

Modeling a Bus Protocol: An Incremental Approach

Ricardo Bedin França^{1, 2}, Jean-Marie Farines¹, Jean-Paul Bodeveix², Leandro Buss Becker¹,
Mamoun Filali Amine²

¹Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88040-200 – Florianópolis – SC – Brazil

²Institut de Recherche en Informatique de Toulouse – Université Paul Sabatier
Toulouse, France.

{rbedin, farines, lbecker}@das.ufsc.br, {bodeveix, filali}@irit.fr,

***Abstract.** In a real-time system architecture, the notion of a bus component plays an important role as it forms the backbone of communication among all the devices of the system. For this purpose, we need a precise specification of buses for applications that will run on top of them and for developers who implement device protocols. In this paper, we propose an incremental methodology to elaborate detailed bus protocol descriptions that may be useful in both design and temporal property verification, by specifying a protocol from a simple representation to a complete one by means of successive refinements, thus also permitting the refinement of model properties. The methodology is then tested with AADL and TLA specifications of the PCI bus protocol.*

1. Introduction

1.1. Context

In the last years, systems designers have witnessed unprecedented advances in available technologies and possibilities to devise embedded systems. As a result, systems become more complex, thus needing more powerful design methodologies to reduce project time and costs. One of the complexities of current embedded systems is the communication among its different parts. This communication is usually made with use of buses, and, since the systems are growing in complexity, bus protocols must be up-to-date in order to meet the timing constraints of such systems, thus needing careful design and analysis.

1.2. Architecture Description Languages

The paradigm of Architecture Description Languages (ADL's) has arisen as an attempt to meet the requirements of modern systems which are increasingly complex, needing a solid blueprint for all the design phases. ADL's represent systems as sets of components that communicate by means of connectors, forming dynamic configurations as needed. They are similar to modeling languages in some aspects, but differ from them mainly in the component-based approach, since modeling languages tend to represent the system as a whole.

In the scope of this work, the representation of the bus systems with components and connectors is highly desirable to have a detailed view of the interaction of the parts and the behavior of each individual component. Thus we have chosen to create architecture specifications with the AADL, an ADL with extensive support to embedded, real-time systems with time constraints.

1.3. Organization of the Paper

This paper intends to expose an incremental methodology to formally describe bus protocols, emphasizing either their architecture or their behavior. Such a methodology facilitates the gradual understanding of the protocols, avoiding too many details in the design levels where they are not needed. The employed methodology is illustrated with the design of the well-studied PCI protocol used in computers. Architectural and behavioral aspects are specified with the Architecture Analysis and Design Language (AADL); while the Temporal Logic of Actions (TLA) is used to specify the protocol behavior in a more adapted manner to model checking tools. Section 2 presents an overview of bus architectures. Our proposed methodology is presented in Section 3 and exemplified in Sections 4 and 5. The conclusions and perspectives drawn by this paper are given in Section 6.

2. Typical Bus Architectures

2.1. Basic Concepts

A bus is, basically, a transmission medium that connects the devices which share it [Stallings 1994]. This connection is made by a certain number of ‘lines’ that may transmit data, control, or both, and are accessible by all the connected devices. Since the bus is, by definition, a shared resource, there must be an access control in order to avoid data overwriting or inconsistencies; usually, one of the devices – a bus controller – manages the use of the bus by the other devices.

