Modeling and Validation of a Data Process Unit Control for Space Applications

Wan Hai, Huang Chongdi,
Wang Yuhui, He Fei and Gu Ming
Key Lab of ISS of MOE, TNList, School of Software,
Tsinghua University, Beijing, China

Chen Rui
Beijing Institute
of Control Engineering,
Beijing, China

Marius Bozga
UJF-Grenoble 1 / CNRS
VERIMAG UMR 5104
Grenoble, F-38041, France

***Abstract*—Data process unit (DPU) is a typical embedded system. It is widely used in space applications to collect data from sensors, process data and send the data to its upper master computer. In this paper, we use the BIP framework to model and validate a DPU system of a real space application. We first build the system model including the control software, hardware and the environment. Validation is by extensive simulation of a monitored system obtained as the composition of the DPU model with monitors. A monitor checks a requirement by continuously sensing the state of the model and reaching an error state if the requirement is violated. We checked fault-tolerance for different fault models and detected several errors that under some conditions, could correspond to real implementation errors.**

I. Introduction

Data process unit (DPU) is a typical embedded system. It is widely used in space applications. DPUs are mainly placed in the middle layer of a space system and are used to obtain data from their lower sensors, process data and send data to their upper master computer. For DPU systems the main concerns are 1) whether a operation is finished in the predefined time; 2) whether an operation is executed according to the predefined protocol; 3) whether the data returned by the DPU conforms to the specification. In this paper, we use a DPU system obtained from a real space application as a case study.

The system structure of the case is shown in Fig. 1. There are four sensors, a master computer, a multiplexer and a DPU. Sensor1 is connected to the DPU directly, while sensor2, sensor3, sensor4 and the master computer are connected to the DPU through the multiplexer, which is controlled by the DPU. There are two interrupts used in the DPU software: the synchronous interrupt, INT0, and the serial port interrupt, INTCOM, which has higher priority than INT0. Any data arriving at the serial port (the UART box in Fig. 1) leads to an INTCOM interrupt and every rising edge of the synchronous signal leads to an INT0 interrupt. There are three routines in the DPU software, respectively: the main routine, which is used to do the initialization work, the INTCOM handling routine and the INT0 handling routine.

The informal requirements of the software can be described using Fig. 2. The cycle length of the synchronous signal is 150ms. For every rising edge of the signal, there is an INT0 interrupt, which in turn triggers an INT0 handling routine. In the INT0 handling routine, the data of sensor1 should be latched and read. 4.5ms after the rising edge, the master computer will send a data-request command to the DPU (through the multiplexer), which triggers an INTCOM interrupt. In the INTCOM handling routine, the program executes the following sequence in 13ms: 1) latch and read the data of sensor1; 2) control the multiplexer to select sensor2 (sensor3, sensor4, resp.), send the data-request command and receive the data sent back by sensor3 (sensor3, sensor4, resp.); 3) process and pack the data obtained from sensor1 to sensor4; 4) control the multiplexer to select the master computer and send the packed data.

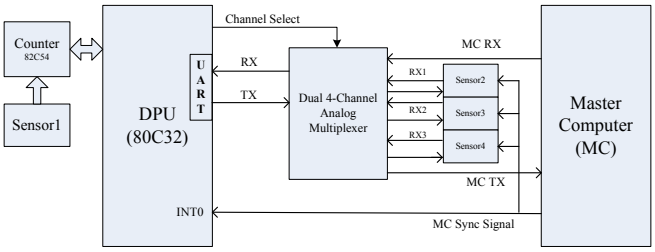

Fig. 1. The system structure

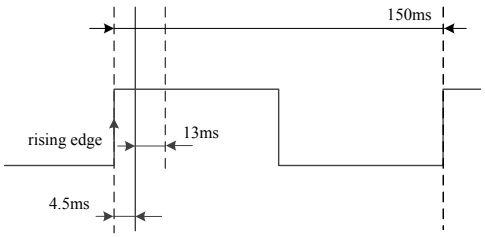

Fig. 2. The timing diagram

Besides the requirements mentioned in the previous paragraph, we have three functional requirements: (a) the master computer should be selected by the DPU, when it sends the data-request command to the DPU; (b) sensor1 provides the latch and read functions, these functions should be called according to a specific protocol; (c) the DPU should send the "right" data to the master computer[1]. There are also two

[1]For the formal description of "right" data, please refer to the technical report [1].

additional requirements expressing real-time constraints: (d) the INT0 handling routine should be finished in 4.5ms and (e) the INTCOM handling routine should be finished in 13ms.

