Requirement Modeling and Validation Using BIP Framework: An Airplane Engine Control Software Case Study

Xudong Tang[1], Qiang Wang[2], Weikai Miao[1], and Joseph Sifakis[4,5]

[1] School of Software Engineering, East China Normal University, Shanghai, China
[2] Chinese Academy of Military Science, Beijing, China
[3] School of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
[4] Verimag, Université Grenoble Alpes, France

Abstract. In this case study, we apply architecture-based rigorous system design approach to the control software of the airplane engine controller. Rigorous system design refers to a formal model-based process that leads from requirements to correct implementations. Architectures are a means for ensuring global coordination properties of system models and thus, achieving design correctness by construction. We implement the architecture diagrams for airplane engine control software in the Behavior-Interaction-Priority (BIP) component-based framework. We show how to obtain a system model by applying architectures to a set of atomic components. We also perform deadlock freedom analysis of the resulting system model by using the BIP tool-set and additionally validation analysis of our approach by using nuXmv model checker to verify that the safety properties enforced by the architecture are indeed satisfied.

1 Introduction

The Power Level Angle (PLA) signal processing system is part of the Full Authority Digital Engine Control System (FADEC), which provides engine control during flight to achieve stable and transient engine characteristics. In the engine control system, the PLA signal processing system is not only a typical system instance of continuous quantity processing, but also the pivotal section assisting pilots controlling the aircraft through the power levers. For PLA signal processing system, we emphasize on the following questions: 1) whether the input signal is correctly processed by the system for one period. 2) whether the system can keep the output correct for a long period of time. 3) whether each part of the system can respond a sudden exception of input in real time. In this paper, we are going to use a simplified PLA signal processing system of real application as a case study.

The purpose of PLA signal processing system is to convert the obtained throttle stick signal value into the original throttle stick angle value and judge it. If it is judged that there is no fault, then it will return the corresponding value of the PLA signal after processing and set a fault signal to invalid, representing that the system has no PLA signal fault in current period. Otherwise it will return the fault information and set the fault signal to valid.

Fig.1 shows the system structure of our case. There are two diagnosis modules, a sensor data unit and a data conversion unit. The sensor data unit task is connected to the BIT diagnosis unit and the data conversion unit direactly while the latter two units are connected to the Extremum & Slope diagnosis unit. BIT diagnosis module consists of three diagnosis units and a decider unit and, analogously, the Extremum & Slope diagnosis module consists of two diagnosis units and a decider unit. Besides BIT diagnosis module, the calibration conversion unit is also driven by sensor data unit task and provides signal for Extremum & Slope diagnosis module.

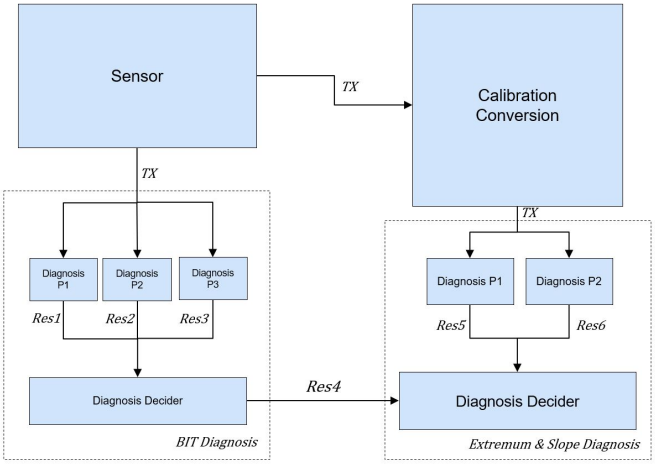

Fig. 1: The system structure

In order to verificate the PLA signal processing system, we use the BIP framework. To help validating the system, we apply NuXmv tool chain on the NuXmv system model which is transformed from the BIP model. In this case, we have to write a series of properties in CTL or LTL specification patterns. To avoid mistake on transforming requirements to specifications and fully specify the PLA signal processing system, we sort out the requirements and write monitors adding to the BIP model. Fig.2 shows the work flow according to our two different system designs.

