A Systematic Approach for Developing Cyber Physical Systems

Xudong He
Florida International University, Miami, 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 systematic approach to develop CPSs with quality assurance throughout the development process. A CPS is abstracted and partitioned into a set of independent executing agents, where each agent is further refined into a set of behaviors. Each behavior is modeled with a high level Petri net, called behavior net. The overall behavior of an agent is modeled by an agent through composing individual behavior nets. Finally, the overall system behavior is modeled by a system net through integrating individual agent nets incrementally. Simulation and model checking can be performed on individual behavior nets, agent nets, and the final system net. The resulting system net is systematically mapped to behavior programs in Java, which are enhanced and extended with domain specific functionality. A set of property patterns based on behavior program is developed, which are used to generate runtime monitors to check behavior program executions. We demonstrate our approach using a multi-car parking system.**

Keywords - cyber physical systems; behavior programming; 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, improving and integrating existing technologies including software engineering processes, design patterns, formal verification, and simulation provides a potential solution [2].

In [3], we provided a concrete framework to realize the ideas in [2], where a model driven approach from high level Petri nets to Java programs was presented. 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. An agent oriented modeling approach is used to capture CPSs at a high abstraction level where meaningful computational components and physical processes with independent behaviors are viewed as agents and modeled using individual high level Petri nets. An aspect oriented approach is used to incrementally integrate system components represented using individual agent nets into a complete system net. Agent nets and the system nets are

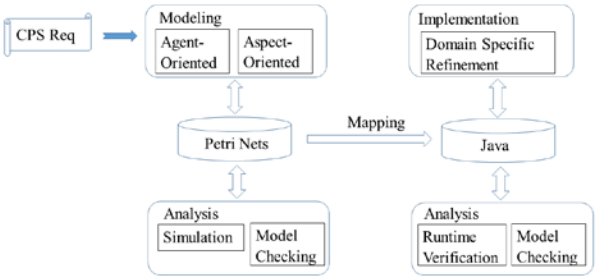

Fig. 1 – A framework for developing cyber physical systems

analyzed through simulation as well as model checking. The above modeling and analysis techniques are supported by tool chain PIPE+ [4] and SPIN [5]. 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.

In this paper, we enhance the above framework with an additional behavior-oriented modeling approach that complements the agent-oriented modeling approach. While the agent-oriented approach provides a higher level system decomposition driven by concurrency, in which physical devices and computational processes are abstracted and modeled as agents; the behavior-oriented approach offers a finer system decomposition driven by unique non-deterministic behaviors within each physical device or computation process. Behaviors provide a more intuitive, natural, and concrete way to incrementally understand and develop CPSs. This systematic and multi-level incremental approach helps us to better understand and develop CPSs. A new set of runtime monitoring property patterns based on behavior programming are developed to ensure the dependability of the implementation.

DOI reference number: 10.18293/SEKE2018-004

Our new contributions include: (1) a systematic approach for modeling and analyzing CPSs, (2) a new behavior-oriented approach to incrementally model and analyze CPSs, (3) a pattern based translation method for generating behavior programs from behavior nets, and (4) a set of behavior based runtime monitoring property patterns. Our systematic approach is demonstrated through a multi robotic car parking system.

II. Cyber Physical System Modeling

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 [6] and special graphical modeling languages such as actor-oriented MoC [7]. High level Petri nets [8] are well suited to model the complex behaviors of CPSs, especially combined with well-established software engineering approaches such as agent-oriented approach and aspect-oriented approach [3]. However most existing techniques only provide very general guidelines and lack fine grained rules. Behavior based modeling [9] provides an intuitive, natural, and concrete way to incrementally understand and develop CPSs. In the following sections, we describe a systematic approach in modeling and analyzing CPSs, which consists of three levels – a behavior-oriented approach for modeling the internal behaviors of an agent; an agent-oriented approach to model the components of a system, and an aspect-oriented approach to synthesize the whole system. We demonstrate our approach using a multi robotic car parking system.

