Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 943094, 8 pages
http://dx.doi.org/10.1155/2014/943094

Research Article

Modeling a Heterogeneous Embedded System in Coloured Petri Nets

Huafeng Zhang,[1] Hehua Zhang,[2] Ming Gu,[2] and Jiaguang Sun[2]

[1] *School of Computer Science, TNList, Tsinghua University, Beijing 100084, China*
[2] *School of Software, TNList, Tsinghua University, Beijing 100084, China*

Correspondence should be addressed to Huafeng Zhang; ron.huafeng@gmail.com

Received 4 February 2014; Accepted 28 February 2014; Published 31 March 2014

Academic Editor: X. Song

Embedded devices are everywhere now and, unlike personal computers, their systems differ in implementation languages and behaviors. Interactions of different devices require programmers to master programming paradigms in all related languages. So, a defect may occur if differences in systems' behaviors are ignored. In this paper, a heterogeneous system which is composed of two subsystems is introduced and we point out a potential defect in this system caused by an interface mismatch. Then, a state based approach is applied to verify our analysis of the system.

1. Introduction

Embedded devices are everywhere from factories to living rooms and embedded development becomes a new trend in current startup teams. According to the physical carrier of their system, traditional embedded devices can be divided into different domains such as DSP, FPGA, ARM, and PLC. Different carriers mean different programming languages and styles which relate closely to the behavior of the system on the device. For example, the VHDL programming language is widely used in the embedded system's implementation on FPGA board and the common behavior of such systems is synchronous reactive and data-flow oriented [1]. An embedded system on ARM board mostly adopts the C programming language which exploits the power of interrupts and threads asynchronously [2]. There are good engineering practices to follow for developers in such application domains. However, as the rapid evolution of embedded devices, embedded systems confront more and more complex usage scenarios with new problems to be resolved [3]. One of these problems is incompatible programming paradigms between heterogeneous developing techniques which today's embedded systems utilize [4]. A typical embedded-C programmer does not know much about how to write a piece of good VHDL program although he may learn VHDL's grammar somehow and neither does a VHDL programmer know much about the embedded-C stuff. The lack of knowledge in other's programming domain leads to misunderstandings of assumptions and guarantees requirements between interfaces of heterogeneous embedded systems. A trend in handling this heterogeneous development work flow is unifying the programming language used by FPGA, ARM, and DSP developers [5], but it takes efforts to learn a brand-new language even for an experienced developer. Another approach is model based development [6–8]. The developer builds a unified model for the whole system and automatically generates dedicated code for each part according to their specific programming paradigm. The model can be analyzed and verified to guarantee its reliability and enhance its performance before the code generation process [9, 10].

Serious defects between interactions may be undiscovered until on-board tests. The longer the defects go unnoticed, the more time will be spent on them. There are some testing based methods targeting this problem [11]. However, they are not sufficient to guarantee the system's correctness. Analysis based methods [12, 13] can also alleviate the pain in embedded system development.

In this paper, we investigate a case of a heterogeneous embedded system in a real engineering practice and build a model for the system using Colored Petri Nets (CPN) [14] modeling language. Then, we reveal how a normal subsystem

on ARM board and a normal subsystem on FPGA board together lead to a defective system step by step. Then, the model checking approach will be applied on the model to verify our analysis.

2. Preliminaries

2.1. Petri Nets. Petri nets [15] are a simple and expressive modeling formalism, which allows users to model complex systems in various paradigms.