In order the validate the DPU system, we use the BIP framework [2]. The main work flow is shown in Fig. 3. First, the system, including hardware, software and environment, is modeled as a BIP component. Then all the requirements are modeled by monitor components. Finally, after composing the DPU system model and the monitors, we use BIP tools for validation.

In the flow, it is essential to come up with an adequate and faithful model. This involves the following tasks: decide which parts of the system should be modeled, decompose the system, choose the abstraction level, model application-specific features in BIP, such as the use of shared variable.

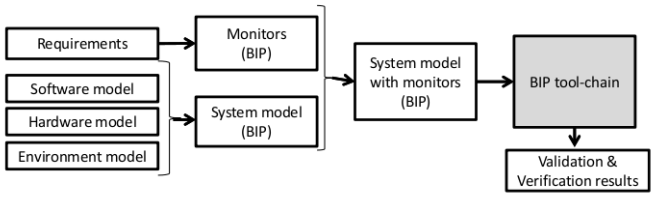

Fig. 3. The work flow

The rest of the paper is structured as follows. Section II gives a short introduction of the BIP framework. In section III we provide an overview of the BIP model of the DPU system. Section IV presents the validation phase and section V concludes the paper.

II. The BIP Framework

BIP (Behavior, Interaction, Priority) is a component framework intended to rigorous system design [3]. It allows the construction of composite hierarchically structured components from atomic components characterized by their behavior and their interface. Components are composed by layered application of interactions and of priorities.

A. BIP Concepts

Atomic components are finite-state automata extended with variables and ports. Variables are used to store local data. Ports are action names, and may be associated with variables. They are used for interaction with other components. 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 has associated a guard and an action, that are respectively, a Boolean condition and a computation defined on local variables. In BIP, data and their transformations are written in C/C++. For example, Fig. 8 provides a graphical representation of an atomic component used for the modeling of the DPU case study.

Interactions express synchronization constraints between ports (actions) of the composed components. Interactions are described in BIP as the combination of two types of elementary protocols: *rendez-vous* to express strong symmetric synchronization and *broadcast* to express triggered asymmetric synchronization. Interactions are defined using 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. Rendez-vous 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. For example, Fig. 6 illustrates several connectors used for the composition of the top-level DPU component. 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, respectively a low-priority and a high-priority one. Whenever the condition (on the state of the system) holds, if the two interactions are enabled 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 choices made in the definition of BIP are paramount. More information about the underlying concepts and the operational semantics of BIP can be found in [2], [4]. In particular, BIP provides separation of concerns between behavioral and architectural aspects in modeling. For BIP components, architecture is meaningfully defined as the combination of interactions and priorities. Component architecture can therefore be easily decoupled, understood and analyzed independently of the associated behavior. Moreover, [5] presents a study about expressivity of BIP and related component-based frameworks. It is shown that the combination of interactions and priorities confers BIP a universal form of expressiveness, actually not matched by any other existing formalism. Furthermore, besides theoretical studies, expressivity of BIP has been confirmed by the very numerous translations defined from existing models of computation and domain-specific languages into BIP. A survey and pointers to relevant publications are available on the BIP web site [6].

B. BIP Tools

The BIP toolbox [6] includes a rich set of tools for modeling, code generation, execution, analysis (both static and on-the-fly), transformations of models. An overview of the toolbox is given in Fig. 4.

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 variables and data type declarations, expressions and statements, and provides additional structural syntactic constructs for defining component behavior, specifying the coordination through connectors, and describing the priorities. Moreover, it provides additional constructs for dealing with parametric descriptions (i.e., where the same component occur replicated in many places) as well as for expressing timing constraints associated with behavior.