A. Modeling Individual Behaviors

In behavior-oriented modeling, the unique behaviors of a physical device (sensors and actuators) or a computation process are identified and abstracted from the requirement specifications and are modeled with individual high level Petri nets called behavior nets that interact with external environments. Specifically, we provide the following general and simple design pattern of a behavior net shown in Fig. 2:

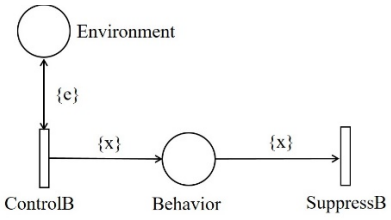

Fig. 2 – A Behavior Net Pattern

Where place *Behavior* models a behavior based on a uniquely identified behavior, which can be further refined by replacing the place with a more detailed net if needed. The type of place is a power set of a Cartesian product to capture multiple instances of behaviors of different objects, where each object has a unique identifier and other fields to capture important information. Place *Environment* models the external environment that can be detected by an object. Transition *ControlB* defines a condition to start the behavior and transition *SuppressB* models the end of the behavior. An additional incoming arc to transition *ControlB* will be created to indicate the selection of the behavior when the behavior net is integrated into an agent net and an outgoing arc from transition *SuppressB* will be created to integrate the behavior net.

We demonstrate our behavior-oriented approach in modeling a multi robotic car parking system. Each robotic car has two motors, two color sensors, and two IR (infrared obstacle) sensors. The color sensors are amounted on both front sides of a robotic car and are used to detect driving lane, two garage entrances, one exit, and four parking lots (all marked with unique colors). The IR sensors are amounted at the left front side (for left turning only) and the front of a robotic car to detect obstacle such as another robotic car or garage wall. Each robotic car has the following unique scenarios: (1) detecting an entrance using color sensors, (2) detecting the exit using color sensors, (3) searching for lane using color sensors, (4) detecting the lane using color sensors, (5) detecting obstacles for collision avoidance using IR sensors, (6) detecting a vacant parking lot using color sensors and IR sensors, (7) entering a parking lot using IR sensors, (8) leaving a parking lot using IR sensors, and (9) exiting the garage. Some of the above scenarios can be combined to form a more complex scenario such as searching and detecting lane, and some scenario such as detecting an entrance can be split into two specific scenarios – detecting entrance one and detecting entrance two. A screenshot of the behavior net search for lane (3) created in PIPE+ is shown in Fig.3. Since there is only one lane, place *Lane* holds only one token modeling the lane. Place *SearchLane* is a power set of tokens that model individual cars (4 cars in this system). Each car has a structured type of 3 string fields, the 1st field denotes car identifier, the 2nd field models a communication socket (not used in the model), and the 3rd field records a car status that is used to keep track behavior history and to select follower up behaviors.

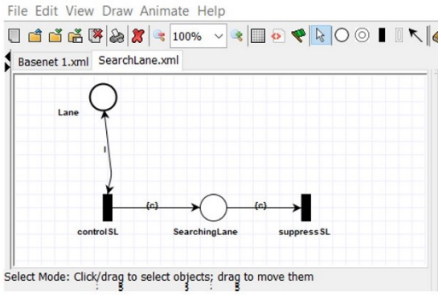

Fig.3 – Behavior Net of Searching Lane

B. 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 an agent net that has its own independent reactive and/or proactive behavior interacting with the external environment. Based on the behavior-oriented modeling, an agent net is obtained by integrating a set of

remarkably simple behavior nets through a place *Arbitrator*, which is used to control the selection of individual behaviors within an agent. The complete agent net of a single car after integrating all 12 behavior nets (the four parking lot behaviors are separately modeled) is shown in Fig. 4, which contains 22 places, 26 transitions, and over 60 arcs (many are bidirectional).