Petri nets are a tuple of $\langle P, T, A\rangle$, where $P = p_1, p_2, \ldots, p_n$ is a finite set of places, $T = t_1, t_2, \ldots, t_n$ is a finite set of transitions, and $A = a_1, a_2, \ldots, a_n$ is a finite set of arcs, while P, T, and A are pairwise disjoint. Each place contains a set of markers called tokens and the number of tokens in each place can be 0, 1, or more. The distribution of tokens in all places is called the marking of the Petri net which is denoted by μ. The marking μ can be viewed as an n-vector, $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, where $n = |P|$ and each μ_i indicates the number of tokens in their corresponding place $p_i, i = 1, \ldots, n$. The marking of a Petri net defines the net's state and it can only be changed by a transition t which moves tokens from a set of places $I(t)$ to another set of places $O(t)$, where $I(t) \subset P$, $O(t) \subset P$, and $t \in T$. The transition t's transfer of tokens is called firing. When t is fired, it removes a token from each place of $p_{t\_\text{in}_1}, p_{t\_\text{in}_2}, \ldots, p_{t\_\text{in}_n}$ in $I(t)$ and adds a token to each place of $p_{t\_\text{out}_1}, p_{t\_\text{out}_2}, \ldots, p_{t\_\text{out}_m}$ in $O(t)$, where $n = |I(t)|$ and $m = |O(t)|$. For a transition t, $I(t)$ is called the input set of t and $O(t)$ is called the output set of t. A transition t is fired only when every place in its input set has at least one token which can be removed in the process of firing. If a transition t's input set fulfilled this condition under a marking μ of a Petri net, t is enabled and can be fired. Otherwise, t is not enabled and cannot be fired. The input set $I(t)$ and output set $O(t)$ of a transition t can be empty and the two sets may have a nonempty intersection; that is, a place may exist in the input set $I(t)$ and the output set $O(t)$ at the same time. If the input set $I(t)$ is empty, transition t can be fired under any markings. If the output set $O(t)$ is empty, transition t will not add tokens to any place.

For the need of graphical representation, the mapping from a place to a transition is defined as an arc. There is an arc from each place in the input set $I(t)$ of a transition t to t and an arc from each place in the output set $O(t)$. The set of all arcs in a net is a relation on $P \times T$ and $T \times P$, where, for every place p_i in the input set $I(t)$ of a transition t, $(p_i, t) \in A$ and, for every place p_o in the output set $O(t)$, $(p_o, t) \in A$.

A Petri net can be represented in a graphical form by a bipartite graph. In a Petri net graph, eclipses represent places, rectangles represent transitions, and connections between eclipses and rectangles represent arcs. An example of graphically represented Petri net is shown in Figure 1. Definitions of P, T, and A of this net are as follows:

$$\begin{aligned} P &= p_1, p_2, p_3, \\ T &= t_1, t_2, \\ A &= (p_1, t_1), (p_2, t_1), (t_1, p_3), (p_3, t_2). \end{aligned} \tag{1}$$

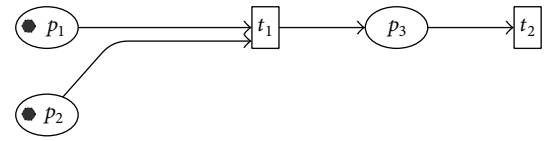

Figure 1: The net in initial state.

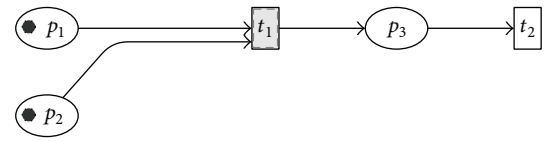

Figure 2: Transition t_1 enabled.

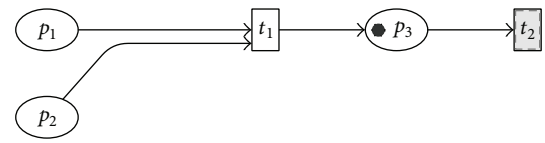

Figure 3: Transition t_2 enabled.

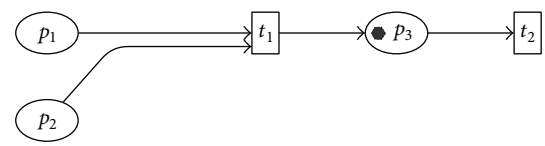

Figure 4: The model in deadlock.

In the initial state of the Petri net, p_1 and p_2 both contain a token, so the initial marking of the net is 1, 1, 0 and transition t_1 is enabled according to aforementioned rules, while transition t_2 is not enabled due to the fact that place p_3 in its input set is empty. This state is illustrated in Figure 2 where the enabled transition, that is, t_1, is highlighted.

If transition t_1 fires, it removes a token from place p_1 and a token from p_2 and adds a token to place p_3. After transition t_1's firing, the Petri net goes to a new marking as shown in Figure 3, which is 0, 0, 1. Now, transition t_2 is enabled and transition t_1 is no longer enabled. If transition t_2 fires, it removes a token from place p_3 and does not generate a token to any place. As shown in Figure 4, all of the Petri net's places are empty now and no transition in the net can fire any more. If no transition is enabled under a given marking of a Petri net, this net is in a deadlock state.

