Deterministic Data Flow Communication in AADL

Mohamed Yassin Chkouri Marius Bozga
Verimag, Centre Equation - 2, avenue de Vignate 38610 GIERES

Abstract

Architecture Analysis and Design Language (AADL) is used to describe both the hardware and software architecture of an application, at system-level. However, AADL data flow communication between components is not deterministic and limits the utility of AADL for critical systems.

This paper provides a general protocol to ensure deterministic data flow communication between threads in AADL. We also present a methodology for translating the architecture into an executable model, which can be simulated and validated together.

1 Introduction

AADL [3] provides the structure of component-based systems as an assembly of software components mapped onto an execution platform. Thus, AADL is used to describe functional interfaces and performance-critical aspects of components. It describes the dynamic behavior of the runtime architecture as well as how components interact. The language is designed to be extensible to accommodate analysis of runtime architectures.

One of the properties that critical systems must usually exhibit is determinism, i.e., any two runs of the system with the same inputs should produce the same outputs. In fact, deterministic execution is one of the pre-requisites for creating high-integrity applications such as control or embedded systems. The main reason is that the validation is usually performed by testing the implementation of the system and testing non-deterministic implementation is difficult. This is due to the fact that non-determinism can make the tests non-reproducible and the test coverage is hard to define and to measure.

Unfortunately, data flow communication between AADL components is not deterministic, thus leading to non-deterministic execution in general and preventing the use of AADL for most critical systems. For example, consider a system with two periodic threads $T_A$ and $T_B$, with periods $P_A$ and $P_B$ respectively where $P_B = 3 \times P_A$. In the case where the period of the sender thread is smaller than the period of the receiver thread e.g., if $T_A$ produces data $u$ to be consumed by $T_B$, then the non-determinism shown in Figure 1 may result. In the first hyperperiod, $T_B$ is launched just after $T_A$ finishes its first job and writes $u_1$, this value is read by $T_B$ as the value of the data flow. In the next hyperperiod, $T_B$ is executed after the second job of $T_A$ and it reads $u_5$ instead of the expected $u_4$.

![Figure 1. Period of sender < Period of receiver](image1.png)

In the opposite case, the period of the sender thread is greater than the period of the receiver thread. If $T_B$ produces a data flow $v$ to be read by $T_A$, then the non-determinism shown in Figure 2 may occur: in the first hyperperiod, the first job of $T_A$ consumes the last data $v_0$ from the last hyperperiod and the second job of $T_A$ consumes $v_1$. However, in the next hyperperiod, the first and second job of $T_A$ consumes the last data of the last hyperperiod $v_1$. Obviously, this type of communication non-determinism is

---

*This work is partially supported by ITEA/Spices and OpenEMBeDD projects
unacceptable in high-integrity applications.

Our first contribution in this paper is a time-triggered protocol for enforcing deterministic data flow communication among AADL threads. It requires the existence of a unique global clock for all the threads in the system. Then, it enforces the following communication discipline: if the sender thread produces data at every dispatch, this data is sent only at the end of its period. Conversely, the receiver thread receives the data available precisely at the beginning of its period and may consume it later within the period.

In addition to testing, which clearly benefits from deterministic execution, it is also possible to improve capabilities of model-based analysis. We have shown in [14], how AADL systems can be automatically translated into the input language of the BIP (Behavior Interaction Priority) framework [7] and analyzed using the BIP toolset. Our second contribution is an extension of our translation, in order to include deterministic communication. More precisely, we will restrict non-determinism by introducing additional atomic components - called DBCI/DBCD (Deterministic BIP Communications for Immediate/Delayed connection) - to the model. This way of controlling non-determinism has the following advantages over built-in syntactic constructs:

- the non-determinism is clearly localized and controlled, and
- the non-determinism can be kept/eliminated, simply by removing/adding DBCI/DBCD components.

This paper is organized as follows. Section 2 gives an overview of AADL. In Section 3, we present our method to solve the non-deterministic data flow communication. Then, we present the translation from AADL to BIP and the modelization of deterministic data flow communication in Section 4. In Section 5, we illustrate our case study. Related work and conclusion closes the article in Section 6.