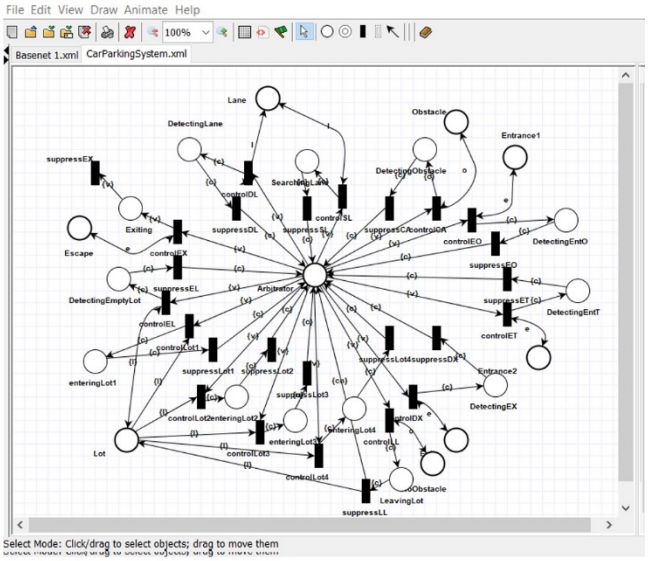

Fig. 4 – The Complete Agent Net of A single Car

C. Modeling the Whole System

The overall system net is obtained by integrating individual agent nets 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 [8] 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. In this multi robotic car system, all the cars have the same behaviors and they do not communicate with each other. Thus the overall system net has the same structure as that of a single car. However multiple tokens with unique identifiers representing different cars are added to the place *Arbitrator* as part of initial marking.

III. CYBER PHYSICAL SYSTEM ANALYSIS

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. High level Petri nets are executable and thus support simulation of hybrid system models. 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. Our tool PIPE+ supports simple reachability analysis and model checking using SPIN in addition to simulation.

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. There are currently two translation schemes:

(1) Translating each transition as an inline function consisting a part realizing the precondition – checking the enabling condition, and another part capturing the post-condition – transition firing. Each transition is non-deterministically selected in a loop within a single Promela process;

(2) Translating each transition as a Promela process. Each translation schema has its own advantages and disadvantages. The first one seems more efficient in checking safety properties, while the second one can be used to check liveness property using strong fairness assumption.

The translated Promela model after adding linear time temporal logic specifying properties is model checked using SPIN. 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.

With regard to behavioral programming, we can define many generic safety and liveness property patterns using linear time temporal logic, and then instantiate the patterns using concrete behaviors and check them using SPIN model checker. Some generic property patterns include (where B, B1, B2 denote place names representing different behaviors, x and y denote symbolic tokens):

(1) $\Diamond B(x)$ (G1)

This liveness property states that a behavior B will eventually active;

(2) $\Box(B(x) \rightarrow \Diamond !B(x))$ (G2)

This liveness property states that an active behavior B will eventually terminate;

(3) $\Box!(B1(x) \wedge B2(y))$ (G3)

This safety property states that two behaviors B1 and B2 cannot be active at the same time due to the sequential nature of behavioral programming;

(4) $\Box(B1(x) \rightarrow \Diamond B2(y))$ (G4)

This liveness property states that a behavior B1 leads to behavior B2.

More sophisticated properties can be defined such as there is one particular behavior in between two other behaviors.

Here we provide our model checking results of the concrete properties for in the car parking system.

$\Diamond$SearchingLane(v1,v2,found) (C1)

$\Box$(SearchingLane(v1,v2,found) $\rightarrow$ $\Diamond$!SearchingLane(v1,v2,found)) (C2)

$\Box$!(DetectingEntO(v1,v2,ent1)$\wedge$SearchingLane(v1,v2,found))

(C3)

[](DetectingEntO(v1,v2,ent1) →
<> SearchingLane(v1,v2,found)) (C4)

Since the concrete values of symbolic variables v1 and v2 are not used in checking the above properties, we use bit type to abstract their types to reduce the number of states and instantiate their values according to the initial marking. Furthermore, these properties are about the same car, we can restrict our initial marking to one car in place Arbitrator. Also checking liveness property (C1) can be done more effectively in SPIN by finding a counter example of its negation:

[] !SearchingLane(v1,v2,found) (C1*)

With the above abstraction and reduction to the resulting Promela model and using the –DBITSTATE storage option in SPIN, we have checked all of the above properties as shown in Table II.