2.2. Coloured Petri Nets. To enhance the expressiveness of the pure Petri nets formalism, we have depicted above that many variants are designed including prioritized Petri nets (PPN) [16], timed Petri nets (TTN) [17], and probabilistic Petri nets (PPN) [18]. One of these extensions is coloured Petri nets (CPN) [14], whose main contribution is to extend the pure Petri nets' type system to a more elaborate one. In a pure Petri net, a token is simply a place holder which means that a place has a unit of data that can be absorbed by a transition, but the value of the token is ignored. A place is defined with a colour which is the type of the place and all tokens residents

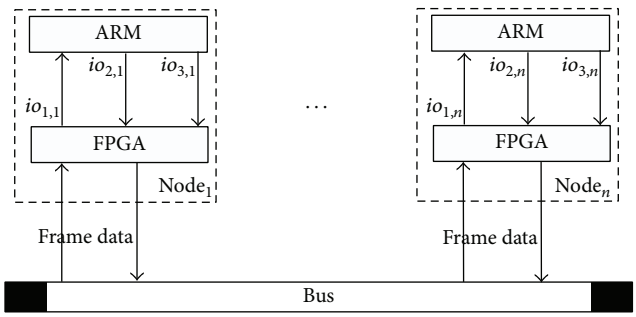

Figure 5: Representation of system.

in this place should be a value of the predefined type. A detailed explanation of its subtle execution semantics will be illustrated in the following demonstration and analysis of a set of CPN models.

3. Modeling of a Heterogeneous Embedded System

3.1. An Embedded System of Vehicle Protocol. Our system is an embedded system which implements the vehicle protocol *IEC-61375* [19]. The system consisted of a set of nodes; all connect to a bus which transfers frame data between these nodes, as shown in Figure 5. Each node consists of an ARM board and an FPGA board. An FPGA board may send frame data to the bus or receive frame data from the bus. It also communicates with an ARM board through GPIO ports in order to transfer messages between each other. An ARM board only interacts with its related FPGA board. There are three GPIO ports which are as follows.

(i) $io_{1,k}$ is used by the FPGA system to send a master frame interrupt to the ARM system in node k.

(ii) $io_{1,k}$ is used by the ARM system to notify the FPGA system of the arrival of a master message in node k.

(iii) $io_{1,k}$ is used by the ARM system to send a master message to the FPGA system in node k.

Among all nodes, there is a master node who monitors the whole network and all other left nodes are slave nodes under the master node's supervision. During a typical work flow, the master node polls slave nodes for new status data according to a predefined order which is stored in the ROM of the master node's ARM board part. This process is called a polling process and a success retrieval of a slave node's status by the master node is called a round of poll. Usually, a number is used to identify the node in order to decide the source and destination of each frame. The master node is numbered **1** under the typical configuration of such an embedded system and slave nodes are numbered from 2. In the system shown in Figure 5, Node$_1$ is the master node and Node$_2$ to Node$_n$ are slave nodes.

To illustrate how nodes communicate with each other, we discuss the polling process of the master node as an example in Figure 6. In the system introduced above, the ARM system is responsible for sending poll messages. As a result, all slave nodes have their FPGA part participated in the polling process and the master node has both the ARM part and the FPGA part involving in the polling process. In the following discussion, only three nodes are involved for simplification among which Node$_1$ is the master node while Node$_2$ and Node$_3$ are slave nodes. We let A_i $(i = 1, 2, 3)$ denote the ARM board of each node and F_i $(i = 1, 2, 3)$ the FPGA board. The bus is denoted by *Bus* : *Line*. The messages sent between the ARM system and the FPGA system are labeled as follows.

(i) **MF_INTR** is the master frame interrupt from the FPGA system to the ARM system.

(ii) **MF_Flag** is the notification signal from the ARM system to the FPGA system.

(iii) **MF_Data** is the poll message from the ARM system to the FPGA system.