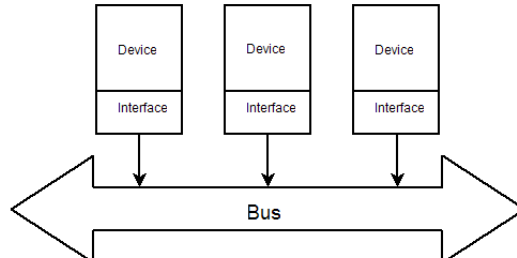


Figure 1. A generic bus

Given the wide range of applications in which buses are employed, their structure may vary. Stallings [Stallings 1994] classifies buses according to their type (dedicated or multiplexed lines), method of arbitration (a centralized or distributed controller) and timing (synchronous or asynchronous), as well as the number of available lines and data transfer types. The behavior of the bus is defined by the control flow among it, its controller and the other devices. Among the main control lines seen in common buses, there are those who deal with bus access requests, I/O and interruption commands, and transfer acknowledgements. The data transmissions across the bus are achieved by two stages: firstly there is an arbitration period, where the bus controller shall decide what component will be able to use the bus and thus become a transfer ‘master’, and a transfer period, where the master sends or receives data from a slave. Figure 2 illustrates these phases in a generic bus.

There are already some previous efforts in formally modeling the behavior of buses. A Colored Petri Net bus model can be seen in [Petrucci et al. 2002] and

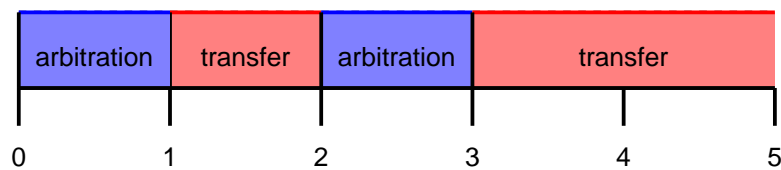


Figure 2. The timeline of a bus transmission

[Kristensen et al. 2002], where it is used in a modeling framework to represent the flow of data in avionics mission systems. Pal [Pal et al. 2004] went further and presented a bus-specific language used to describe protocols and temporal properties. Oumalou [Oumalou et al. 2004] verifies PCI bus properties with SystemC, starting from a UML representation followed by an Abstract State Machine model which is then translated to SystemC.

3. An Incremental Modeling Methodology

Currently, there is a vast number of bus protocols available to several types of application, hence their complexity may also vary, giving origin to very complicated protocols which might be hard to study directly in a low-level specification. Instead of a straightforward modeling of the specific protocol aspects, we prefer to start from a higher level of abstraction where it is already possible to grasp a first view of it.

The basic bus architecture taken into account to our description can be seen as a series of devices and a centralized bus controller, also known as an arbiter, which communicate with the bus lines. If it is wanted to model explicitly the bus lines, it also becomes a component, but it may be assumed that the connections among devices and the controller are the bus itself. Figure 3 illustrates the way this architecture follows the already described generic bus pattern. With this simplified architecture, the bus can be described by means of properties that are independent from a specific protocol, such as the transmission time across the bus or the delay between the sending of a data block by a device and the reception of this data block by the bus.

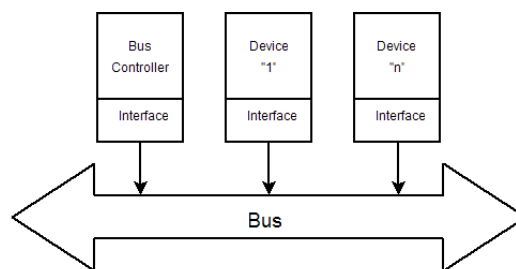


Figure 3. The basic architecture of our bus system

The architecture shall then be refined into a deeper level, where the first protocol aspects start to appear. A protocol is described in the controller and device component specifications, as well as the connections and configurations that they form. At this level, the verification of component properties with respect to actual implementations becomes feasible to the high-level protocol properties. Afterwards, the system architecture can be further refined to present protocol details that are necessary to a more complete property

verification. The protocol phases are then clearly specified, hence new properties can be checked and the already specified ones are translated to be represented adequately in this level.

4. Abstract bus properties

In the highest level of our modeling framework, there are the properties which can be specified without needing to go further than the abstraction level of the Figure 3 architecture. In fact, the AADL language has a predefined bus component, whose internal description relies solely in its abstract properties. The AADL standard properties which apply to buses are:

Allowed_Connection_Protocol, Allowed_Access_Protocol Specify the permitted types of connections (data, event or both) and components (devices, memories or both) that access the bus.