Property	Satisfied	Depth	Stored States	Time
C1*	No	128	201	5(ms)
C2	Yes	763	554077	895(ms)
C3	Yes	611	550966	923(ms)
C4	Yes	140	540451	897(ms)

IV. 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 [10], 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. In [3], we presented a model driven approach to realize our high level Petri net models, which provided 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 is 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. Different from [3] where each agent net was mapped to a thread in Java, this paper maps each behavior net into a behavior program [11] that includes 3 template methods: boolean takeControl(),void action(), and void suppress().

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

(1) A behavior program is generated for each behavior net, in which 3 methods are created corresponding to transitions *controlB* and *suppressB*, and place *Behavior* shown in Fig.3. The body of each method is empty and requires manual refinements. The constraint of the transition is attached as comments for ensuring the correct implementation;

(2) A behavior object class is created, which is to be manually refined according to the application domain;

(3) The main program is created based on the initial system net with a single place *Arbitrator*, and includes the definition of an arbitrator object and the instantiations of all the behaviors. In a behavior model, the control flows between behaviors are enforced through a data field in a behavior object. In behavior programming, the control flows are based on the priorities of the behaviors according to their appearances in the behavior array from low to high. As a result, manual reordering is needed to ensure correct control flows. Additional manual refinements are necessary to make the program complete;

(4) A Java project is created to include the above code files.

The above code generation rules are implemented in PIPE+. A Java project is automatically generated from net by selecting BehaviorProgram under Export button in File pulldown menu.

The following code segments are automatically generated by PIPE+ from the multi robotic car parking system model:

```
package parkingsystem.behavior;
public class DetectEntryOne implements Behavior {
  private boolean suppressed = false;
  public boolean takeControl() {
  //TODO:: to be implemented
  //Pre: v.field3=waiting
  //Post: c.field1 == v.field1 && c.field2 ==
  v.field2 && c.field3 == ent1
  }

  public void action() {
  //TODO:: to be implemented
    suppressed = false;
  }

  public void suppress() {
  //TODO:: to be implemented
  //Pre:
  //Post:
    suppressed = true;
  }
}
...
package parkingsystem.object;
public class Robot {
  //TODO:: to be implemented
}
package parkingsystem.main;
public class Main {
//TODO:: to be implemented
   public static void main(String[] args) throws
Exception {
    //TODO:: to be refined
    Robot robot = new Robot();
    ...
    Behavior detectEntryOne = new DetectEntryOne();
    ...
    Behavior[] behavior_array =
    { ...
      detectEntryOne,
      ...
    };
    Arbitrator arbitrator = new
Arbitrator(behavior_array);
    ...

  }
}
```

The actual LEGO car parking system implementation refines the above code templates with domain specific functions imported from lejos.robotics and lejos.hardware packages.

V. Runtime Verification

Runtime verification is a lightweight formal approach to detect violation of properties during the execution of a system. It complements the formal methods applied to system models such as model checking and theorem proving by detecting errors either introduced in the process of model implementation or undetected at model level due to the abstraction of models and limitations of formal methods.

Runtime verification was adopted in the multi-car parking system to ensure dependability at implementation level. In our work, properties are specified using linear temporal logic (LTL) formula built from the atomic propositions defined using events written in JavaMop [12]. Monitors are generated from LTL formulas and woven into system implementation as aspects using AspectJ [13]. This ensures the independence of system implementation from monitor – the runtime verification code.