The rest of the paper is structured as follows. In Section **??** and Section **??**, we introduce the preliminaries and the BIP modeling language. In Section **??** and Section **??**, we review the most related works and draw some conclusions and outline directions for future work.

2 BIP framework

In this section, we present the BIP framework. In BIP framework, a model is constructed by composing three layers of modeling, that is behavior, interaction and priority. More information about the underlying concepts and the operational semantics of BIP can be

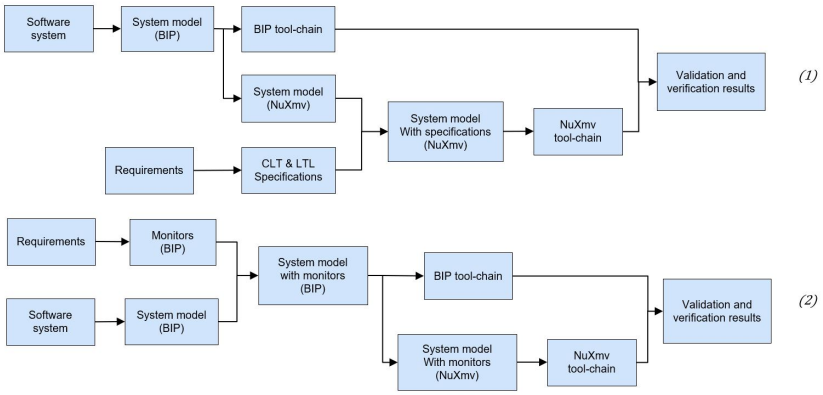

Fig. 2: (1) Work flow of system model with specification
(2) Work flow of system model with monitors

found in []. In particular, BIP provides separation of concerns between behavioral and architectural aspects in modeling.

The Behavior-Interaction-Priority (BIP) framework [1] is proposed as a component-based system design framework, which comes with a formal language with well-defined semantics and a tool chain supporting rigorous system design process. The BIP language offers a three-layered modeling mechanism for constructing complex system behavior and architectures [4], i.e., Behavior, Interaction, and Priority. Behavior is characterized by a set of components, which are formally defined as automata extended with linear arithmetic. Interaction specifies the multiparty synchronization of components, among which data transfer may take place. Priority can be used to schedule the interactions or resolve conflicts when several interactions are enabled simultaneously. The key insight underlying this three-layered modeling mechanism is the principle of separation of concerns, that is, system computation is captured by a set of components, and system coordination is modeled by interaction and priority.

An atomic component is described by a finite-state automata extended with variables and ports, where variables are used to store local data, and ports form the interface of the component. Ports are used to define the interaction and communication with other components, and may be associated with variables. Moreover, ports also serve as the labels of the automata transitions. States denote control locations at which the components await for interaction. A transition is a step, labeled by a port, from a control location to another. It may be associated a guard (i.e. a boolean expression) and an action (i.e., a function over variables). In BIP, actions are written in C/C++. An example BIP component is shown in Fig.**??**.

Composition of components is defined by interactions. Essentially, interactions express synchronization constraints between transitions of the composed components. Interactions are described in BIP as the combination of two types of elementary protocols: rendezvous to express strong symmetric synchronization and broadcast to express triggered asymmetric synchronization. Interactions are structured in connectors, that is,

sets of ports plus additional information. Within connectors, every port is typed either as synchron or as trigger. Trigger ports are used to initiate broadcast, that is, any subset of ports containing at least one trigger port denote a valid interaction of the connector. Rendezvous synchronizations are obtained on connectors where all the ports are synchrons. For such connectors, the only valid interaction is the maximal one, that is, the whole set of ports. Finally, connectors provide mechanisms for dealing with data associated to (ports of) interacting components. Every interaction has a guard, that is, an enabling condition and an action, that is, an update (data transfer) function, operating on data associated to ports participating in the interaction. Circles (resp. triangles) denote synchron (resp. trigger) ports.