2 AADL

The Architecture Analysis & Design Language (AADL) [3] is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. AADL is standardized by the Society of Automotive Engineers (SAE), it plays a central role in several projects such as Topcased [5], OSAE [4], etc. A system modelled in AADL consists of application software mapped into an execution platform. In AADL, data, subprograms, threads, and processes collectively represent application software. They are called software components. Processor, memory, bus, and device collectively represent the execution platform. They are called execution platform components. Execution platform components support the execution of threads, the storage of data and code, and the communication between threads. Systems are called compositional components. They permit software and execution platform components to be organized into hierarchical structures with well-defined interfaces.

In this paper, we will focus on data port connection between periodic threads.

2.1 Threads

A thread represents a sequential flow of control that executes instructions within a binary image produced from source text. A thread always executes within a process. A scheduler manages the execution of a thread.

A thread type declaration contains ports such as data port, event port, and event data port, subprogram declarations, and property associations. A thread component implementation contains data declarations, a calls subclause, annex behavior, and thread property associations.

Thread properties are used to represent attributes and other characteristics, such as the period, dispatch protocol, deadline of the threads, etc. The Deadline property specifies the maximum amount of time allowed between a thread dispatch and the time that thread begins waiting for another dispatch. The Period property specifies the time interval between successive dispatches of a thread. Dispatch protocol is a property which defines the dispatch behavior for a thread. Four dispatch protocols are supported in AADL: periodic, aperiodic, sporadic, and background with the following meaning:

- **periodic** means that the thread must be activated according to the specified period;
- **aperiodic** means that the thread is activated via one of the other components output port, called an event port;
- a **sporadic** thread is a mixture between aperiodic and periodic: it can be activated either by events, or periodically;
- **background** thread is always active, but have the lowest priority.

2.2 Data Connection

A data port connection is a linkage that represents communication of data between components. This can be the transmission of data between a sender thread and a receiver thread. Ports are directional. An out port represents output provided by the sender and an in port represents input needed by the receiver. A data port connection is declared to be immediate denoted by ”->” or to be delayed denoted by ”--->”. The intuitive meaning is the following:
thread ThreadA
features
  outp : out data port integer;
properties
  Dispatch_protocol => Periodic;
  Period => 100ms;
end ThreadA;

thread ThreadB
features
  inp : in data port integer;
properties
  Dispatch_protocol => Periodic;
  Period => 100ms;
end ThreadB;
process Partition
end Partition;

process implementation Partition.Impl
subcomponents
  T1 : thread ThreadA;
  T2 : thread ThreadB;
connections
  immediate cnx : data port
  T1.outp => T2.inp;
end Partition.Impl;

Figure 4. AADL thread and process

Immediate Connection: For immediate data port connections the data transmission is dispatched when the source thread completes computation. Immediate connections are illustrated in Figure 3. Thread A and Thread B are two periodic threads executing at a rate of 10Hz, i.e., they are logically dispatched every 100 ms. For immediate connection, the actual start of execution of the receiving thread (Thread B) will be delayed after its dispatch event until the sending thread (Thread A) completes execution and its output data value has been transferred into the input port of the receiving thread. For the two threads illustrated in Figure 3, a textual specification is shown in Figure 4.

Delayed Connection: For delayed data port connections, the data transmission is initiated at the deadline of the source thread. This is shown in Figure 5. The output of Thread A is made available to Thread B at the beginning of its next dispatch. Figure 5 shows the same system as Figure 3 with the difference in the declaration of connection type (immediate or delayed).

3 Deterministic Data Flow Communication

In this section, we explain in detail the data flow communication in AADL and provide a communication protocol that ensures determinism. Without loss of generality, we consider systems of the following form:
- The system is real-time. Periodic tasks are guaranteed to be dispatched at each period;
- The source and destination threads have a Dispatch Protocol => Periodic;
- The connection must be a data connection.