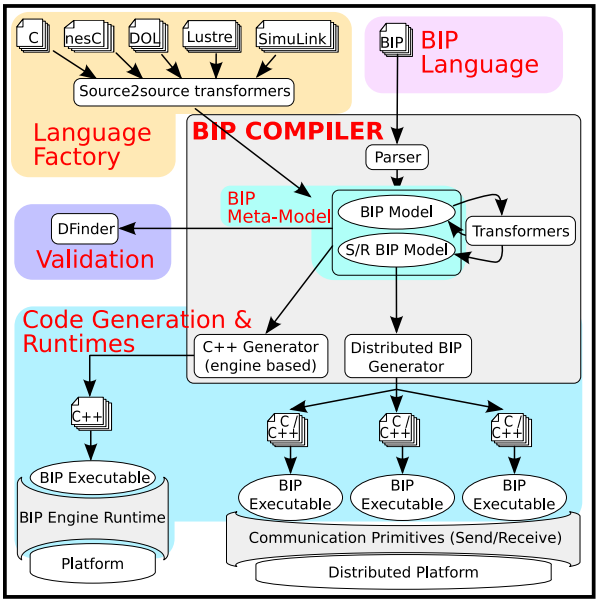

Fig. 4. An overview of the BIP toolbox

The toolbox 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 [7] and distributed execution [8]. Back-end tools include specific runtimes for analysis (through simulation) and efficient execution on machines with different characteristics and OS support (e.g., real-time, mono/multi-thread, single/multi-core).

Validation of BIP models can be achieved by using static and/or runtime validation techniques. Static validation techniques are provided by the D-Finder tool [9]. D-Finder implements state-of-the-art compositional methods [10] for computing invariants for systems consisting of interacting components. Invariants are safe (over) approximations of the set of reachable states of the system and can be used to prove safety requirements. In the case of BIP components, invariants computed are conjunctions of local invariants for atomic components and interaction invariants characterizing the interactions glue. Local component invariants are generated by static (and individual) analysis of atomic components. Interaction invariants are generated from abstractions of the interacting components and the interactions glue.

Runtime validation techniques available for BIP are based on construction and execution of monitored systems. Historically, this validation approach is oriented towards finding errors rather than proving their absence from designs[2]. This approach has been adapted for BIP components as explained in [11]. It consists in constructing an executable model of the designed system together with monitors responsible for evaluation of safety requirements. Monitors are atomic components that sense the system state (through interaction with system's components) and react by moving to error states whenever the safety requirement is violated i.e., if an inconsistent state is reached or an invalid sequence of interactions has been executed, etc. The BIP framework provides native support for building and running executable models for monitored systems.

III. Overview of the BIP Model

The hierarchy of the system with monitors is shown in Fig. 5. The overall structure of the BIP model for the system is shown in Fig. 6. The DPU control software is furthermore decomposed into six subcomponents as shown in Fig. 7. We adopt the following decomposition principles:

- *Distinguish two levels separating hardware from software*. For hardware, we have 6 components: one component for each sensor, one for the multiplexer and one for the master computer. At the DPU software level, decomposition follows on the software structure: besides the components for each interrupt routine, we have the serial port component, the scheduler component, which is used to accept the interrupts and schedule the routines, and the shared-variable component.
- *Adopt an adequate abstraction level*. For example, sensor1 is a single microcomputer. It can be decomposed to subcomponents as we do for the DPU. Nevertheless, our modeling focuses only on two services it provides, i.e., lock and read. Therefore, an atomic component is sufficient.
- *The decomposition follows the functionality*. For example, the serial port component (shown in Fig. 8) consists of two subcomponents because it provides two functions: send data and receive data, each of which is modeled by a subcomponent.

Generally speaking, the ports of a component represent the services or events the component provides or uses. For example, sensor1 provides two services. Hence, it has two corresponding ports. An important issue in identifying ports is to decide the type of synchronization for ports: strong synchronization (drawn as a bullet in figures) or broadcast (drawn as a triangle in figures). Notice that ports modeling a service used or provided as well as tick ports are involved in strong synchronizations. On the contrary, ports used for sending messages which may not be received or emitting an interrupt signal, initiate broadcasts.