In this demo, Node$_1$ is the master node, so the polling process starts from A_1 and F_1. F_1 first triggers a master frame interruption of A_1 through $io_{1,1}$ of the ARM board. A_1 then handles this interruption and starts polling the slave nodes in the network in a predefined order. A_1 sets $io_{2,1}$ to high level to notify F_1 the coming of a polling message, and then A_1 puts the content of a polling message with value **2** which means the message's target is Node$_2$) to $io_{2,1}$. F_1 is triggered by the high level of $io_{2,1}$ and encapsulates the message received from A_1 to a frame which has the format of (**type**: **poll**/**response**, **source**, and **target**). This frame is put on the bus in a broadcast way, meaning that all nodes (including the sender of the frame) connected to the bus know the existence of this frame. But only the frame's target will handle the content of it, and all other nodes will neglect the frame's content automatically. So, F_2 receives this frame from the bus and sends out its response frame back to the bus which targets Node$_1$. When F_1 gets the response frame from F_2, it triggers A_1 again to start another polling which targets Node$_3$.

3.2. Details of the System. The GPIO interface between the ARM board and the FPGA board works as an intermediate data store for the interaction of a sequential program written in C and a parallel program written in VHDL. We demonstrate an abstract version for the C program in Algorithm 1 and an abstract version for the VHDL program in Algorithm 2. The C code in Algorithm 1 is an interrupt handler which sends a master frame numbered from **2** to **SLAVE_COUNT** in a roll. The VHDL code in Algorithm 2 checks **mf_flag** every cycle and transfers the value from **mf_data** to **target_node**. The following CPN model is based on these codes.

Figure 7 shows a CPN model of the ARM system which receives a master frame interrupt and sends a poll message to the FPGA system. Figure 8 shows a CPN model of the FPGA system which interrupts the ARM system and transfers the polling information to the bus. Colour sets used in both models are defined in Table 1 which contains a list of places of the specified colour set. All related variables are listed in Table 2.

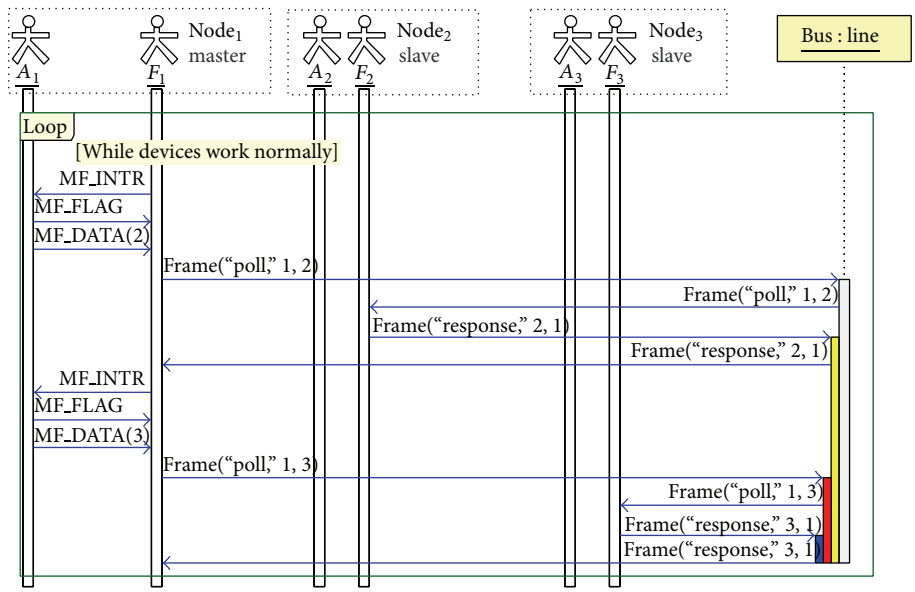

Figure 6: Message chart of polling process.