To monitor systems developed with the behavior-oriented approach, several major events are defined for each behavior: *takecontrolT*, *takecontrolF*, *actionR*, *actionE*, and *suppress*. Event *takecontrolT* occurs whenever the method takeControl() in the behavior-generated code is executed and returns true. Event *takecontrolF* is similar to *takecontrolT* except the value false is returned. Event *actionR* occurs whenever the method action() of the behavior becomes active, while event *actionE* occurs whenever the method action() is executed. Event *suppress* occurs whenever the method suppress() of the behavior is executed. To distinguish these events defined for different behaviors, behavior name is added in the front of these events. To make the formula more concise, we use the behavior name only to represent the event *actionE*. The following JavaMop code shows an event definition for the behavior DetectingEntranceOne. Event definitions for other behaviors are similar.

```
event DetectingEntranceOne_takecontrolT
after(DetectingEntranceOne bhv) returning(boolean b):
execution(public boolean
DetectingEntranceOne.takecontrol()) && this(bhv) &&
condition(b)
{       //code to be executed when the event occurs;
}
```

In JavaMOP, the properties to be monitored are specified as LTL formulae using defined events, and are evaluated against an execution trace abstracted as a sequence of events. As event definition implies, an event represents the occurrence of a concrete action, typically the entry or exit of an action, which can be calling a method, executing a method, or updating a primitive variable. Events are atoms when used in a LTL formula. In a sequence of events, an event atom is true only when it matches the corresponding event occurrence.

Due to the competition and sequential nature of behaviors in the multi-car parking system, we divided the properties to be monitored into two groups: properties of the arbitrator, and properties on the temporal relations among different behaviors. The former properties ensure the correctness of the arbitrator. The latter properties ensure the correct behavior order from the system specification.

Property patterns of the arbitrator include:

(A1) A behavior b becomes active only if it is selected by the arbitrator: [] (b_actionR → <*>b_takecontrolT), where <*> is the past temporal operator previously;

(A2) A behavior b will become active: <> (b_actionR);

(A3) Current behavior b will eventually terminate if the arbitrator calls its suppress() method: [] (b_suppress → <> b_action);

(A4) Two behaviors b1 and b2 cannot be active at the same time: [] ((b1_actionR → !b2_actionR U b1) ∧ (b2_actionR → !b1_actionR U b2), where U is the until operator;

(A5) If both behaviors b1 and b2 are ready to become active, the arbitrator always picks b1 assuming b1 has a higher priority over b2: [] ((b2_takecontrolT → <*>b1_takecontrolF).

Properties (A2) and (A4) correspond to the generic properties (G1) and (G3) at the model level. Property (A1) involves some past concept that cannot be represented in SPIN model checker. Property (A5) with regard to behavior priorities is dealt with using an attribute of a token at the model level. Properties (A2) and (A3) are liveness properties, therefore cannot be verified at runtime since the monitor doesn't know when "the good thing" will happen. To effective monitor these liveness properties, a timeout event is introduced to make these properties bounded (thus turning them into safety properties). For example (A2) becomes (A2'): <> (!timeout U b_actionR).

Property patterns relating different behaviors include:

(B1) An event e1 occurs at most once before another event e2: [](e1 →○(!e1 U e2)), where ○ is next operator;

(B2) Whenever an event e1 occurs, another event e2 should occur later: [](e1 → !timeout U e2), which corresponds to generic property (G4) at the model level;

(B3) Whenever an event e1 occurs, another event e2 must occur before it: [](e1 → <*> e2);

(B4) An event e1 should never occur before the first occurrence of another event e2: !e1 U e2;

(B5) An event e1 should never occur after another event e2: [](e2 → []!e1);

(B6) An event e1 should never occur between event e2 and event3: [](e2 →!e1 U e3).

An experiment was conducted to verify the effectiveness of monitoring behavior of the arbitrator and temporal orders of multiple behaviors. In the experiment, we have two LEGO cars running the same piece of code in a parking garage with two different entrances and one exit. The LEGO cars can enter the parking garage through the same or different entrances. When entering the garage through entrance one 1, a car can park at lot 1, 2, 3, or 4; otherwise, the car can only park at lot 3 or 4. To simplify the situation, there is only a one-way lane without

circle. Both cars monitor the same set of properties including 5 concrete A type and 6 concrete B type properties. During the experiments, some properties failed due to the unreliable nature of color sensors. We also noticed the priorities of the behaviors have a major impact on the overall system performance: frequent checking of the readiness of a high priority behavior has a huge negative impact on the performance of the robotic cars. Thus it is important to design the monitors carefully to reduce the performance penalty.