The behavior of a component is modeled by a finite state automaton extended with data and functions written in C/C++.

[2]actually, stronger guarantees on correctness can be obtained by combining simulation with statistical techniques as in statistical model-checking

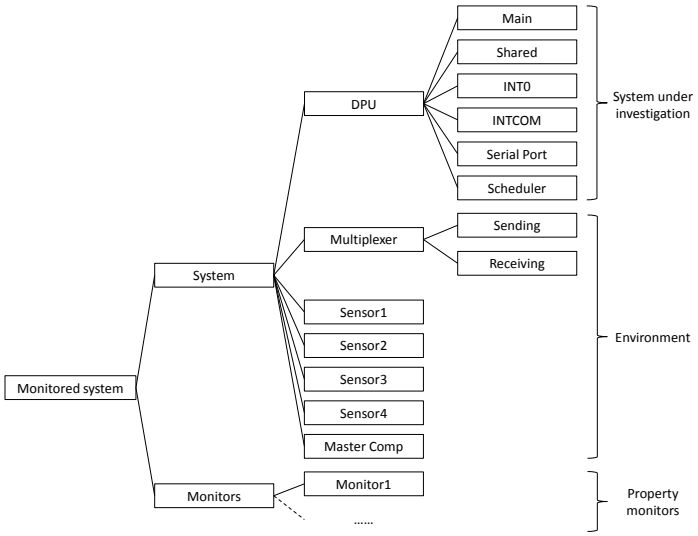

Fig. 5. The hierarchy of the BIP model with monitors

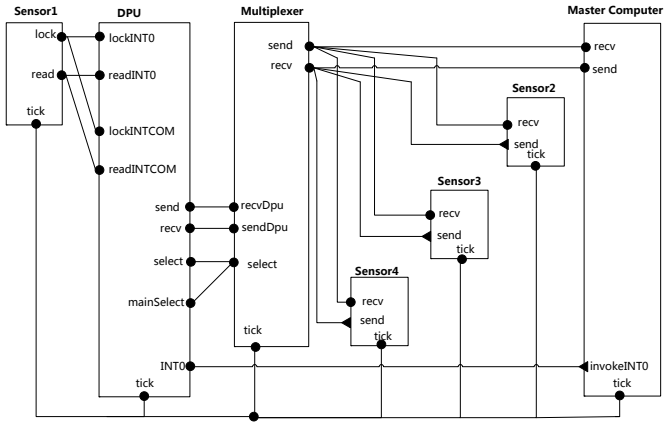

Fig. 6. The overall structure of the BIP model

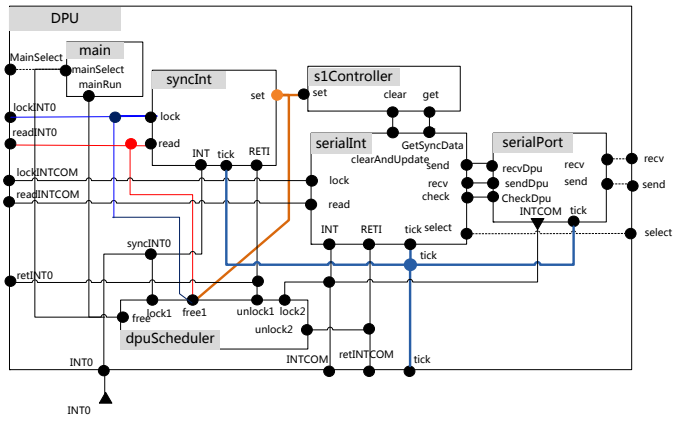

Fig. 7. The BIP model of DPU

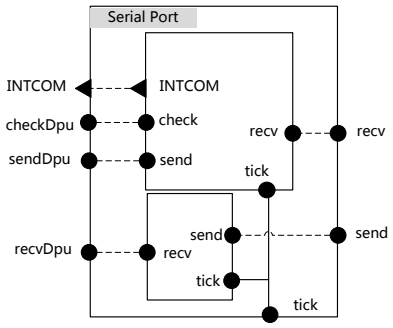

Fig. 8. The BIP model of the serial port

For example, the receiving part of the serial port is shown in Fig. 9. Port recv is used to receive data from the multiplexer, port check is used for the routines to check whether there is a new data arrived, port send for the routines to read the data of the serial port and port INTCOM for emitting the interrupt signal. The initial state is IDLE. When it receives a data from the multiplexer, it will issue an INTCOM interrupt signal. In the automaton, we know that 1) if a newly arrived data is not read then the serial port cannot receive a data; 2) if a routine does not check whether there is a new data before reading the serial port, it may read an old data.

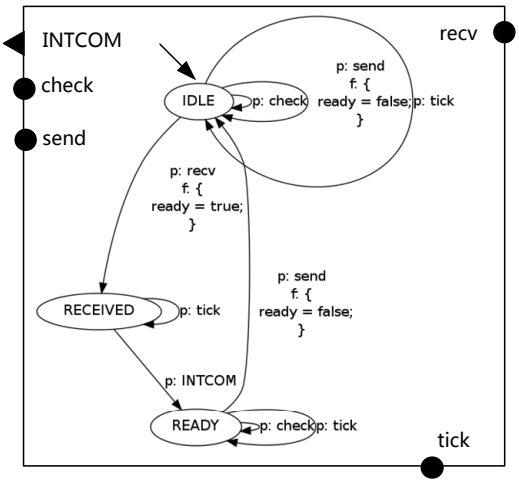

Fig. 9. The receiving part of serial port

There are two main lessons we learnt from the constructions of models:

- behavior should not necessarily capture all the implementation details. Take the serial port for example. At first, we want to build a model based on the hardware structure of serial port. Later, we find that it is too complex to follow and it is not necessary for our case since the modeling of serial port is not the key concern. Finally, by reading the specification of serial port and guiding by the actual needs, we build a simplified and correct model for the serial port.
- modeling choices must not add irrelevant constraints

into the design. For example, components should not be unexpectedly blocked, especially for those modeling environment. Take the component of the master computer for example. This component should emit the synchronous signals every 150ms. The first version of the component may be blocked if the DPU does not return the data in time. Although this deadlock can be found through simulation or verification, in real implementations this situation cannot happen.

