Real-time Connectors for Deterministic Data-flow

Irfan Hamid*, Elie Najm*
*GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
Email: {firstname.lastname}@enst.fr

***Abstract*— In this paper we introduce deterministic bridge connectors, a type of construct that ensures deterministic data-flow communication in asynchronous real-time systems. We also present a methodology for generating these connectors automatically from the application's architecture description in order to reduce programmer effort and the chance of error. Finally, we provide a process algebraic verification of the determinism property of these connectors.**

***Index Terms*— Ada, Ravenscar, real-time, code generation, connectors, AADL.**

I. Introduction

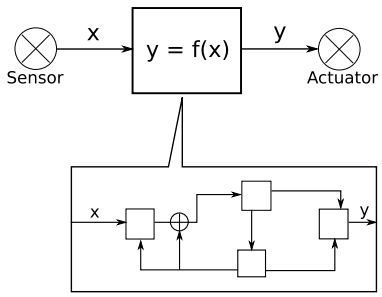
Vehicle control systems are one of the most safety-critical categories of software. Stringest standards of code review must be met before deployment on-board the platform. Such systems are invariably implemented in the form of control laws that take input data from sensors, carry out transformations on that data and apply the outputs to actuators (figure 1). There are—invariably—two orthogonal sets of requirements that are always leveraged on such systems:

- Timing requirements that relate to the timeliness of the produced results. The validity of results is strongly dependant upon the time at which they are available.
- Functional requirements that describe the output of the system as a function of the input provided, i.e.: they describe the algorithmic portion of the control law's response loop.

One of the properties that these systems must always exhibit is determinism, i.e.: two equivalent runs with the same inputs should produce the same results, every time. One approach to achieving this determinism is to use a synchronous reactive language such as Lustre [1] to implement the control laws. In this paradigm there is only one thread of execution and the entire functionality of the control system is *collapsed* into it. The various *functional nodes* of the model are then executed in a synchronous lockstep fashion. Another approach is to write asynchronous multithreaded control applications, which run atop an operating system and behave in an asynchronous manner, communicating with each other [2]. The advantages of a multithreaded asynchronous control system is the possibility of higher processor utilization as well as flexibility of adding threads later in the design stage. This approach also allows designers to *piggyback* non-control system threads such as alarm and status monitors onto the same partition.

In all control systems, the communication is carried out in the form of dataflows as this is the paradigm that is most native to control law concepts. The system architecture is usually represented in a block structured formalism. Such an architecture is shown in figure 1. The blackbox above computes the *transfer function*. In an asynchronous control system this would be refined to a set of periodic threads that exchange data among themselves to compute the final value of the output (as shown in the lower part of the figure).

Fig. 1. Control loop and its block diagram representation



The simplest method of implementing a dataflow between two co-located threads is to use a shared variable protected against concurrent access by a mutex. However, this introduces the danger of non-determinism in the dataflow, due normally to auxiliary load on the system. From the set of periodic threads in a system consider two (τ_s and τ_l) with periods P_{short} and P_{long} respectively such that $P_{long} = 3 \times P_{short}$. The priority of τ_s is higher than that of τ_l, thus, the single job of τ_l in each mutual hyperperiod is executed between jobs of τ_s. If τ_s produces data α to be consumed by τ_l then the non-determinism shown in figure 2 might result: in the first hyperperiod, τ_l is launched just after τ_s finishes its first job and writes α_1, this is read by τ_l as the value of the dataflow. In the next hyperperiod, due to extraneous factors—such as a sporadic alarm thread firing—τ_l is executed after the *second* job of τ_s and it reads α_5 instead of the expected α_4.

Vice versa, if τ_l produces a dataflow β to be read by τ_s then the following non-determinism might occur (also shown in figure 2): in the first hyperperiod, τ_l finishes *after* the last job of τ_s and thus all three jobs of τ_s in this hyperperiod use β_0 (an initial value). However, in the next hyperperiod, τ_l finishes just before last job of τ_s and that job reads β_2 instead of β_1.

This type of non-determinism is unacceptable in control applications. To guard against this breach, the protocol given in figure 3 is widely used in industry. All dataflow values are taken from the last job of the previous mutual hyperperiod of both threads involved in the dataflow. This eliminates non-determinism due to auxiliary load or scheduling decisions by the RTOS.