3.1 Non-deterministic Data Flow Communication

We will illustrate the non-deterministic data flow communication in a simple example. From the set of periodic threads in a system, consider two threads \( T_A \) and \( T_B \) with periods \( P_A = 20 ms \) and \( P_B = 30 ms \), respectively.

3.1.1 Immediate data connection:
- The period of sender is smaller than the period of receiver: Thread \( T_A \) produces data \( u \) to be consumed by thread \( T_B \), then the non-determinism as shown in Figure 6 may result. In the first hyperperiod, \( T_B \) is launched just after \( T_A \) finish it first execution and produces data \( u \), which is consumed by \( T_B \) as the data of the data flow. In the next hyperperiod, \( T_B \) consumes the last data from the first hyperperiod.
- The period of sender is greater than the period of receiver: Thread \( T_B \) produces data \( v \) to be consumed by thread \( T_A \), then the non-determinism as shown in Figure 7 may result. In the first hyperperiod, \( T_A \) consumes the last data \( v_0 \) from the last hyperperiod. In the next hyperperiod, \( T_B \) is launched before \( T_A \) and produces data \( v_3 \), which is consumed by \( T_A \) as the data of the data flow.
3.1.2 Delayed data connection:

- The period of sender is smaller than the period of receiver: Thread $T_A$ produces data $u$ to be consumed by thread $T_B$, then the non-determinism as shown in Figure 8 may result.
- The period of sender is greater than the period of receiver: Thread $T_B$ produces data $v$ to be consumed by thread $T_A$, then the non-determinism as shown in Figure 9 may result.

3.2 Time Triggered Communication Protocol

The existing approach to solve the problem of non-determinism in AADL is to use data of the last hyperperiod [13]. The drawback of this approach is that if the hyperperiod is very long, then data is used rarely. Consequently, this approach is not recommended in control applications.

For this reason, we propose a new protocol, that consist of the following two rules:

1. The data produced by sender thread is made available only at the end of its period.
2. The data is consumed by receiver thread at the next period that occurs at or after the sender thread deadline, as shown in Figure 10.

The protocol works for all possible cases:

- **Immediate connection (IC):** In this case, we apply the two rules. For example, Figure 11 shows our approach applied to immediate data connection. All the data produced by thread $T_A$ is available at the end of its period. This data are consumed by thread $T_B$ at the beginning of its period that occurs at or after the thread $T_A$ deadlines.

- **Delayed connection (DC):** In this case, we apply just the second rule, because the first rule is enforced in the semantics of delayed data connection. For example, Figure 12 shows our approach applied to delayed data connection.

This protocol allows us to make sure we take the good data, and eliminates non-determinism as shown in the next section.
3.3 Correctness of Communication Protocol

In this section, we provide a formal argument for the correctness of our protocol. Let us first introduce some notations. The notation $T_A(i) \rightarrow u(t)$ signifies that the $i^{th}$ activation of the thread $T_A$ produces data available at time $t$. Similarly, the notation $u(t) \rightarrow T_B(j)$ signifies that data available at moment $t$ will be consumed by the $j^{th}$ activation of thread $T_B$. With this notation, our protocol states that:

$$\forall i \geq 1: T_A(i) \rightarrow u(i \cdot P_A)$$

$$\forall j \geq 1: u((j-1) \cdot P_B) \rightarrow T_B(j)$$

From these two rules, we obtain that:

$$\exists i \geq 1: T_A(i) \rightarrow u(i \cdot P_A) = u(j \cdot P_B) \rightarrow T_B(j)$$

In fact, the unique $i$ satisfying condition is such that:

$$(j-1) \cdot \frac{P_B}{P_A} < i \leq j \cdot \frac{P_B}{P_A}$$

This relation ensures the deterministic data flow communication between the two threads: every data consumed by one activation ($j$) of the receiver thread corresponds to a uniquely identified activation ($i$) of the producer thread.

