A Framework for Developing Cyber Physical Systems

Xudong He
Florida International University, Miami, USA
Heng Yin
University of California at Riverside, Riverside, USA

Zhijiang Dong
Middle Tennessee State University, Murfreesboro, USA
Yujian Fu
Alabama A & M University, Huntsville, USA

***Abstract*— Cyber physical systems (CPSs) are pervasive in our daily life from mobile phones to auto driving cars. CPSs are inherently complex due to their sophisticated behaviors and thus difficult to build. In this paper, we propose a framework to develop CPSs based on a model driven approach with quality assurance throughout the development process. An agent-oriented approach is used to model individual physical and computation processes using high level Petri nets, and an aspect-oriented approach is used to integrate individual models. The Petri net models are systematically mapped to classes and threads in Java, which are enhanced and extended with domain specific functionalities. Complementary quality assurance techniques are applied throughout system development and deployment, including simulation and model checking of design models, model checking of Java code, and run-time verification of Java executable. We demonstrate our framework using a car parking system.**

Keywords - cyber physical systems; model driven development; high level Petri nets; simulation; model checking, runtime verification

I. INTRODUCTION

Cyber physical systems (CPSs) are pervasive in our daily life and need to be extremely reliable since they are often safety critical. CPSs consisting of computation and physical processes are inherently complex and demonstrate many sophisticated behaviors including synchronous, asynchronous, distributed, real-time, discrete, and continuous [1]. In [2], several major design challenges of CPSs were discussed, including concurrency and timing, which are intrinsic and critical in CPSs but are not adequately addressed in current computing abstractions. While fundamental new technologies are needed to develop CPSs, incremental improvements of existing technologies including formal verification, simulation, software engineering processes, and design patterns are important parts of a potential solution [2].

In this paper, we provide a concrete framework to realize the ideas in [2]. We present a model driven approach from high level Petri nets to Java programs where several design heuristics are provided for program derivation and system properties mapping. Essential CPS design issues including concurrency and timing are modeled using high level Petri nets and analyzed through model checking and simulation. Assumed environment constraints from hardware devices are checked during implementation and runtime verification. The overall framework is shown in Fig. 1.

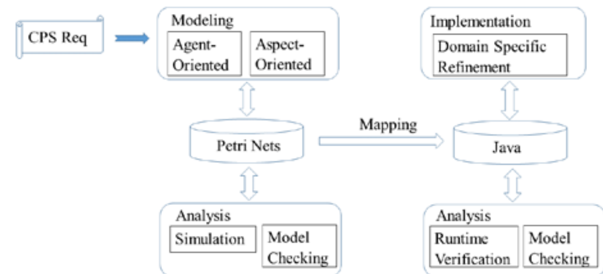

Fig. 1 – A framework for developing cyber physical systems

Petri nets are a formal method well suited for modeling concurrent and distributed systems. Various time extended Petri nets are capable to deal with real-time systems [3]. High level Petri nets can use time stamps associated with tokens and timing related transition constraints to simulate time Petri nets [4]. Thus high level Petri nets are an excellent formal method for modeling essential features of CPSs. In addition, we have developed an agent oriented modeling approach to capture CPSs at a high abstraction level where meaningful computational components and physical processes with independent behaviors are viewed as agents and modeled individual high level Petri nets. An aspect oriented approach is used to incrementally integrate system components represented using individual high level Petri nets into a complete system represented in a single system high level Petri net. The resulting system net can be analyzed through simulation as well as model checking. The above modeling and analysis techniques are supported by tool chain PIPE+ [5] and SPIN [6]. A systematic translation approach has been developed, where a set of translation rules is used to map the individual agent nets into corresponding Java threads to form the general program structure. A complete Java program is obtained by combining the translated general program structure with domain specific program refinements. The additional refinements are necessary to realize CPSs, especially domain dependent physical devices. Bounded symbolic model checking and runtime-time verification are performed to ensure model level properties and additional properties are not violated in the implementation. The model level analysis and implementation level analysis are complementary. At model level, both safety and liveness properties can be checked to detect potential errors in the requirements with environmental assumptions such as the hardware devices working properly. At the implementation level, safety properties can be checked through bounded symbolic model checking and monitoring the actual behavior of hardware devices.