Priorities are used to filter amongst possible interactions. They are expressed as conditional priority rules between two interactions. Whenever the condition (on the state of the system) holds, and if the two interactions are enabled at the same time, then only the high priority interaction is allowed for execution. In practice, priorities steer system evolution so as to meet performance requirements e.g. to express scheduling policies.

The BIP toolbox includes a rich set of tools for modeling, transformation, analysis and code generation of models. The toolbox provides a dedicated modeling language for describing BIP components. It is a user-friendly textual language which provides syntactic constructs for describing components conforming to the formal framework. The BIP language leverages on C-style expression declarations, and provides additional syntactic constructs for defining component behavior, specifying the coordination through connectors, and describing the priorities. The toolbox also provides front-end tools for editing and parsing of BIP descriptions, as well as for generating intermediate models, followed by code generation (in C++). Intermediate models can be subject to various model transformations focusing on construction of optimized models for respectively sequential and distributed execution. Back-end tools include specific runtime for efficient execution on machines with different characteristics (e.g., real-time, mono/multi-thread, single/multi-core). Besides, various analysis tools are provided to perform validation and verification of BIP models, such as deadlock analysis using the D-Finder tool [2], reachability analysis using the model checker [3] and the runtime verification tool [].

3 Formal system model in BIP

In this section, we introduce an airplane engine control software case study. For presentation simplicity, we choose one of its subsystems, the PLA signal processing system, as representative. We will give an informal description, focusing on its functional structure, properties and conditional constraints. Then we give the formal design model in BIP. Finally, we provide model verfication and validation.

3.1 Overview of the system model

We construct the BIP model of the PLA signal processing system through the composition of atomic and composite components with BIP connectors. The block diagram for

our system is shown in Fig.1, it consists of a Task Unit and a Signal Processing Unit. The Signal Processing Unit consists of atomic component c1, composite components p1 and c2. In the Task Unit, there is an atomic component t1, which produce signal value for Signal Processing Unit.

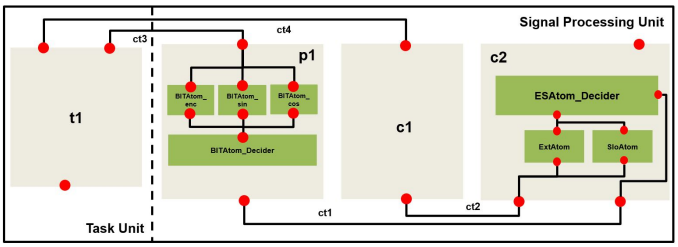

Fig. 3: System Architecture

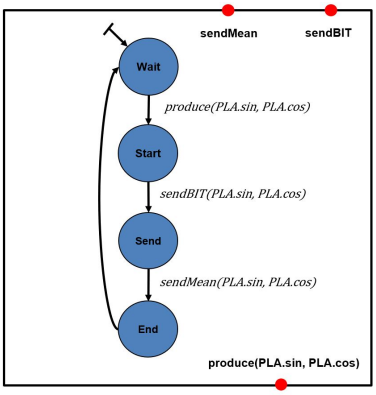

Fig. 4: The Task Component

The behavior of atomic component t1 is shown in Fig.2. It is used as simulator of the input stimuli. In the graphical notation, the atomic is initialized to *Wait*, and the following three transitions are driven by corrensponding events: *produce*, *sendBIT* and *sendMean*, which represents three corrensponding ports in the atomic. For instance, whenever the current state is at *Start*, the port *sendBIT* will be activated and the transition will be executed once the event *sendBIT* is done.

3.1.1 BIT Diagnosis module In our design, the BIT Diagnosis module, Calibration Conversion module and Extremum/Slope Diagnosis module are three subsystems of PLA signal processing system. The BIT Diagnosis module is shown in figure 3. In the