For example, we take the case when the period of sender thread is smaller than the period of receiver thread (immediate connection) as shown in Figure 13. Thread $T_A$ produces every 20ms a new data $u$. This data is available at the end of its period: $T_A(1) \rightarrow u(20), T_A(2) \rightarrow u(40), T_A(3) \rightarrow u(60), T_A(4) \rightarrow u(80), T_A(5) \rightarrow u(100), T_A(6) \rightarrow u(120)$.

This data are consumed by thread $T_B$ every 30ms at the beginning of its period that occurs at or after the thread $T_A$ deadlines: $u(0) \rightarrow T_B(1), u(20) = u(30) \rightarrow T_B(2), u(60) \rightarrow T_B(3), u(80) = u(90) \rightarrow T_B(4)$.

4 Experimentation Using BIP Framework

The translation from AADL [3] into BIP [7] is described in [14]. In this section, we present an overview of BIP, the modelization of thread, data connection and modelization of deterministic data port connection.

4.1 The BIP Component Framework

BIP (Behavior Interaction Priority) is a framework for modeling heterogeneous real-time components [7]. The BIP framework consists of a language and a toolset including a frontend for editing and parsing BIP programs and a dedicated platform for model validation. The platform consists of an Engine and software infrastructure for executing models. It allows state space exploration and provides access to model-checking tools of the IF toolset [11] such as Aldebaran [10], as well as the D-Finder tool [9]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability. The BIP language allows hierarchical construction [12] of composite components from atomic ones by using connectors and priorities. Several case studies were carried out such as an MPEG4 encoder [15], TinyOS [8], and DALA [6].

4.2 Modelization of Threads

An AADL thread has been modelled in BIP by an atomic component as shown in Figure 14. The initial state of the thread is HALTED. On an interaction through port load the thread is initialized. Once initialization is completed the thread enters the READY state, if the thread is ready for an interaction through the port req_exec. Otherwise, it en-
Figure 14. BIP model for thread behavior

ters the SUSPENDED state. When the thread is in the SUSPENDED state it cannot be dispatched for execution.

When in the SUSPENDED state, the thread is waiting for an event and/or period to be activated depending on the thread dispatch protocol (periodic, aperiodic, sporadic). In the READY state, a thread is waiting to be dispatched through an interaction in the port get_exec. When dispatched, it enters the state COMPUTE to make a computation. Upon successful completion of the computation, the thread goes to the OUTPUTS state. If there are some out_ports to dispatch the thread returns to the OUTPUTS state, otherwise, it enters the FINISH state.

4.3 Modelization of Data Connection

In our previous translation, AADL data port connection has been modelled on BIP depending on the categories:

- **Immediate data connection** is translated into a strong synchronization between the corresponding ports with transfer of data.

- **Delayed data connection** is translated into an atomic component that takes as input a data from thread sender and wait for end of period of sender to send a data to the receiver thread. This atomic component ensure the delayed connection between the two threads.

However, this modelization does not ensure the determinism. In order to obtain deterministic communication, we rely on two atomic BIP components:

- The first atomic component named get_data is shown in Figure 15. The role of this component is to delay the transfer of data at the end of the period of sender thread. The get_data component has as input in_data produced by the sender thread ($T_A$), data store to save in_data through an interaction in the port in_data_port, and send data store through an interaction in the port out_data_port when the period ($P_A$) of sender thread ($T_A$) is achieved.

- The second atomic component named set_data is shown in Figure 16. The role of this component is to delay transfer of data at the next period that occurs at or after the thread $T_A$ deadline. It takes as input data produced by get_data and stores in a data store through an interaction in the port in_data_port. If it receives new data before the end of the period $P_B$ of receiver thread ($T_B$), then the old data is replaced by the new one through an interaction in the port in_data_port. Otherwise, if the period $P_B$ is achieved, then it sends data out_data through an interaction in the port out_data_port to the thread $T_B$.

The full BIP description of set_data and get_data components of Figures 15 and 16 is available in [1].