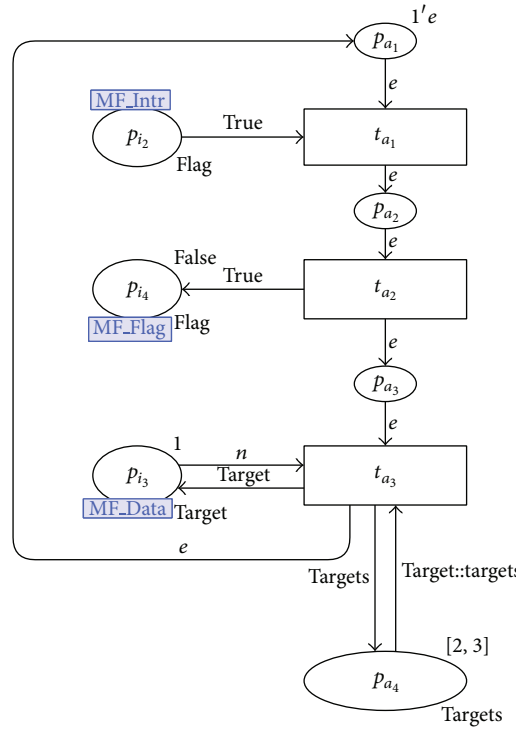

Figure 7: The ARM system's CPN model.

3.2.1. The ARM System. In the CPN model of the ARM system, the meanings of places are as follows.

(i) Places that represent states of the ARM system are p_{a_1}, p_{a_2}, and p_{a_3}, all of which have type *State*. Because tokens in these places can only have value **e**, we call these tokens *unit tokens*.

(ii) p_{i_2} and p_{i_4} are places with type *Flag*. p_{i_2} represents the master frame interrupt and p_{i_4} the signal for the coming master frame data.

(iii) Place p_{i_3} with type *Target* is used to store the master frame data sent from the ARM system to the FPGA system.

(iv) Place p_{a_4} with type *Targets* stores a list of numbers which represents the order of slave nodes to be polled.

In the initial state of the system, a unit token is located in p_{a_1}. t_{a_1} is enabled for the unit token in p_{a_1} and the token with value **true** in p_{i_4}. When t_{a_1} fires, it removes a unit token from p_{a_1} and a token from p_{i_4} and generates a unit token to

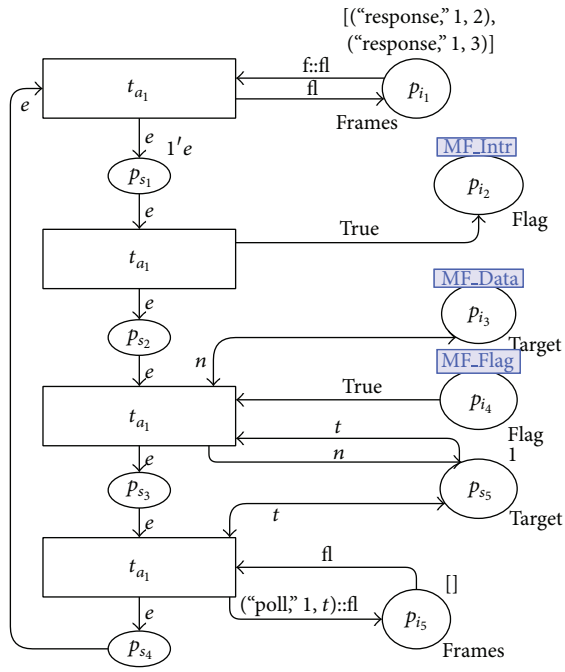

Figure 8: The ARM system's CPN model.

```
void master_frame_interrupt_handle ()
{
     node_number++;
     if (slave_node_number >= SLAVE_COUNT)
     {
          slave_node_number = 2;
     }
     arm_set_main_frame_flag (TRUE);
     arm_send_main_frame_data (node_number);
}
```

Algorithm 1: C code on the ARM system.

```
process (reset, clk, current_state,
          mf_flag, mf_data)
     begin
          if reset = '1' then
               current_state <= start;
          elsif clk'event and clk = '1' then
               if mf_flag = '1' then
                    target_node <= mf_data;
               current_state <= next_state;
               end if;
          end if;
end process;
```

Algorithm 2: VHDL code on the FPGA system.

place p_{a_2}. Then, t_{a_2} is enabled immediately when p_{a_2} receives a unit token. t_{a_2}'s firing removes a unit token from place p_{a_2} and then puts a unit token to p_{a_3} and a token with value **true** to p_{i_4}. The token with value **true** in p_{i_4} indicates the FPGA system of a master frame poll message from the ARM system. Then, t_{a_3} is enabled due to the unit token in p_{a_3} and the token in p_{i_3}. t_{a_3} has three input places: p_{a_3}, p_{a_4}, and p_{i_3}. The firing of t_{a_3} consumes a unit token from p_{a_3}, a token with type *Target* from p_{i_3}, and a token with type *Targets* from p_{a_4}. The *Targets* token from p_{a_4} is bound to two variables: **target** which corresponds to the number of the slave node being polled currently and **targets** which corresponded to numbers of remaining slave nodes waiting to be polled. The firing of transition t_{a_3} sends a unit token to p_{a_1} indicating the change of system state, a *Target* token called **target** to p_{i_3} indicating the slave node's number to be polled and a *Targets* token to p_{a_4} with numbers of remaining slave nodes to be polled in the coming polls. When t_{a_3} finishes its firing, the ARM system waits the master frame interrupt again to start a new poll.