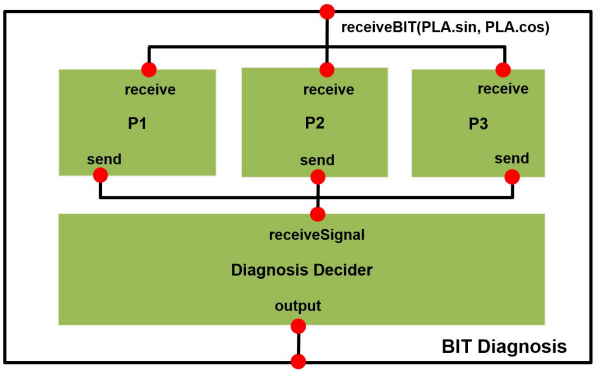

Fig. 5: Architecture of BIT Diagnosis

graphical notation, the three atomic component instances *p1*, *p2* and *p3* receive BIT signal through port *receiveBIT*, a connector distributes the signal to these components and another connector merges their output and sends it to the an instance of *Decider* component. The BIP description of the definition and instantiation of *BIT Diagnosis* is shown below:

compound *BITCompound_P1*
  component *BITAtom_enc enc*
  component *BITAtom_sin sin*
  component *BITAtom_cos cos*
  component *BITAtom_Decider d*

  connector *ThreeToOne o(enc.sendBIT,sin.sendBIT,cos.sendBIT,d.receiveSignal)*
  connector *Merge m(enc.receiveBIT,sin.receiveBIT,cos.receiveBIT)*

  export port *m.Merged_Signal* **as** *receiveBIT*
  export port *d.output* **as** *sendBIT*
end

We define the two ports in *BITCompound_P1* as export to give them an external property which enables them to be triggered by other components.

connector type *ThreeToOne(PLAPort_t1 r1, PLAPort_t1 r2, PLAPort_t1 r3, PLA-Port_t3 r4)*
  define *r1 r2 r3 r4*
  on *r1 r2 r3 r4* **down** {*r4.a=r1.a; r4.b=r2.a; r4.c=r3.a;*}
end

connector type *Merge(PLAPort_t2 r1, PLAPort_t2 r2, PLAPort_t2 r3)*

data int *mysin,mycos*
export port *PLAPort_t2 Merged_Signal(mysin,mycos)*
define *r1 r2 r3*
on *r1 r2 r3* **down** {*r1.a=mysin; r2.a=mysin; r3.a=mysin; r1.b=mycos; r2.b=mycos; r3.b=mycos;*}
end

In our system, atomic component is considered as the smallest unit of the architecture, while in the combination of several components, such as our BIT Diagnosis module, we have to ensure that the output of three components can simultaneously reach the connoector, otherwise it will fall into deadlock status. To avoid this problem, the three atomic components are designed homogeneous, we take one of these three components, *BITAtom_enc p1*, as an example, its ports and behavior is shown below:

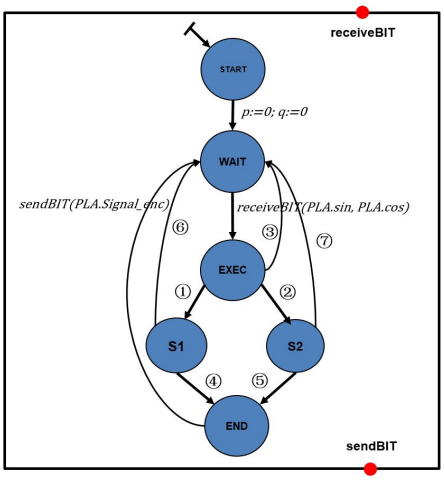

Fig. 6: BIT Diagnosis P1

During execution, we consider a WAIT to WAIT transition set as a cycle. In each cycle, we ensure that an *sendBIT* event will be activated to avoid the system falling into deadlock status. With the combination of connector, *P1* gives a synchronous output together with other two atomic component instances. We define two variables p and q as counters to control the vary of signal. Their constraint indicates that for every continuous five cycles, the system does a setting and gives a corresponding output through port *sendBIT*.

As shown in Fig.5, the BIT Diagnosis Decider is responsible for integrating the signals output by three BIT Diagnosis. We also produce a WAIT to WAIT transition set, *receiveSignal* as input event and *output* as output event. We define a bool type variable *PLA.Signal* as the BIT fault signal for output.