Allowed_Message_Size Specifies a range of permitted message sizes to be transmitted across a bus.

Transmission_Time Used to declare a linear model of the elapsed time between the first and last data bit transmitted.

Propagation_Delay Gives the time taken between the sending of the first data bit to the bus and the reception of this bit by the bus.

Assign_Time, Assign_Byte_Time, Assign_Fixed_Time Specify the time required to move a block of N bytes in the bus. Normally *Assign_Time* is assumed as $(N * Assign_Byte_Time) + Assign_Fixed_Time$.

Some properties, defined by the Software Engineering Institute (SEI) of the Carnegie Mellon University, are also commonly used in AADL specifications at this level:

PowerCapacity, PowerBudget Specify the amount of power provided and required by buses, in multiples of Watts.

BandWidthCapacity, BandWidthBudget Specify the bandwidth provided and required, in multiples of bits per second.

After specifying these abstract properties, it is possible to continue the bus specification by starting to model the protocol aspects. The separate specification of the abstract properties make them more understandable, by encapsulating all other elements, and, a posteriori, these properties may be verified in the lower-level specifications.

5. Bus Protocol Specification

The second level of our methodology consists in adding the protocol aspects inside the bus architectural and behavioral specifications. According to the complexity of the protocol, this level can have several refinement steps to improve the readability of the architecture. We specify the bus protocol as a set of concurrent periodic threads synchronized on a global clock. Moreover, we suppose that the communication between them occur at well-defined times: threads get their input data at the beginning of each period and their output data is made available at the period end. In fact, this concurrent execution model can be considered as the synchronous one seen in [Berry and Cosserat 1984, le Guernic et al. 1991, Caspi et al. 1987]. In the framework

of AADL, this execution model is supported through the use of periodic synchronized threads (same period and logical phase) and communication via data ports.

In the scope of this work, the PCI protocol does not need to be represented in a too complex way, hence we have opted to represent it in two sub-levels: one with the external aspects of the architecture components and other with the behavior of these components.

5.1. The Bus Controller

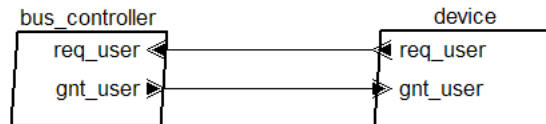


Figure 4. The controller with a connected device

The bus controller is responsible for the bus access management. This control is made with ‘REQ’ and ‘GNT’ lines connected to each device that should be capable of demanding bus control. The REQ lines are used by the connected devices to signal that they request (or release) bus control, and the GNT lines are used by the controller to grant bus control to a device. Figure 4 shows the controller connected to a single device – in an actual application, there will be two or more. The represented components are the processes that contain the specified threads.

```

1 thread bus_controller_thread
2   features — control lines modeled by sets of boolean data ports
3     req_user: in data port behavior::boolean {
4       Multiplicity => value(pci::NbUsers);
5     };
6     gnt_user: out data port behavior::boolean {
7       Multiplicity => value(pci::NbUsers);
8     };
9   properties
10    Dispatch_Protocol => Periodic;
11    Period => 10 ns;
12 end bus_controller_thread;
  
```

These threads are refined in the implementation part, where their behavior is described. If the bus is idle, it makes a non-deterministic choice of a device which sets its req line. It then grants the access by setting the device’s gnt line. Each dispatch of the thread execute transitions until a complete state is reached. It should be noted that fairness is not enforced through such a description. However, we can refine it by, for instance, specifying a round-robin scheduling policy that shall ensure a fair behavior.

```

1 thread implementation bus_controller_thread.i
2   annex behavior_specification {**
3     state variables
4     master : behavior::integer; — the selected master device
5     initial
6     master := 1; — default master
7     states — controller states
8     idle: initial complete state;
  