Our main contributions include: (1) a formal framework for developing CPSs supported by a tool chain, (2) an incremental agent-oriented modeling methodology for representing CPSs using high level Petri nets, (3) a model level analysis methodology combining simulation and model checking, (4) a pattern based translation method for generating Java threads from high level Petri net models, and (5) an implementation level analysis methodology combining bounded symbolic model checking and dynamic runtime verification.

DOI reference number: 10.18293/SEKE2017-095

II. CPS MODELING AND ANALYSIS

To effectively model and analyze the complex behaviors of CPSs, many modeling techniques have been proposed and adapted in recent years including formal methods such as hybrid automata [7] and special graphical modeling languages such as actor-oriented MoC [8]. High level Petri nets [9] are well suited to model the complex behaviors of CPSs. The graphical representation and data flow nature of Petri nets provide a natural and easy to understand model to capture physical and computation processes in CPSs. The executability of Petri nets further facilitate model level analysis. In this paper, we propose an agent-oriented approach to model individual physical and computation processes by extending our prior work [10] and an aspect-oriented approach [9] to synthesize individual models to obtain a complete system model. In the following sections, we provide some design heuristics of applying high level Petri nets to model CPSs and demonstrate them using a car parking system. The detailed Petri net definitions are omitted due to space limit.

A. *Modeling Individual Components*

A high level Petri net can be used to capture the structure and the behavior of a physical or computation process. Petri nets naturally support synchronous, asynchronous, and distributed control and data flows. High level Petri nets are capable to model virtual time through time stamps associated with tokens and transition constraints representing delays and durations. Continuous behaviors of physical devices can be abstracted and discretized using real typed places and the associated transitions, and can be further refined during implementation.

Each type of physical devices (sensors and actuators) or computation processes is modeled with a high level Petri net called an agent net that has its own meaningful and demonstrates independent reactive and/or proactive agent behavior interacting with external environments, while concrete physical devices or computation processes are with structured tokens containing unique identification in the 1st field. Specifically, we provide the following general design heuristics in building an agent net:

- Attributes of a physical device or states of a computation process are defined by places with appropriate types. Discrete values are defined using string or integer types and continuous values are defined using real type. Structured types (Cartesian product of basic types) are used to define complex attributes. Powerset is used to define multiple physical devices and computation processes;
- Actions or state transitions are modeled with transitions containing first order logic formulas defining the preconditions and post-conditions;
- The interaction between a physical device and an external environment can be modeled with a transition containing a random function emulating the possible values from the environment (open system) or with a transition picking up a possible value from an additional place denoting the external environment (closed system);
- Virtual time is modeled with tokens having an additional field denoting time stamps and a special place modeling a logical clock.

We demonstrate the above design heuristics in modeling a robotic car parking system. Each robotic car has one color sensor for navigating a path with colored lines and an ultrasonic sensor for detecting obstacle during parking. The car parking process involves the following steps: (1) finds the entrance of a parking garage by detecting a green line, (2) moves forward along a red line, (3) makes a turn when a blue line is detected, and (4) completes the parking when the minimum specified distance is reached. Based on the above simple system description. Three individual agent nets corresponding to the color sensor, the ultrasonic sensor, and the car parking process are constructed. Only essential attributes of the sensors and car are represented, for example, only one place *ColorSensor* for holding the current detected color is needed for the color sensor.

B. *Modeling the Whole System*

The overall agent system is obtained by integrating individual agent nets to form a system net that shows the interaction, communication, and cooperation among different agents. Synchronized activities are modeled through new joint transitions with modified constraints, and asynchronous activities are modeled through connecting a place in one agent net to a transition in another agent net. An aspect oriented approach [9] is used to build a complex model incrementally through weaving individual Petri nets representing agents capturing physical devices and computation processes. This aspect oriented approach further supports system adaptation and evolution, and facilitates compositional analysis. The overall car parking system model after weaving three agent nets is shown in the following Fig. 2.