VI. Related work

Our CPS development approach covers many research topics including system modeling, system analysis using simulation and model checking, model driven development, and runtime verification. Our main contribution is a systematic CPS development approach by integrating successful existing technologies. Thus we only discuss several most relevant CPS development methodologies.

In [14], a general model-based design methodology for CPSs was proposed. A cook book process was defined, which contains ten general steps in developing a CPS. The process was demonstrated through a bouncing ball example. This methodology is generic and independent of a particular formal model, and thus is not supported by a tool chain.

In [7], an actor-oriented design approach was described for modeling CPSs. Actors are used to model components that communicate through ports. This design approach adopts a multiple model view and is supported by the modeling and simulation environment Ptolemy for heterogeneous systems. Several experimental component modeling modules have been developed, including discrete events (DEs), continuous time (CT), finite state machines (FSMs), synchronous reactive (SR), process networks (PNs), and data flow models. Hybrid system models are obtained by hierarchically composing CT models with discrete models such as FSM or DE. Although this approach provides powerful system modeling and analysis capabilities, it does not cover code generation and code level analysis.

In [15], a foundational framework, called VeriDrone, for reasoning about CPSs at all levels from high-level models to C code was presented. VeriDrone becomes a built-in library of the theorem prover CoQ and enables CoQ users define and verify CPS related properties. This work focuses on formal analysis of CPS, but does not address how to model and design CPSs.

In [3], we developed an overall framework for developing CPSs. This framework is model driven and based on a single formalism – high level Petri nets. This paper extends our framework in [3] with the following new results: (1) a new behavior-oriented approach for modeling internal behaviors of within agents, (2) a net pattern for modeling individual behaviors, (3) a set of property patterns for specifying behaviors, (4) a new translation scheme for generating behavior programs from high level Petri nets, and (5) a set of runtime property patterns for monitoring behavior program execution.

VII. Conclusion

This paper presented a systematic approach 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. This approach supports a multi-level incremental modeling consisting of behavior-oriented approach for capturing the internal behaviors within agents, agent-oriented approach for system decomposition, and aspect-oriented approach for system composition. The resulting models are analyzed using simulation and model checking to detect early design problems. A translation method for generating general behavior program structure from high level Petri net models is provided. The resulting general behavior program is manually refined with domain specific code 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. We are currently working on genetic algorithms to further automate the code refinement process. Implementation level quality assurance is carried out using runtime verification. We have developed a set of property patterns based on behavior programming. We demonstrated our approach thorough a multi robotic car parking system. We are currently working on 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] X. He, Z. Dong, H. Yin, Y. Fu: “A Framework for Developing Cyber Physical Systems”, Proc. of the 29th International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, July 5-7 , 2017.

[4] D. Alam and X. He: “A Method to Analyze High Level Petri Nets using SPIN Model Checker”, Proc. of the 29th Int’l Conf. on Software Engineering and Knowledge Engineering, Pittsburgh, 2017.

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

[6] 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.

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

[8] 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.

[9] D. Harel, G. Katz, R. Marelly, and A. Marron: “First Steps towards a Wise Development Environment for Behavioral Models”, International Journal of Information System Modeling and Design, vol. 7, no. 3, July-September, 2016.

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

[11] http://www.lejos.org/nxt/nxj/tutorial/Behaviors/BehaviorProgramming.htm.

[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] J. Jensen, D. Chang, and E. Lee: “A Model-Based Design Methodology for Cyber-Physical Systems”, Proc. of the First IEEE Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems (CyPhy), Istanbul, Turkey, 2011.

[15] G. Malecha, D. Ricketts, M. Alvarez, and S. Lerner: “Towards Foundational Verification of Cyber-physical Systems”, 2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS), 2016.