```

```

9     busy: complete state;
10    transitions
11    — indexed set of transitions processing user requests
12    request(r in 1 .. value(pci::Nbusers)):
13    idle ¬[ on req_user[r] ]→ busy {
14    — signals to the device if it is the master or not
15    for s in 1 .. value(pci::Nbusers) gnt_user[s] := r = s;
16    master := r;
17    };
18    — releasing if the master does not request the bus anymore
19    busy ¬[ on not req_user[master] ]→ idle {
20    for s in 1 .. value(pci::Nbusers) gnt_user[s] := false;
21    }
22    **};
23 end bus_controller_thread.i;

```

The TLA language is used to define the semantics of the AADL specification. Output variables are initialized by the empty value so that updates can be detected.

```

1 VARIABLES req_user , gnt_user , master , state
2 TypeInvariant ≜
3   ∧ state ∈ {"idle", "busy"}
4   ∧ req_user ∈ [ pci_users → BOOLEAN ] \* input data ports
5   ∧ gnt_user ∈ [ pci_users → BOOLEAN ∪ {"empty"} ] \* output data ports
6   ∧ master ∈ pci_users \* who is the master
7 request ≜ \* translation of the AADL request transition
8   ∧ state = "idle"
9   ∧ ∃ r ∈ pci_users:
10     ∧ req_user[r]
11     ∧ gnt_user' = [s ∈ pci_users: ↦ r = s]
12     ∧ master' = r
13   ∧ state' = "busy" ∧ UNCHANGED <<req_user>>

```

5.2. The Devices

The devices here specified represent any kind of device that might be connected in the PCI bus. Since our interest is mostly for the arbitration part of the protocol, only the already seen REQ and GNT ports are specified, as well as a ‘self’ port which contains a unique identifier for each device.

```

1 thread pci_device
2   features
3     self : in data port; — identification of the device
4     req_dev: out data port behavior::boolean;
5     gnt_dev: in data port behavior::boolean;
6   properties
7     Dispatch_Protocol => Periodic;
8     Period => 10 ns;
9 end pci_device;

```

As in the controller, the devices are also refined with an implementation declaration. A device may be idle, awaiting the granting of bus control (request state), or in a data transfer as master or slave. When idle, the device may request bus control or act as a slave of a transfer. In the request state, the device may receive the bus control and become master, or be the slave of another transfer in progress. In the slave state, the device may

return idle, or, if it has already requested bus control, go to the request state, or, if it is the next transfer master, go directly to the master state. After a transfer, the master gets idle again. The states represented with AADL may also be seen with UML statecharts, as in the Figure 5.

```

1 thread implementation pci_device.i
2 annex behavior_specification {**
3   states
4     idle: initial complete state;
5     request: complete state;
6     master: complete composite state;
7     slave: complete composite state;
8     ...

```

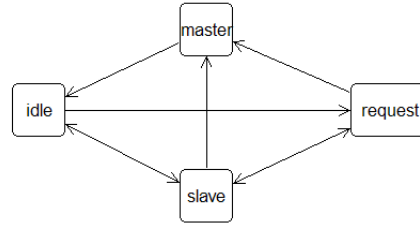


Figure 5. The device statechart

In order to ensure a fair scheduling of requests, we provide the following refinement of the transition request in the TLA specification:

```

1 next_turn  $\triangleq$  \* computes the next master wrt round robin (start from master)
2 IF  $\exists r \in \text{pci\_users} : \text{req\_user}[r] \wedge \text{master} < r$  THEN
3   min( $\{r \in \text{pci\_users} : \text{req\_user}[r] \wedge \text{master} < r\}$ )
4 ELSE min( $\{r \in \text{pci\_users} : \text{req\_user}[r]\}$ )
5 request  $\triangleq$ 
6  $\wedge \text{state} = \text{"idle"} \wedge \exists r \in \text{pci\_users} : \text{req\_user}[r]$ 
7  $\wedge \text{gnt\_user}' = [s \in \text{pci\_users} \mapsto s = \text{next\_turn}]$ 
8  $\wedge \text{master}' = \text{next\_turn}$ 
9  $\wedge \text{state}' = \text{"busy"} \wedge \text{UNCHANGED} \ll \text{req\_user} \gg$ 

```

Note that the TLA specification ignores timing aspects. In fact, a synchronous subset of AADL has been used, which allows a translation to TLA where actions performed by threads during a single dispatch are mapped to a unique TLA transition. So, a TLA step corresponds to a period, here 10 ns, which allows the verification of time-relative abstract bus properties.

