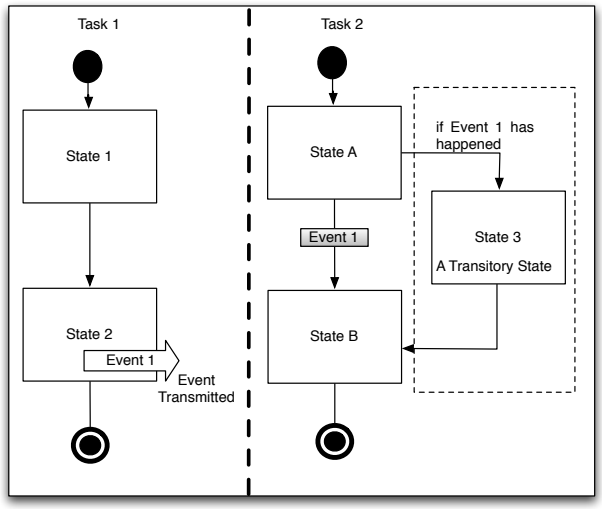

Figure 7. The 1^{st}step to correct the models. Deadlock can happen in the original design by following the trail (State 1 $\rightarrow$ State 2 $\rightarrow$ State A), due to that the Event 1 can happen earlier than the receptor is ready to receive. The transitory state is added by transformation in Figure 6.

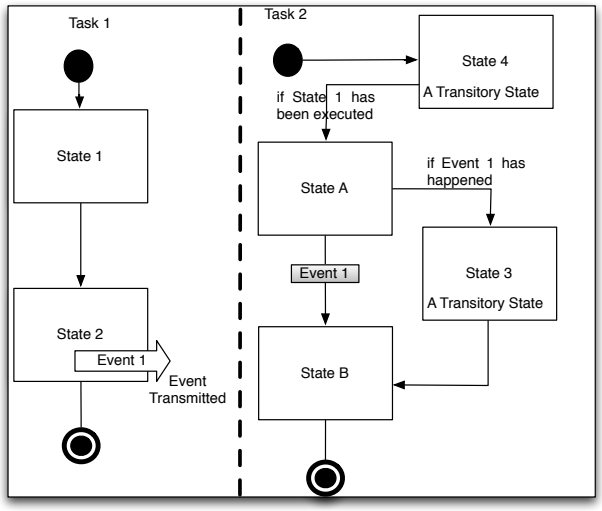

Figure 8. The final result after fully running the automation loop in Figure 1. A transitory state (State 4) is added to ensure that State A always happen after State 1.

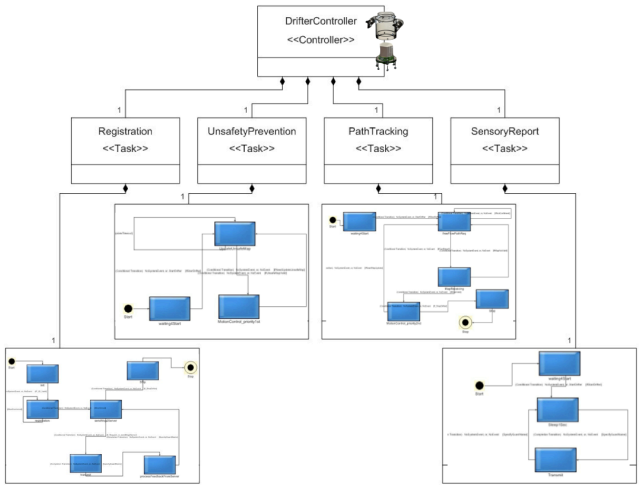

Figure 9. Model on the drifter side.