Fig. 2. Non-determinism in dataflow across two hyperperiods.

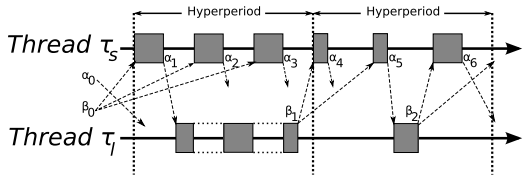
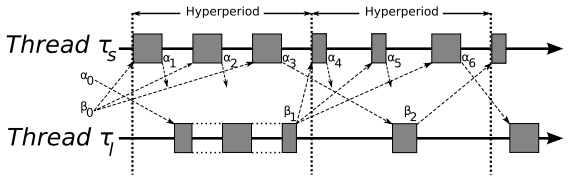


Fig. 3. A delayed communication protocol that ensures determinism.



We present an efficient method for implementing this type of deterministic dataflow among hard real-time threads using a connector abstraction. The protocol is given in section II and its properties are explained. Our proposed solution to implement the protocol is given in section III. The details of implementation as well as the tool support developed are given in section IV. The verifications carried out are detailed in section V. Finally, the article is rounded up with our conclusions and discussion of future work.

II. Deterministic Protocol and its Properties

In this section we will formulate the mathematical properties of the dataflow. But before that can be done, we need to explicitly state the assumptions and hypotheses that can be made upon the system and the consequences thereof:

- The system is hard real-time. Periodic tasks are guaranteed to be dispatched at the start of each period (dispatching is taken to mean being put in the ready state, not necessarily being given the processor).
- Deadlines are met, otherwise the schedulability has been breached and the system should go into a graceful degrade mode.
- Both the source and destination threads are periodic with $P_{long} = r \times P_{short}$ and $r \neq 1$.
- The source thread writes to the dataflow at each job, the destination thread reads from the dataflow at each job, this is a reasonable assumption for a control system.
- The scheduler is preemptive and priority based.
- Priority assignment is rate monotonic [3], i.e.: priority of τ_s is greater than the priority of τ_l. Thus, at the start of each mutual hyperperiod P_{long} the thread τ_s will be *executed* before τ_l, even though both were made ready simultaneously.

A. Protocol

Given the above assumptions we can obtain a mathematical model for the dataflow protocol such that determinism is maintained. For the dataflow from a high-frequency to low-frequency thread every r^{th} write from the source thread is used. Similarly, for a dataflow from a low-frequency to high-frequency thread every write from the source thread is read r times. The notation $\tau_l\ (i) \leftarrow \alpha_j$ signifies that the i^{th} job of thread τ_l gets j^{th} instance of the dataflow (α_j). For the example of figure 2 where $P_{long} = 3 \times P_{short}$, the first few dataflows are:

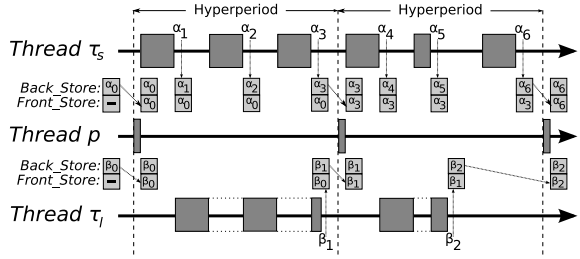
$$\begin{aligned} &\tau_s(1) \leftarrow \beta_0,\ \tau_s(2) \leftarrow \beta_0,\ \tau_s(3) \leftarrow \beta_0,\ \tau_s(4) \leftarrow \beta_1 \\ &\tau_l(1) \leftarrow \alpha_0,\ \tau_l(2) \leftarrow \alpha_3,\ \tau_l(3) \leftarrow \alpha_6,\ \tau_l(4) \leftarrow \alpha_9 \end{aligned} \tag{1}$$

We can generalize this into the following equations:

$$\left.\begin{aligned} \forall i : \tau_s(i \times r + 1) \leftarrow \beta_i \ldots \tau_s(i \times r + r) \leftarrow \beta_i \\ \forall i : \tau_l(i) \leftarrow \alpha_{r \times (i-1)} \end{aligned}\right\} r = P_{long}/P_{short} \tag{2}$$

These equations collapse correctly for the degenerate case of $r = 1$ by stipulating that the value from the previous job of the source thread be used. The previous job falls in the previous hyperperiod by definition as there is one job of each thread in one hyperperiod for $r = 1$.