3.2.2. The FPGA System. In the CPN model of the FPGA system, the meanings of places are as follows.

(i) Places that represent states of the FPGA system are p_{s_1}, p_{s_2}, p_{s_3}, and p_{s_4} all of which have type *State*. Tokens in these places are *unit tokens* just like those in the ARM system's state places.

(ii) p_{i_2}, p_{i_3}, and p_{i_4} are the same places in Figure 7.

(iii) p_{i_1} and p_{i_5} simulate the frames sent and received between the bus and the FPGA board. p_{i_1} represents frames from the bus and p_{i_5} represents frames to the bus.

Table 1: Colour set definitions.

Colour set	Definition	Places
State	*Unit* **withe**	p_{a_1}, p_{a_2}, p_{a_3}, p_{s_1}, p_{s_2}, p_{s_3}, p_{s_4}
Flag	*BOOL*	p_{i_2}, p_{i_4}
Frame	**product** *STRING* × *INT* × *INT*	
Target	*INT*	p_{s_5}, p_{i_3}
Frames	**list** *Frame*	p_{i_1}, p_{i_5}
Targets	**list** *Target*	p_{a_4}

Table 2: Variables in model.

Variables	Colour set
n, **t**	*INT*
flag	*BOOL*
f	*Frame*
fl	*Frames*
target	*INT*
targets	*Targets*

(iv) Place p_{s_5} with type *Target* stores the number of slave node that will be used to construct a poll frame.

In the initial configuration, p_{s_1} contains a unit token, so t_{s_1} outputs a token with value **true** to p_{i_2} and a unit token to p_{s_2}. The system of ARM treats a token with value **true** in p_{i_2} as a trigger to start its master frame process. t_{s_2} has four input places: p_{s_2}, p_{i_3}, p_{i_4}, and p_{s_5}. p_{i_3} and p_{s_5} already contain a token in each of them, so when the ARM system gives p_{i_4} a token with value **true**, t_{s_2} is enabled. p_{i_3} offers t_{s_2} a *Target* token which is bound to variable **n** indicating the number of slave node to be polled next. p_{s_5} is updated in t_{s_2}'s firing and a *Target* token with value **n** replaces the old value in this place. p_{s_3} gets a unit token, meaning the change of the FPGA system's state. t_{s_3} retrieves a *Target* token **t** from p_{s_5}, composes a poll frame in the form of (**poll**, **1**, **t**), then appends this frame to the *Frames* token from p_{i_5}, and sends an updated token back to this place. A unit token is sent to p_{s_4} by t_{s_3} and the FPGA system goes to the monitoring state. When p_{i_3} has tokens, t_{s_4} fires to remove the head of the **Frames** token in p_{i_1} and sends a unit token to p_{s_1}. The FPGA system goes back to its initial state and starts another master frame poll process of the ARM system.

4. Analysis of the Model

4.1. Review of the Model. The FPGA system cyclically checks the state of master frame flag on the port between the ARM system and itself and once the value of the flag becomes **true** in a cycle, the FPGA system tries to retrieve new master frame data from the corresponding port immediately. As shown in Algorithm 2, the FPGA system gets master frame data and sends the data to a signal called **target_node** that represents the current number of the slave node to be polled. Then, the value of **target_node** is used to construct a poll frame sent to the bus.