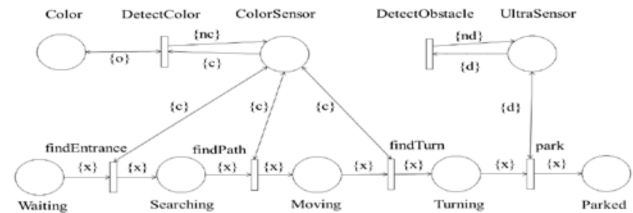

Fig. 2 – The overall system integrating three agent models

C. *Analyzing the System Model*

A CPS system is often a hybrid system consisting of both continuous hardware devices and discrete computation processes. In most cases, the only available technique for continuous components is simulation. However, formal verification techniques based on symbolic reachability analysis is available for sub classes of hybrid systems such as those can be modeled using linear hybrid automata [1] where the state transition rates are constants with restricted checking and updating actions. High level Petri nets are executable and thus support simulation of hybrid system models. Furthermore our tool PIPE+ supports reachability analysis and model checking using SPIN in addition to simulation.

1) Simulation and Simple Reachability Analysis

Simulation is carried by firing enabled transitions. Two modes of simulation can be done in PIPE+: single steps and multiple steps. Simulation can be used to test whether a model

satisfies the CPS requirements by examining the tokens in places of interest or transition firing history. In addition to simulation, we have also implemented a simple reachability analysis that checks whether a place hold a particular value during an execution, which either confirms our intention (witness) or finds a potential error (counter example). When a reachability is confirmed, the simulation time and the transition firing sequence are recorded. However simple reachability analysis may not be conclusive when a search does not end. Repeated checking can be used to eliminate possible false negative in systems with finite execution sequences.

The following table provides a simple reachability analysis of whether a waiting car will be parked.

Table I – Simple reachability checking results

Parked	Reachability	Time (ms)
c	Yes	459

The firing sequence is very long in this case since both the color sensor and the ultrasonic sensor are modeled as independent agents, which are always enabled and keep firing.

2) Model Checking

Model checking performs exhaustive search on finite state systems and thus is not directly applicable to continuous systems. However we may be able to model check the bounds (called barrier certificates) of some continuous state variables. PIPE+ has a translator that automatically converts a high level Petri net model to a Promela program in SPIN. During the translation, each place is translated into a channel with the place's type. This kind of conversion may not always work due to the loss of precision since Promela only supports integer.

Properties to be checked are expressed using linear time temporal logic formulas. Safety and liveness properties are expressed in the general form []*placename*(*x*) and <> *placename*(*x*) respectively, where [] and <> are the temporal operators always and sometimes in SPIN and x can be a variable or a constant (a specific token). More complex formulas are defined using logical connectives. A safety property of the parking system is that a car is collision free during parking []!(*UltraSensor*(*v*)&&*v*<5), where s is the ultrasonic sensor detected distance value. In the model, we can only use assumed value range for checking. A liveness property of the car parking system is that a waiting car will eventually be parked. Table II provides model checking results of the above properties.

Table II – Model checking results

Property	Satisfied	Time (ms)
[]!(*UltraSensor*(*v*) && *v* <5)	Yes	47
Waiting(*c*)-><>*Parked*(*c*)	Yes	1

III. Model Realization

Design models help us to better understand system features including functionality, structure, and behavior as well as to detect and prevent early system development errors. To leverage the design models to increase productivity and improve code quality, model driven development based on UML emerged in the last decade [11], in which UML based models are translated into programs of object oriented programming languages. However since there are multiple UML notations such as class diagram, state machine diagram, and sequence diagram for representing different aspects of a system, it is not easy to obtain a coherent set of code. We present a model driven approach to realize our high level Petri net models, which provides a systematic way of writing Java programs and establishes the traceability between the models and resulting programs. Our model driven approach consists of the general code structure and domain specific refinement. The general code structure can be systematically generated from the agent models and the overall system model. However the domain specific refinement requires manual process in identifying and defining additional features of the system, especially with regard to the physical devices.