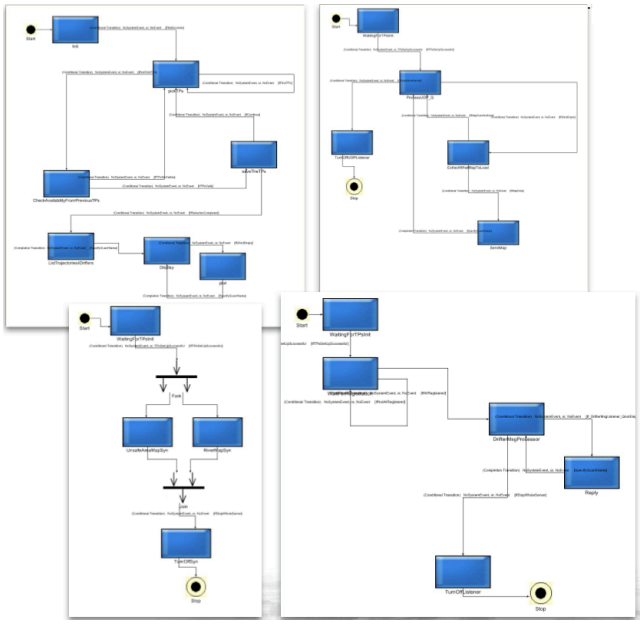

Figure 10. Model on the server side.

is assumed to be 100 since the research team had built up to 100 drifters. Under this circumstances, the automation loop approach we proposed can provide large potential help, during the early design stage.

We specify the set of accepting states in those FSMs since we intent to stop the whole software properly. The activity modeling provides a few constraints, such as the speed report should be later than receiving a valid GPS signal. After satisfying the constraints, the model design phase ends, and the remaining phases just follow the regular waterfall development pattern.

6. Conclusion

We study the approach of how to close the loop of design automation by exploring an example in the FSM domain. Specifically, we use GME to construct the metamodel and models. The behavioral constraints are also incorporated into the DSML. An interpreter translating models and constraints into Promela code is developed. Spin is used as the checker to verify the Promela code. The rest part in the automation loop is glued up by a Java-written software. The set of transformation solutions is produced based on analyses of all possible problems that can violate those constraints. We apply this approach to the application of model based design in configurable river sensor network. The closed-loop design automation can correct flaws in modeling stage with a relatively low cost.

Acknowledgments

This research was funded by NSF CPS award # CNS-0930919.

References

[1] Whitsitt, Sean, ”Automatic Verification of Dynamic Constraints in LTI Control Systems Through Model Transformations”, NSF Young Professionals Workshop on Exploring New Frontiers in Cyber-Physical Systems. NSF, Washington, DC, 2014.

[2] http://spinroot.com/spin/whatispin.html

[3] Jiang, Ke. ”Model Checking C Programs by Translating C to Promela.” , 2009.

[4] SchŁfer, Timm, Alexander Knapp, and Stephan Merz. ”Model checking UML state machines and collaborations.” Electronic Notes in Theoretical Computer Science 55.3 (2001): 357-369.

[5] http://www.isis.vanderbilt.edu/tools/GReAT

[6] Zhang, Kun, and Jonathan Sprinkle. ”Model-Based Software Synthesis for Self-Reconfigurable Sensor Network in Water Monitoring.” Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the. IEEE, 2013.

[7] Fleurey, Franck, et al. ”Qualifying input test data for model transformations.” Software & Systems Modeling 8.2 (2009): 185-203.

[8] Sen, Sagar, Benoit Baudry, and Jean-Marie Mottu. ”Automatic model generation strategies for model transformation testing.” Theory and Practice of Model Transformations. Springer Berlin Heidelberg, 2009. 148-164.

[9] Sani, Asmiza Abdul, Fiona AC Polack, and Richard F. Paige. ”Model transformation specification for automated formal verification.” Software Engineering (MySEC), 2011 5th Malaysian Conference in. IEEE, 2011.

[10] Pedro, Luis, Levi Lucio, and Didier Buchs. ”Principles for system prototype and verification using metamodel based transformations.” Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on. IEEE, 2006.

[11] http://www.isis.vanderbilt.edu/projects/GME