According to the designer's intention, the value in **target_node** should be updated every time the FPGA system retrieves master frame data. But t_{s_2} in Figure 8 updates the token in t_{s_5} according to the token in t_{i_2}. However, the process of updating may be ineffective and, as a result, the token in t_{s_5} contains a stale value. To explore this scenario in detail, we focus on the interaction between the ARM system and the FPGA system. There are five transitions which participate in the interaction:

(i) t_{s_1} and t_{s_2} of the FPGA system,

(ii) t_{a_1}, t_{a_2}, and t_{a_3} of the ARM system.

t_{s_1}, t_{a_1}, and t_{a_2} always execute in an order according to the strict dependency caused by tokens' generation and consumption in p_{i_2} and p_{i_4}. So, the execution trace of the three transitions is decidable. The other two transitions which are involved in the interaction act in a different manner. After t_{a_2} fires, a token with value **true** appears in p_{i_2} and a unit token appears in p_{a_3}, which activates t_{s_2} and t_{a_3}, respectively. Normally, t_{s_2} should fire after t_{a_3}, so that the FPGA system can construct a right frame which is sent to a slave node. However, the nature of C programs running on an ARM platform is asynchronous sequential execution, while the nature of VHDL programs running on an FPGA platform is synchronous concurrent execution. Usually, the FPGA system will complete the check of the master frame flag and retrieval of the master frame data in the same cycle. So, possibility exists that the FPGA system retrieves the value of the master frame data before its value is updated by the ARM system and thus gets a token with a stale value. We cannot predicate when will the ARM system updates the value of the master frame data after the value of the master frame flag is set to **true** for the undecidability of the operating system's scheduling scheme. So, if the FPGA system's new execution cycle starts before the value of master frame data is set by the ARM system, a timing mismatch happens. To verify the analysis of the data inconsistency in the CPN model, we apply the approach of model checking on it in the following section.

4.2. Property Analysis. To analyze the behavior of a modeled system, a state-space based approach can be used to dive into the details of models and get a thoroughly understanding of the system. A state of a CPN model is the marking of the model, that is, the number and value of tokens in each place in the whole model, and the initial marking of a model is its initial state. A CPN model changes its state when a transition of the model fires. All the states that a CPN model possibly accesses through a series of transition firing are called the state space of the model which may be finite or infinite. A property formula is an assertion about the state space of a model that whether the property holds in all states of the model and is usually called *property* for short. Model checking is an approach that checks whether a property about a model is fulfilled. If the property is not fulfilled, the model checking algorithm gives a counter example which violates the property. There are many description formalisms for properties, such as *CTL*, *LTL*, and *CTL** (add some refers) and many model checking algorithms target these

Table 3: Property related functions.

Function	Type
isStale	*Transition* → *BOOL*
myASKCTLformula	*Node* → *BOOL*
eval_node	(*Node* → *BOOL*) → *BOOL*

formalisms. In the field of CPN, *ASK-CTL* is used to describe the property of a CPN model. CPN tools implement the representation of *ASK-CTL* formulas and its model checking algorithms in SML language. A state is also called a node because the state space is represented as a state graph in which an edge represents a firing of a transition and a node represents a marking of the model. In the following statements, we will refer to states as nodes when needed.

Let A be the abstract *ASK-CTL* formula type; then the following formulas with type $A \rightarrow A$ are considered.

(i) *POS* is a node formula which is true if it is possible, from the current node, to reach a node where A is true.

(ii) *INV* is a node formula which is true if A is true for all reachable nodes from the current node.

(iii) *EV* is a node formula which is true if A eventually becomes true from the current node.

(iv) *ALONG* is a node formula which is true if there is a path for which A is true for every node.

The CPN model of the master node system has the potential to enter a node in which there is a data mismatch caused by interface incompatibility. This kind of defect can be expressed as:

"when transition t_{a_2} fires, t and n which bind to the transition are equal in value."

So, if this property is true in a node, that is, the value of master frame data read by the FPGA system is equal to the value of **target_node** got in previous cycles, a data mismatch appears. This property can be coded as *ASK-CTL* formula in SML form.