The following translation rules are used to generate the general code structure from high level Petri net models:

(1) A class is generated for each agent net, where attributes are defined based on the unique data type fields of the places, and methods are the transitions. The behavior of objects (tokens) is defined by the net structure;

(2) A thread is created based on each class in (1) to capture the independent active behavior of agents modeled by the agent net;

(3) Agent interactions modeled through agent net weaving are translated into method calls between threads in (2);

(4) A main program is generated for the overall model, which starts all the threads generated in (2) according to the initial marking;

(5) A package is created to include the above code files.

Applying the above translation rules to the car parking system, we obtain the following Java code skeleton (due to space limit, only the translation of car parking agent model and overall system model are shown) together with some blue colored domain specific refinement code:

- The `Car.java` class and the `Parking.java` thread from parking process agent model:

```
package EV3;
import lejos.hardware.motor.Motor;
import lejos.hardware.motor.NXTRegulatedMotor;
…
public class Car {
    float wheelWidth = 5.5; // in cm
    float trackWidth = 30.0; // in cm
    NXTRegulatedMotor leftM = Motor.A;
    NXTRegulatedMotor rightM = Motor.B;
    UltraSensor ultrasensor;
    ColorSensor colorsensor;
    …
}
package EV3;
…
public class Parking extends Thread {
    private Car carobj;
    public void findEntrance (…) {…}
    public void findPath (…) {…}
    public void findTurn (…) {…}
    public void park (…) {…}
    …
    public void run {
```

```
                findEntrance(…);
                findPath(…);
                findTurn(…);
                park(…);
        }
    }
```

- The main program from the system model:

```
package EV3;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import lejos.robotics.Color;
import lejos.utility.Delay;
public Class Init {
    public static void main(String[] args){
     Car car = new Car();
     ColorDetection cd = new ColorDetection(car);
ObstacleDetection od = new ObstacleDetection(car);
     Parking p = new Parking(car);
     cd.start();
     od.start();
     p.run();
   }
}
```

In the above code, `ColorDetection` is a thread object that asks the color sensor to keep reading and storing the current color value as long as parking is not completed. The color sensor is represented by the class `ColorSensor`. `ObstacleDetection` is also a thread object that asks the ultrasonic sensor to keep reading the distance to the object ahead of the car in current path as long as parking is not completed. The ultrasonic sensor is represented by the class `UltrasonicSensor`.

IV. Runtime Verification

A good model can speed up the development of the system and provide quality assurance to the implementation. However, it doesn't guarantee the correctness of the implementation due to several reasons. First, the model is an abstraction of the system. Some details and algorithms are omitted intentionally and some system properties cannot be specified. Second, the model driven approach requires some manual refinements and the complete system with additional code needs to be checked.

To overcome the limitations and restrictions of model driven development, we adopt runtime verification to ensure system properties at implementation level. Runtime verification is a lightweight formal approach to detect violation of properties. In our work, properties are specified using linear temporal logic (LTL) formula. Monitors are generated from these LTL formulas using JavaMop [12] and woven into system implementation as aspects using AspectJ [13]. This ensures the independence of system implementation from monitor – the runtime verification code.

LTL properties specified and analyzed at model level need to be monitored in implementation level to improve confidence of system implementation. However, atomic predicates of LTL formula in implementation level typically represent occurrences of events such as object creation, object initialization, method call, member data access and mutation. Such events can be specified in conjunction with conditions on function arguments, which are absent at model level. The following principles are provided to guide the mapping of atomic predicates (places) in Petri net model to implementation:

(1) A place representing a state of the agent net:

a. If a class generated from the agent net has a member data representing the state, then the place is mapped to the event of changing member data. For example, place Parked represents a state of car and class `Car` has a member data `parkComplete` denoting this state, thus Parked(c) is mapped to the event of `Car.parkComplete` being set to true.

b. If a class generated from the agent net doesn't have a member data representing the state, then an object reaching this state is a result of a function call and is mapped to the event that occurs whenever the function call is completed.

(2) A place representing the duration of an action such as Turning in the agent net: the place is mapped to the event of executing the function starting the action.