Fig. 4. Protocol thread p transfering values in the double buffer at the start of each mutual hyperperiod.



B. Suboptimal Implementation

A suboptimal—yet correct—manner of implementation would be to introduce a special thread to assure the protocol. The dataflow would be implemented via a double buffered variable protected by a mutex. The source thread would write to the back buffer and the destination thread would read from the front buffer. A special protocol thread with priority greater than that of both the producer and consumer threads and period equal to P_{long} would be created. This thread would copy the back buffer to the front buffer at the start of each mutual hyperperiod. The evolution of such a system is shown in figure 4. This approach has multiple drawbacks:

- Increases the number of threads in the system.
- Introduces an impact from the non-functional architecture (periods of the two threads, the type of the dataflow etc.) towards the functional code of the protocol thread.
- Involves direct manipulation of priorities (thread p has longer period but higher priority than τ_s, non-conformant with RMA).

III. Deterministic Bridge Exchangers

We propose an alternative approach for the implementation of the dataflow protocol. Our proposal consists of using a connector abstraction. Connectors are defined in [4] as:

Connectors mediate interactions among components: that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.

One of the main advantages of using a connector abstraction is that we can clearly separate components and communication mechanisms, which are two orthogonal concerns of a system. Our dataflow presents an ideal case where the components are the threads that participate in the dataflow. If we can subsume the functionality of the protocol into a certain connector then we can eliminate any pollution of functional code with communication logic.

We call a *stepper deterministic bridge exchanger* (stepper DBX) the connector that implements a dataflow from a high-frequency thread to a low-frequency thread. Conversely, we call the connector that implements a dataflow from a low-frequency thread to a high-frequency one a *stagger deterministic bridge exchanger* (stagger DBX). Both types of connectors:

- Have an internal double buffer of the same data type as the dataflow, called the *back store* and the *front store*.
- Expose procedures `Set_Value` and `Get_Value`, these procedures are concurrency safe.

Below we give an abstract overview of both types of connectors. Detailed implementation and their integration into our existing tools is given in section IV.

A. Stepper DBX

This connector serves to implement a dataflow from a high-frequency thread, τ_s, to a low-frequency thread τ_l. The source thread computes a new value and calls `Set_Value` on the connector each time it is dispatched. The destination thread calls `Get_Value` each time it is dispatched to read the dataflow. From the assumptions made on the system we know that in each mutual hyperperiod P_{long}, there will be $r = P_{long}/P_{short}$ invocations of `Set_Value` and one invocation of `Get_Value`. Also, at the start of each P_{long}, it will be `Set_Value` which will be invoked first, as τ_s has a higher priority.

In order to implement the deterministic dataflow protocol, a stepper DBX embeds buffer handling logic into the `Set_Value` procedure. This procedure must save the data provided by the source thread as well as copy data from the back store to the front store at every invocation corresponding to the first job in a P_{long} hyperperiod.

1) On the first dispatch of every hyperperiod—i.e.: on invocation number $\forall i \in \mathbb{N} : i \times r + 1$ of `Set_Value`—the procedure copies the back store into the front store.
2) On the last dispatch of every hyperperiod—i.e.: on invocation number $\forall i \in \mathbb{N} : i \times r + r$ of `Set_Value`—the procedure copies the provided data into the back store.
3) Every invocation of `Get_Value` returns the front store to the caller (the destination thread).

The procedure `Get_Value` simply returns the value of the front store to the calling thread. The pseudocode for the procedure `Set_Value` is given below.