For the case study, we construct 13 atomic components and 3 composite components. The BIP model is about 1100 lines of BIP code. The total development time is about three weeks including learning the use of BIP language and associated tools.

IV. Overview of the Validation

Validation involves three main steps, respectively, 1) building a set of environment models, 2) building monitors to express functional requirements, 3) composing the system model and the monitors, and run the simulation BIP tool.

A. Environment modeling

The system may always behave correctly for perfect scenarios. In order to evaluate correctness we need to design a set of potentially faulty environment models. In our case, we design four models for the master computer:

1) PMC: a perfect master computer which has perfect timing and does not miss signals (as shown in Fig. 2);
2) NPMC1: a non-perfect master computer that has perfect timing but may miss arbitrary signals[3];
3) NPMC2: a non-perfect master computer that has perfect timing and may miss signals, but cannot miss both signals in one cycle (as shown in Fig. 10);
4) NPMC3: a non-perfect master computer that has perfect timing and may miss signals, but cannot miss two adjacent signals;

B. Requirements modeling

As mentioned in section I, there are five functional requirements we want to validate. Each requirement is modeled by a monitor component. Typically, a monitor contains one or more states. Reaching error states means that the requirement is violated. In this case study, there are two monitor design principles: 1) The monitor should not influence the execution of the system; 2) The monitor should monitor every event it needs – if needed, we should add specially designed ports to the system under investigation.

Not influencing the system involves basically two conditions: the monitors should not change the inner state of the system neither block (i.e., restrict) the execution of the system by synchronization. For the first condition, the monitors should only read the data of the system – writing is forbidden. There are two ways to ensure the second condition. We take

[3]There are two signals that may be lost: the rising edge of synchronous signal and the data-request command of the master computer.