The types of these variables/functions are listed in Table 3.

isStale is a transition function which returns **true** when the parameter is a transition equal to t_{a_2} bound with $n = t$ where $n = 2, 3$ in this case. myASKCTL-formula is a node function which takes a node as the only parameter and returns **true** when there is a node in which isStale is evaluated to be **true**. eval_node is a function which takes a node function as the property formula and a node as the starting node and returns **true** when the application of the formula on the node returns **true**. In this case, we evaluate the property written in SML code above and get the following result: **var it** = **true**: *BOOL*, which means the property holds with respect to the initial state in state space (see Algorithm 3); that is, there is a node in which p_{s_5} gets a token with a stale value.

```
fun isStale a =
    (Bind.FPGA'Receive_MF_Data
        (1, {n = 3, t = 3})
    = ArcToBE a) orelse
    (Bind.FPGA'Receive_MF_Data
        (1, {n = 2, t = 2})
    = ArcToBE a);
val myASKCTLformula =
    POS(MODAL(AF("stale value", isStale)));
eval_node myASKCTLformula InitNode;
```

Algorithm 3: Propery formula in SML code.

5. Conclusion

We will not blame a C programmer or a VHDL programmer for writing the code shown in Algorithms 1 and 2, since they both obey the common paradigm in their respective domain. However, in development of a heterogeneous system, a programmer should be more alert and pay special attention to the situation when a normal assumption is not fulfilled by the guarantee of other domains in which a defect in the resulted system may occur. Many approaches can be used to reduce this potential risk and one of them is model checking method. In this paper, we demonstrate a heterogeneous system which is composed of two subsystems: an ARM system and an FPGA system. One of the subsystems behaves synchronously and the other asynchronously. Through a state based analysis way, we find defect in the design of the system which may lead to data inconsistency between the two subsystems. Then, our analysis is verified by constructing a property formula which is computed by a model checking program.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," *ACM Sigplan Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[3] E. A. Lee, "Cyber physical systems: design challenges," in *Proceedings of the 11th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '08)*, pp. 363–369, May 2008.

[4] S. Vercauteren, B. Lin, and H. de Man, "Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications," in *Proceedings of the 1996 33rd Annual Design Automation Conference*, pp. 521–526, IEEE, June 1996.

[5] S. Kumar, *The Codesign of Embedded Systems: A Unified Hardware/Software Representation*, Springer, 1996.

[6] B. Schatz, A. Pretschner, F. Huber, and J. Philipps, "Model-based development of embedded systems," in *Advances in Object-Oriented Information Systems*, pp. 298–311, Springer, 2002.

[7] M. Törngren, D. Chen, and I. Crnkovic, "Component-based vs. model-based development: a comparison in the context of vehicular embedded systems," in *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA '05)*, pp. 432–440, IEEE, September 2005.

[8] D. C. Schmidt, "Guest editor's introduction: model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[9] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–389, 1997.

[10] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*, vol. 11, Kluwer Academic Publishers, 1995.

[11] A. Fin, F. Fummi, M. Martignano, and M. Signoretto, "SystemC: a homogenous environment to test embedded systems," in *Proceedings of the 9th International Symposium on Hardware/Software Codesign*, pp. 17–22, ACM, April 2001.

[12] Y. Jiang, H. Zhang, X. Song et al., "Bayesiannetwork-based reliability analysis of plc systems," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 11, pp. 5325–5336, 2013.

[13] "BPDF: a statically analyzable dataflow model with integer and boolean parameters," in *Proceedings of the 11th ACM International Conference on Embedded Software (EMSOFT '13)*, V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, Eds., pp. 1–10, September 2013.

[14] K. Jensen, "Coloured petri nets," in *Petri Nets: Central Models and Their Properties*, pp. 248–299, Springer, 1987.

[15] H.-M. Hanisch, J. Thieme, A. Lueder, and O. Wienhold, "Modeling of PLC behaviour by means of timed net condition/event systems," in *Proceedings of the IEEE 6th International Conference on Emerging Technologies and Factory Automation (ETFA '97)*, pp. 361–369, September 1997.

[16] G. Balbo, "Introduction to stochastic petri nets," in *Lectures on Formal Methods and Performance Analysis*, pp. 84–155, Springer, 2001.

[17] L. Popova-Zeugmann, "Timed petri nets," in *Time and Petri Nets*, pp. 139–172, Springer, 2013.

[18] M. A. Marsan, "Stochastic petri nets: an elementary introduction," in *Advances in Petri Nets 1989*, pp. 1–29, Springer, 1990.

[19] C. Schaefers and G. Hans, "Iec 61375-1 and uic 556-international standards for train communication," in *Proceedings of the IEEE 51st Vehicular Technology Conference (VTC '00)*, vol. 2, pp. 1581–1585, IEEE, Tokyo, Japan, May 2000.