Using the above atomic components, we ensure deterministic communication between AADL threads as follows:

- **Immediate data connection**: We call a Deterministic BIP Communications for Immediate connection by DBCI, the pair of atomic components (get_data & set_data) that implements a data flow between the two threads. The communication between the two threads and DBCI are shown in Figure 17 (This BIP model is the modelization of the system as shown in Figure 3).
• **Delayed data connection:** We call a Deterministic BIP Communications for Delayed connection by DBCD, the atomic component set_data that implements a data flow between two threads. The communication between two threads and DBCD is shown in Figure 18 (This BIP model is the modelization of the system as shown in Figure 5).

The type of BIP connectors between threads and DBCI/DBCD is strong synchronization with transfer of data from in ports to out ports.

4.4 **Integration of DBCI/DBCD in AADL2BIP**

The AADL2BIP tool [1] generating BIP from AADL has been implemented in Java, as a set of plugins for the open source Eclipse platform. It takes as input an AADL model(.aaxl) conforming to the AADL metamodel and generates a BIP model conforming to the BIP metamodel [2].

In [14], we gave a mapping for translating AADL component, connections, ports, etc, into BIP. This translation allows simulation of systems specified in AADL and application to these systems of formal verification techniques developed for BIP, e.g., deadlock detection.

We extend our translation by integrating the DBCI/DBCD into our tool AADL to BIP [14]. This extension allow us to make a deterministic execution and to improve capabilities of model-based analysis. The following conditions are tested before instantiating a DBCI/DBCD:

1. The connection must be a data connection (Immediate or delayed), not an event or event data connection and have the property DBCI/DBCD true.

2. Both threads must satisfy the property association $\text{Dispatch\_Protocol} \Rightarrow \text{Periodic}$.

5 **Case Study**

We used an AADL case study to check the feasibility of our method. Consider a simple application consisting of three threads sensors with period $= 20\text{ms}$, one thread Data_Fusion with period $= 30\text{ms}$, and one Actuator that works at 20ms. All the threads are periodic ($\text{Dispatch\_Protocol} \Rightarrow \text{Periodic}$). The application architecture is as shown in Figure 19 (for more details about source code and execution see [1]).

- **Sensor_A, Sensor_B and Sensor_C** increments an integer variable and sends it periodically to Data_Fusion over its data port.

- **Data_Fusion** reads the values of its data ports, adds them, and periodically sends the sum to the Actuator thread. The thread Data_Fusion control its inputs. When the variables received from Sensor_A, Sensor_B and Sensor_C are not equal, then the thread prints an error statement. Otherwise, it does nothing.

- **Actuator** reads and print the value of its data port at every dispatch.

Simulation: The AADL model of the control system is transformed into BIP automatically by using our AADL to BIP translation tool. We execute the BIP model. The Data_Fusion thread prints out an error message each time it encounters an unexpected situation: the values received from Sensor_A, Sensor_B and Sensor_C are not equal.
These errors are due to the non-deterministic data flow communication between threads. In order to fix this problem, we generate the BIP model using a DBCI component. This time, the errors at execution are completely eliminated.

Figures 20 and 21, describes the data received by thread called `DataFusion` from `sensorA`, `sensorB`, and `sensorC` every 30ms. We see clearly the difference between these two figures. Figure 20 shows a correct behavior when we use a DBCI component: the three input of `DataFusion` have always the same values. However, Figure 21 shows an incorrect behavior: the inputs of `DataFusion` have different values (for each execution).

6 Conclusion

To the best of our knowledge, there is one proposal to solve the non-determinism in AADL. The work of [13] solves this problem but partially. It relies on the analysis of the hyperperiod to make a transfer of data. The drawback of this approach is that the use of hyperperiod is not efficient when the hyperperiod is very long. Moreover, the method is restricted to the case where the ratio between the long and the short periods is an integer. For example, the case when: \( P_{long} = 30 \) and \( P_{short} = 20 \) leading to \( r = 1.5 \) is not accepted by [13].

We propose a general time-triggered protocol to ensure deterministic data flow communication between periodic threads in AADL. This protocol is integrated in our AADL to BIP tool chain allowing for the translation of AADL into executable models, which can be simulated and validated.

References