In our study, five different properties are designed and monitored at runtime, including:

Property 1: After detecting the entrance, a car will park successfully within a given time period;

Property 2: A car starts turning to a parking lot only after a blue line is detected;

Property 3: During the parking process, a car will never approach too close to any object on the path.

Due to space limit, we only provide and discuss the monitor code of property 1. This liveness property is to guarantee the correctness of the overall program: A car eventually parks successfully. In the model level, it is specified as: [](*Waiting*(*c*) -> <>*Parked*(*c*)). This property can be verified in the model but its violation cannot be determined at runtime due to its infinite nature. Therefore, we modify the property to reflect the reality – the car must be parked within a given time period. Such modification is reasonable given the unreliable nature of robots: the color sensor may report wrong color or miss a color. The monitor of this property contains the following three main events: `entranceDetectedtrue`, `parkCompletedtrue` and timeout. Event `entranceDetectedtrue` occurs when car's member data `entranceDetected` becomes true. When `entranceDetectedtrue` occurs, a timer is started. Event `parkCompletedtrue` occurs when car's member data `parkComplete` is set to true. Event timeout occurs when a given time expires.

```
event entranceDetectedtrue after(Car car, boolean val) :
     set(private boolean Car.entranceDetected)
     && args(val)  && target(car) && condition(val){
        //start timer
        ...
     }
event parkCompletedtrue after(Car car, boolean val) :
     set(private boolean Car.parkComplete)
     && args(val) && target(car) && condition(val)) {
        //kill timer
     }
event timeout after(TimeoutEventGenerator t) :
execution(publicvoidTimeoutEventGenerator.timeoutEvent())
     && target(t) && condition(t == teg) {
     }
ltl:[](entranceDetectedtrue =>
          (!timeout U parkCompletedtrue))
```

The second property in model level specified as: [](*Searching*(*c*) => <>*ColorSensor*(*Blue*)) can be verified at both model level and implementation level.

The third property is a safety property to ensure collision

free during parking. In model level, it is specified as: $[](UltraSensor(v) \wedge v < \text{value})$, where value is a given constant. However, the verification results of such safety properties are based on some assumptions, and thus their violations must be monitored during runtime.

Therefore runtime verification complements model level and implementation analysis through monitoring program behaviors that cannot be fully verified at model and implementation levels due to the limitation of models or bounded space explosion at implementation level.

V. Implementation-Level Model Checking

Implementation level model checking is complementary to both modeling and runtime verification. While modeling is concerned about the correctness at design level, as the name suggests, implementation level model checking aims to verify the properties in the actual software implementation, because errors may be introduced during the software implementation.

It is also well accepted that model checking and runtime verification should be combined to address each other's weaknesses [14, 15]. Model checking achieves completeness, but often is too conservative and impractical for many realistic applications, whereas runtime verification can perform excellent checks over a portion of the program that is actually executed during the real deployment.

Researchers have explored different approaches to combine model checking and runtime verification. For instance, [15] presents an analysis framework that can reuse the same analysis/verification algorithms for both static and dynamic analysis, in other words, model checking and runtime verification. While this is useful, it does not answer when to use model checking and when to use runtime verification. [14] explores the idea of partitioning a software system, such that one partition can be verified using model checking and the other can be checked via runtime verification. However, a user must identify this partitioning boundary based on her domain knowledge, which may not be practical in reality.

A. *Bounded Symbolic Model Checking*

In this work, we explore a new approach to combine model checking and runtime verification, namely bounded symbolic model checking. Specifically, we perform symbolic execution to examine the program execution space as much as possible to directly verify the property validation logic inserted by runtime verification. Given that a program may have infinite loops or loop conditions depending on symbolic inputs, the program execution space is infinitely large. As a result, our symbolic execution is bounded to search limited iterations in each loop.