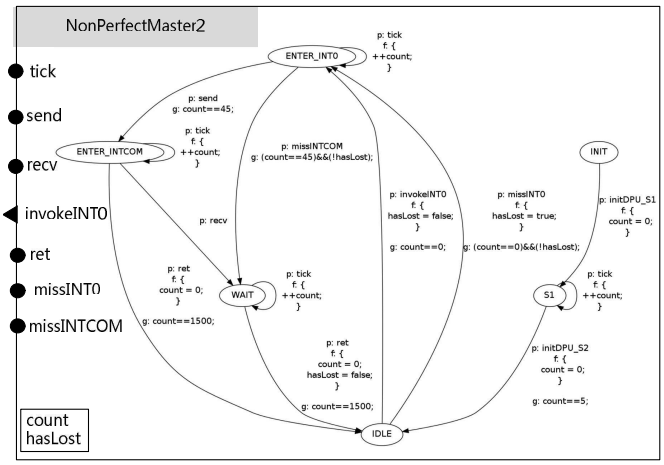

Fig. 10. The non-perfect master computer NPMC2

for example the monitor for checking if the execution time of INT0 handling routine exceeds the predefined time (as shown in Fig. 11). All events the monitor observes should be connected by broadcasts, such as the connectors between INT and all.invoke0 and between RETI and all.ret0. The ports on the system model side should be triggers. However, if the port is used for strong synchronization (such as the tick port used to synchronize time between the monitor and the system) then the port should be enabled on all monitor states except for the error states. In other word, every state except for the error states has a self-cycle labelled by the port.

In order to illustrate the second principle, we take NPMC2 for example. The BIP model of NPMC2 is shown in Fig. 10. NPMC2 would miss signals that are modeled by two transitions labelled missINT0 and missINTCOM. At first, these two ports are inner ports that cannot be observed by other components. Nonetheless, during the construction of right data monitor, we found that the monitor should perceive the events of missing signals. Hence, we export these two ports.

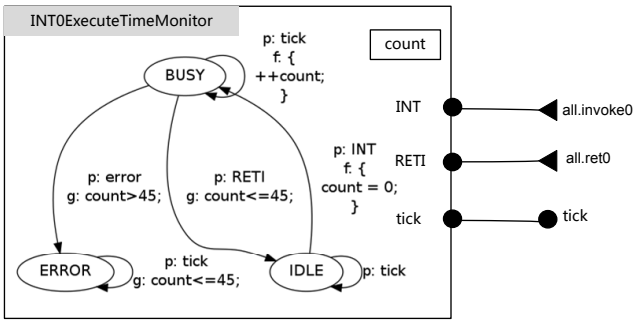

Fig. 11. The INT0 handling routine execution time monitor

C. Validation results

The system model together with monitors form a BIP model as well. The generation of a executable simulation program includes two steps. First, the model is compiled into C++

code using the BIP compiler tools. Second, the C++ code is linked with the BIP engine and compiled into an executable program. Simulation of the model is done by executing the above program.

Reaching an error state would usually cause a global deadlock. For example in Fig. 11, if the monitor reaches state ERROR, since there is no tick that can be triggered, the system time cannot advance, which results in a global deadlock. This would reduce the effectiveness, because only one bug can be found during one simulation run. In order to improve this, we modified monitors in order to allow the system to resume its execution whenever a violation is found.

We perform validation of the five functional requirements, for each fault model. The results are summarized in Tab.I.

TABLE I
VALIDATION RESULT

	PMC	NPMC1	NPMC2	NPMC3
Multiplexer Monitor (a)	√	√	√	√
Latch instruction Monitor (b)	√	√	√	√
Right data Monitor (c)	√	×	×	×
INT0 routine time Monitor (d)	√	√	√	√
INTCOM routine time Monitor (e)	√	√	√	√
Simulation time	10m	10m	10m	10m
Number of simulated cycles	6627	6627	6900	7746

After analyzing the traces generated by the simulation, a counterexample for requirement (c) using faulty environment model NPMC1 is shown in Fig. 12. In the trace, the data-request command of cycle 1 and the INT0 interrupt of cycle 2 are lost. S_{11} to S_{16} are the values of sensor1 at the corresponding time positions. The expected return value is $S_{16} - S_{13}$ but the DPU returns $S_{13} - S_{11}$.