3.1.2 Calibration Conversion module

Table 1: Transitions of P1

Transition	Guard	Action
1	PLA.sin $<$ 500$\wedge$ PLA.cos $<$ 500	$p := p + 1; q := 0$
2	PLA.sin $>=$ 500$\wedge$ PLA.cos $>=$ 500	$p := 0; q := q + 1$
3	PLA.sin $>=$ 500$\oplus$ PLA.cos $>=$ 500	$p := 0; q := 0$
4	$p >= 5$	PLA.Signal_enc: $= 1$
5	$q >= 5$	PLA.Signal_enc: $= 0$
6	$p < 5$	null
7	$q < 5$	null

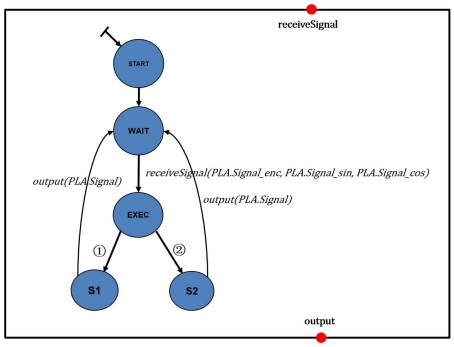

Fig. 7: BIT Diagnosis Decider

The Calibration Conversion module collects the signal from signal collector and converts it to the corresponding engineering value by means of a calibration curve and an index table. This module is designed as an atomic component c1 in the BIP model, its ports and behavior is shown below:

It has been introduced that every atomic component can be converted to its corresponding BIP description. The BIP description of the definition of *Calibration Conversion* is shown below:

Table 2: Transitions of Decider

Transition	Guard	Action
1	PLA.Signal_enc $= 0\wedge$ PLA.Signal_sin $= 0\wedge$ PLA.Signal_cos $= 0$	$PLA.Signal := 0$
2	PLA.Signal_enc $= 1\vee$ PLA.Signal_sin $= 1\vee$ PLA.Signal_cos $= 1$	$PLA.Signal := 1$

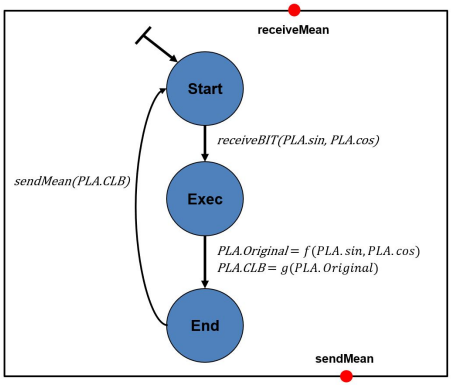

Fig. 8: Calibration conversion module

```
atom CalAtom
   data int mysin,mycos
   data int myangle

   export port PLAPort_t2 receiveMean(mysin,mycos)
   export port PLAPort_t4 sendMean(myangle)

   place START, EXEC, END

   initial to START
      do {myangle = 0;}

   on receiveMean from START to EXEC

   internal from START to EXEC
      do {myangle = mysin * mycos + 1;}

   on sendMean from END to START
end
```

In the BIP description, we define internal data, external ports, serveral states and transitions. Transitions begin with field *on* are external transitions which are driven by external events. Relatively, transitions begin with field *internal* literally internal transitions which are executed as soon as the atomic component reach the state. We define an internal variable *myangle* for output. System calculate its value through calibration curve and index table from two variables *mysin* and *mycos*.

3.1.3 Extremum/Slope Diagnosis module

The Extremum/Slope Diagnosis module is almost the same as BIT Diagnosis module. It picks up signal value, diagnoses them and integrates the results into one variable for output. We also provide the architecture of Extremum/Slope Diagnosis:

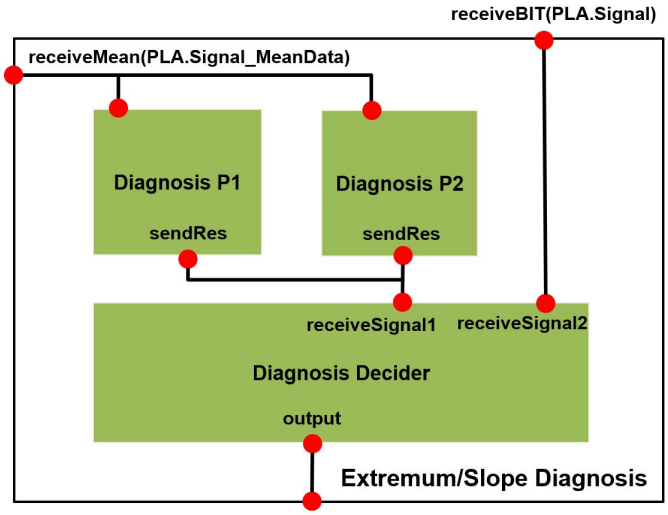

Fig. 9: Architecture of Extremum/Slope Diagnosis

It should be noticed that in Extremum/Slope Diagnosis module, besides signals sent by internal atomic components p1 and p2, the *Decider* also pick up signal from BIT Diagnosis module as input. We give the BIP Description of the Extremum/Slope Diagnosis module below:

```
compound ESCompound_C1
   component ExtAtom enc
   component SloAtom slo
   component ESAtom_Decider d

   connector TwoToOne o(ext.sendRes,slo.sendRes,d.receiveSignal1)
   connector Merge_two m(ext.receiveMean,slo.receiveMean)

   export port d.receiveSignal2 as receiveBIT
   export port m.Merged_Signal_two as receiveMean
end
```

The connector types *TwoToOne* and *Merge_two* do almost the same work as the two connector types used in BIT Diagnosis module except that the connector types used here have only two parameters, not three.

In the entire system, the compound type *ESCompound* will be instantiated as c2, which is introduced in Fig.1.

Till now, we have introduced our system informally and in formal BIP description. To give a formal proof whether our system and BIP model is safety and effectiveness or not, the verification and validation of the system above will produced in the next chapter.

3.2 Model verification and validation

4 Related work

In the BIP framework, DFinder [2] is a dedicated tool for invariant generation and deadlock-freedom detection. DFinder computes the system invariant in a compositional manner: it first computes a component invariant over-approximating the reachable states of each component and then computes an interaction invariant over-approximating the global reachable states. The system invariant is then the conjunction of all component invariants and the interaction invariant. Though being scalable for large system models, DFinder does not handle system models with data transfer, which hampers the practical application of DFinder and of the BIP framework, since data transfer is necessary and common in the design of real-life systems (e.g. message passing). Besides, it is not clear in DFinder how to refine the abstraction automatically when the inferred invariant fails to justify the property.

A compositional encoding of BIP into nuXmv and an efficient instantiation of Explicit Scheduler Symbolic Thread (ESST) framework for BIP have been presented in [3]. The encoding into nuXmv allows one to exploit the state-of-the-art verification techniques to verify BIP models. The ESST based technique encodes the components as preemptive threads with predefined primitive functions and utilizes a dedicated stateful BIP scheduler to orchestrate the abstract reachability analysis of the components. The scheduler interacts with components via primitive functions, and also respects BIP operational semantics. Moreover, partial order reduction techniques are applied in the scheduler to reduce the states of scheduler.

In [5], we present a lazy predicate abstraction technique for BIP, and we also propose a novel reduction technique called simultaneous set reduction, which is further combined with lazy abstraction to reduce the search space of the abstract reachability analysis.

5 Conclusion and future work

References

1. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. Software, IEEE (2011)
2. Bensalem, S., Bozga, M., Nguyen, T., Sifakis, J.: Compositional verification for component-based systems and application. IET Software (2010)
3. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal verification of infinite-state BIP models. In: ATVA (2015)

4. Konnov, I., Kotek, T., Wang, Q., Veith, H., Bliudze, S., Sifakis, J.: Parameterized systems in bip: design and model checking. In: Proceedings of the 27th International Conference on Concurrency Theory (CONCUR 2016). pp. 30–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2016)
5. Qiang, W., Bliudze, S.: Verification of component-based systems via predicate abstraction and simultaneous set reduction. In: Trustworthy Global Computing, pp. 147–162. Springer (2015)