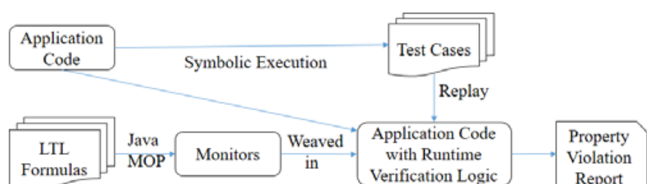

Fig. 3 – Workflow of Bounded Symbolic Model Checking

Fig.3 illustrates the workflow of this technique. Given a CPS program, we perform symbolic execution to generate a set of test cases, which will exercise as many code paths as possible. These test cases are then fed into the program with runtime verification code weaved in, to verify the properties.

It is worth noting that we do not directly perform symbolic execution on the program with runtime verification code weaved in. This is because the inserted runtime verification code greatly increases the complexity of the code logic, and symbolic execution on it will quickly run into state explosion problem.

Moreover, for convenience, we directly take advantage of the runtime verification code that has already been inserted, rather than verify the properties separately. In this way, we can also detect errors introduced by runtime verification.

B. *Implementation*

We implemented this idea as a plugin to the Java Path Finder framework [16] to perform model checking on LeJOS programs

1) *JPF model classes*

A LeJOS program can only run inside the LeJOS environment. As a result, it cannot run directly within JPF. In order to analyze a LeJOS program within JPF, we need to provide proper model classes to simulate the behaviors of related LeJOS APIs and classes. More specifically, we model following LeJOS classes:

```
lejos.hardware.lcd.LCD;
lejos.hardware.motor.BaseRegulatedMotor;
lejos.hardware.sensor.EV3ColorSensor;
lejos.hardware.sensor.EV3UltrasonicSensor;
lejos.hardware.sensor.SensorMode;
lejos.hardware.sensor.UARTSensor;
lejos.robotics.navigation.MovePilot;
```

2) *JPF Symbolic Execution*

We make use of JPF-symbc to perform symbolic execution. For the auto parking example, the program reads input from the color sensor and the ultrasonic sensor. Therefore, by adding JPF symbolic annotations, we make these sensors return symbolic values.

Since the auto parking program repeatedly reads from the sensors and perform corresponding behaviors, there exist several infinite loops. To ensure symbolic execution terminates within a reasonable time frame, we limit the number of loop iterations, by using JPF Verify API.

Following is a code snippet showing how we limit loop iterations in the auto parking program:

```
int in_count = 0;
int out_count = 0;
while (carObj.isFindLine()) {
    int color = getColorSin(1);
    while(color != Color.RED && color != Color.BLUE){
        in_count ++;
        Verify.ignoreIf(in_count > 3);
        ...
    }
    out_count++;
    Verify.ignoreIf(out_count > 2);
    ...
}
```

In this example, there are two while loops, inner loop and

outer loop. There is a chance of program being trapped in either inner loop or outer loop (or both) during execution. So, we add a counter for each loop. When the counter is larger than a threshold, JPF will not continue execute the current path. We then solve the constraint for each path, and generate test inputs.

C. Evaluation results

Table III – Bounded symbolic model checking results

# of test cases	# of violations	Generation time	Replay time
688	344	519 s	13120 s

Table III lists the performance results for our bounded symbolic model checking on the original auto parking example. Symbolic execution generated 688 test cases totally in 519 seconds.

We then replayed these test cases on the auto parking program with runtime verification logic weaved in. The replay lasted for 13120 seconds. This process is significantly slower, because the auto parking program periodically calls `Delay.msDelay(int time)` to wait for amount of time. Symbolic execution would ignore this delay to quickly explore multiple execution paths, while during replay in order to validate time-related properties, we cannot ignore these delays.

Eventually, we observed violations in 344 test cases. All the violations happened on only one LTL property, which is collision free property. This property is violated when the EV3 car approaches too close to an obstacle.

```
public void turn() {
    if (carObj.getCMD() == 1){
        carObj.pilot.stop();
        carObj.pilot.setAngularSpeed(7);
        carObj.pilot.arc(25, 90, true);
        carObj.setMoving(true);
        Delay.msDelay(15000);
    }else if (carObj.getCMD()==0){
        carObj.pilot.stop();
        carObj.setMoving(false);
    }
}
```

In the above example, the EV3 car would possibly turn for 15000 milliseconds. After each turn, Ultrasonic sensor is used to detect distance between an obstacle and EV3. When the EV3 car is running in a physical system (parking lot), this could be a safe action due to parking lot layout. However, during symbolic execution without knowledge about physical system, JPF explores all possible paths, and could read in a distance value less than safe distance after such a long turn, and results in a violation of collision free property.