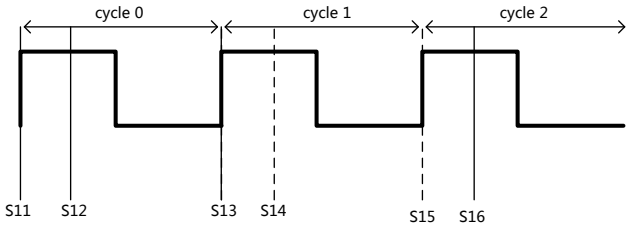

Fig. 12. A counterexample

V. CONCLUSION

Data process unit (DPU) is a typical embedded system. It is used in space applications to collect data from sensors, process data and send the data to its upper master computer. In this paper, we use the BIP framework to model and validate a DPU system obtained from a real space application. We first build the system model including the control software, hardware and the environment. In order to do validation, we devise several fault models and for each requirement, we build a monitor component. After composing the system model and monitors together, we use BIP tool chain to do the validation based on simulation. We found several bugs in the design, at different stages of the development.

This experiment perfectly illustrates the expressive power and the modeling and validation facilities provided by the BIP framework. The models developed have been shared with industrial designers of DPU applications. The feedback obtained is particularly positive: the BIP models are clear, understandable and useful. BIP models definitely help to clarify design issues and to identify and correct rapidly potential design problems.

The work on this case study is currently being extended in two directions. First, we are investigating the applicability of more powerful verification techniques and tools, such as compositional generation of invariants [10]. Contrary to simulation, these techniques are computationally much expensive but they can provide stronger guarantees about the correctness of safety properties. The second direction concerns implementation. We are planning to adapt the BIP code generator and to develop a domain-specific extension allowing to produce the low-level implementation of the DPU software on specific processors.

ACKNOWLEDGEMENT

This work was supported by the 973 Program of China (No. 2010CB328003), the National Natural Science Foundation of China(No. 91018015, No. 60811130468) and the EU FP7 Project COMBEST (IST STREP 215543).

REFERENCES

[1] H. Wan, C. Huang, Y. Wang, and F. He, "Dpu case study using the bip framework," Institute of Software Theory and System, School of Software, Tsinghua University, Tech. Rep. TR-2011-BIP-DPU-1, May 2011.

[2] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Software Engineering and Formal Methods, 2006, SEFM 2006, 4th IEEE International Conference*. IEEE, 2006, pp. 3–12.

[3] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[4] S. Bliudze and J. Sifakis, "Causal semantics for the algebra of connectors," *Formal Methods in System Design*, vol. 36, no. 2, pp. 167–194, 2010.

[5] ——, "A Notion of Glue Expressiveness for Component-Based Systems," in *CONCUR'08*, ser. LNCS, vol. 5201. Springer, 2008, pp. 508–522.

[6] "The BIP Toolset," http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html.

[7] M. Bozga, M. Jaber, and J. Sifakis, "Source-to-source architecture transformation for performance optimization in bip," *IEEE Trans. Industrial Informatics*, vol. 6, no. 4, pp. 708–718, 2010.

[8] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," in *Proceedings of the 10th International conference on Embedded software, EMSOFT 2010*, 2010, pp. 209–218.

[9] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *Computer Aided Verification, 21st International Conference, CAV 2009, Proceedings*, ser. LNCS, vol. 5643, 2009, pp. 614–619.

[10] ——, "Compositional verification for component-based systems and application," *IET Software*, vol. 4, no. 3, pp. 181–193, 2010.

[11] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem, "Runtime verification of component-based systems," in *Software Engineering and Formal Methods, 2011, SEFM 2011, 9th International Conference*, 2011, pp. 204–220.