6. Conclusions and Future Work

This paper described a methodology to model bus protocols. We have used the AADL execution model since it is recognized for its real-time precise semantics, inherited from MetaH [Vestal 1998]. We could have also used UML by adopting a real-time profile (UML-RT [OMG 2002], MARTE [ProMARTE], SPT). Future works might include a more detailed study of the arbitration protocols, as well as the utilization of another step in the methodology, consisting in the specification of an actual implementation of the protocol. Then, the verification of abstract bus properties relies on their precise definition in TLA.

In the scope of this work, we have translated the AADL specifications to TLA manually. A possible follow-up of this study would be to implement such translations within the TOPCASED [TOPCASED] project which offers metamodelling and model transformation tools. Automated translation tools to model checkers like TINA [Berthomieu and Vernadat 2002] and CADP [Garavel et al. 2002] are also planned in this framework.

References

- AS5506 (2004). *Architecture Analysis & Design Language (AADL) AS5506*. Society of Automotive Engineers.
- Berry, G. and Cosserat, L. (1984). The ESTEREL synchronous programming language and its mathematical semantics. volume 197, pages 389–448, Berlin, Germany. Springer-Verlag.
- Berthomieu, B. and Vernadat, F. (2002). TINA: Time petri Net Analyzer V2-4. <http://www.laas.fr/tina,LAAS/CNRS>.
- Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. (1987). Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188.
- Garavel, H., Lang, F., and Mateescu, R. (2002). An overview of CADP 2001. *European Association for Software Science and Technology*, 4:13–24.
- Kristensen, L. M., Billington, J., Petrucci, L., Qureshi, Z. H., and Kiefer, R. (2002). Formal Specification and Analysis of Airborne Mission Systems. In *The 21st Digital Avionics Systems Conference*, Irvine, USA.
- Lampert, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- le Guernic, P., Gautier, T., le Borgne, M., and Maire, C. L. (1991). Programming real-time applications with SIGNAL. In *Proceedings of the IEEE*, volume 79.
- OMG (2002). UML profile for schedulability, performance, and time specification.
- Oumalou, K., Habibi, A., and Tahar, S. (2004). Design for Verification of a PCI Bus in SystemC. In *Proceedings of the 2004 International Symposium on System-on-Chip (SOC '04)*, pages 201–204. IEEE.
- Pal, B., Banerjee, A., Dasgupta, P., and Chakrabarti, P. (2004). The BUSpec Platform for Automated Generation of Verification Aids for Standard Bus Protocols. In *Formal Methods and Models for Co-Design – MEMOCODE '04*, San Diego, USA.
- Petrucci, L., Kristensen, L. M., Billington, J., and Qureshi, Z. H. (2002). Towards Formal Specification and Analysis of Avionics Mission Systems. In *CRPIT '02: Proceedings of the Conference on Application and theory of Petri Nets*, pages 95–104, Darlinghurst, Australia. Australian Computer Society, Inc.
- ProMARTE. ProMARTE Working Group. <http://www.promarte.org>.
- Stallings, W. (1994). *Computer Organization and Architecture - Designing for Performance*. Prentice Hall.
- TOPCASED. Toolkit in Open-source for Critical Applications & Systems Development. <http://www.topcased.org>.
- Vestal, S. (1998). *MetaH User's Manual*. Honeywell Technology Drive. <http://www.htc.honeywell.com/metah/uguide.pdf>.