Through bounded symbolic model checking, we have verified that four LTL properties are ensured at least within a limited search scope. When resource is limited on the LeJOS system, one might consider removing the runtime monitor code for checking these four properties. On the other hand, collision free property cannot be properly verified using model checking, so runtime verification on this property is necessary.

VI. Conclusion

This paper presented a framework for developing CPSs supported by a tool chain. High level Petri nets are used for modeling CPSs due to their capability in addressing the critical features including concurrency and timing of CPSs. An incremental agent-oriented modeling methodology is used for creating CPS models. The resulting models are analyzed using simulation and model checking to detect early design problems. A translation method for generating general Java thread structure from high level Petri net models is provided. The resulting general Java code structure is manually extended with domain specific code refinement to obtain a complete program. This partial manual process of domain specific refinement requires creativity in adding details and thus is unavoidable; however is minimized in our framework. Implementation level quality assurance is carried out by combining bounded symbolic model checking and dynamic runtime verification. We demonstrated our framework thorough a simple parking system. There are still many gaps to fill to make our framework successful. We are currently working on a multi-car parking system and a drone system to gain more experience with regard to the applicability and scalability of our approach.

Acknowledgment

This work was partially supported by AFRL under FA8750-15-2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] R. Alur: "Principles of Cyber-Physical Systems", MIT Press, 2015.

[2] E. Lee: "Cyber Physical Systems: Design Challenges", Proc. of International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing, Orlando, FL, 2008, 363-369.

[3] D. Xu, X. He, and Y. Deng: "Schedulability Analysis of Real-Time Systems Using Time Petri Nets", IEEE Transaction on Software Engineering, vol.28, no.10, 2002, 984-996.

[4] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzi: "A Unified High-Level Petri Net Formalism for Time-Critical Systems", IEEE Transactions on Software Engineering, vol.17, no. 2, 1991, 160-172.

[5] Su Liu and Xudong He: "PIPE+Verifier - A Tool for Analyzing High Level Petri Nets", Proc. of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE15), Pittsburgh, July 6 – 8, 2015.

[6] Gerard Holzmann: The SPIN Model Checker, Addison Wesley, 2004.

[7] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine: "The algorithmic analysis of hybrid systems", Theoretical Computer Science, vol. 138, 1995, 3 – 34.

[8] P. Derler, E. Lee, and A. Vincentelli: "Modeling Cyber-Physical Systems", Proceedings of the IEE, vol. 100, no.1, 2012, 13 – 28.

[9] X. He: "A Comprehensive Survey of Petri Net Modeling in Software Engineering", International Journal of Software Engineering and Knowledge Engineering, vol. 23, no. 5, 2013, 589-626.

[10] L. Chang, S. Shatz, and X. He: "A Methodology for Modeling Multi-agent Systems using Nested Petri Nets", International Journal of Software Engineering and Knowledge Engineering, vol.22, no.7, 2012, 891-926.

[11] B. Selic: "The Pragmatics of Model-Driven Development", IEEE Software, 2003, 10 – 25.

[12] D. Jin, P. Meredith, C. Lee, and G. Rosu: "JavaMop: Efficient Parametric Runtime Monitoring Framework", International Conference on Software Engineering, Zurich, Switzerland, June 2 – 9, 2012.

[13] The AspectJ Project homepage: https://eclipse.org/aspectj/.

[14] T. Hinrichs, P. Sistla, and L. Zuck: "Model Check What You Can, Runtime Verify the Rest", EPiC Series in Computing, vol. 42, 2014, 234 – 244.

[15] C. Artho and A. Biere: "Combining Static and Dynamic Analysis", Electronic Notes in Theoretical Computer Science, vol.131, 2005, 3 – 14.

[16] Java Path Finder: http://javapathfinder.sourceforge.net.