We call this connector a stepper exchanger because it *steps over* a certain number of input data to the dataflow. It is apparent that only every r^{th} data value is copied into the back store.

```
Procedure Set_Value(in Data)
  Invocation: static integer initialized to 0
  Data      : input for dataflow, provided by source thread
  r         : integer corresponding to P_long/P_short
  begin
    Invocation ← Invocation + 1
    if Invocation = 1 then
      Front_Store ← Back_Store
    end
    if Invocation = r then
      Back_Store ← Data
      Invocation ← 0
    end
    return
  end
```

B. Stagger DBX

A stagger DBX implements a deterministic dataflow from a low-frequency thread to a high-frequency thread. The `Set_Value` and `Get_Value` procedures of a stagger DBX are called by the threads τ_l and τ_s respectively. Therefore, in accordance with the properties enunciated, we can deduce that it is the `Get_Value` procedure that is invoked at the start of each mutual hyperperiod P_{long}. In each P_{long}, there will be $r = P_{long}/P_{short}$ jobs of τ_s and the same number of invocations of `Get_Value`. Conversely, in each P_{long} there will be one job of τ_l and one invocation of `Set_Value`.

In order to implement the deterministic dataflow protocol, a stagger DBX embeds buffer handling logic into the `Get_Value` procedure. This procedure must return the data present in the front store as well as copy the back store to the front store at every invocation corresponding to the first job in a P_{long} hyperperiod.

1) On the first dispatch of every hyperperiod—i.e.: on invocation number $\forall i \in \mathbb{N} : i \times r + 1$ of `Get_Value`—the procedure copies the back store into the front store.
2) Every invocation of `Set_Value` copies the data provided by the source thread into the back store.

We call this connector a stagger exchanger because it *staggers* on a dataflow value a certain number of times before getting a new one. The procedure `Set_Value` simply copies the value passed by the thread into the back buffer of the exchanger. The pseudocode for the procedure `Get_Value` is given below.

IV. Implementation and Tooling

We integrated the stepper and stagger DBX into ARC [5], our code generator that transforms system architectures specified in the AADL [6] architecture description language to Ada Ravenscar [7] source code[1]. Below we provide an introduction to both AADL and the Ravenscar Profile of Ada before moving on to an explanation of how the deterministic bridge exchangers are implemented in our tooling.

[1] Available at `http://aadl.enst.fr/arc/`

```
Procedure Get_Value(out Data)
  Invocation: static integer initialized to 0
  Data      : output for dataflow, sent to destination thread
  r         : integer corresponding to P_long/P_short
  begin
    Invocation ← Invocation + 1
    if Invocation = 1 then
      Front_Store ← Back_Store
    end
    if Invocation = r then
      Invocation ← 0
    end
    Data ← Front_Store
    return
  end
```

A. AADL

The Architecture Analysis and Design Language [6] is a new architecture description language designed specifically for real-time systems such as avionics and automotive control. It uses a component-centric model to define the system architecture. System descriptions in AADL consist of a set of components, each of which exposes well-defined interfaces. These interfaces are connected together to form a communication topology among the components. Component categories defined in the language include software (processes, threads, data, and subprogram) as well as hardware (processors, devices, buses and memories).

As AADL is an architecture description language—its main concern being the system architecture—it allows primarily the description of non-functional aspects of the components. These include, among others, the periods of threads, their deadlines, their stack sizes, their interface specifications etc. Functional aspects such as source code for software components is given separately.

Among the various different types of interfaces that can be put on components in AADL (called *features* in the language), are `data ports`. Two connected data ports on different components behave like a dataflow channel. A value written on the source side becomes visible on the destination side. Data ports also have associated *qualifiers*, which are AADL data components which give the data type of the flow implemented by these ports.

AADL components can also have *properties* assigned to them. Properties are name/value pairs that represent certain aspects of components. Among others, properties are used to give the periods and dispatch protocols for threads. Users can also define their own properties which are project and/or tool specific.

B. Ada Ravenscar

The Ada programming language provides rich tasking and inter-task communication constructs. It provides constructs within the language that are usually relegated to an operating system by other languages, e.g.: POSIX threads and mutexes used by the C programming language.

The Ada Ravenscar Profile [7] is a profile (restriction) of the Ada programming language's rich tasking constructs. This profile aims to ease the schedulability analysis of the system. Furthermore, through the judicious use of the priority ceiling protocol [8], it eliminates the posssibility of deadlock in the system. It also includes restrictions on the dynamic creation/destruction of tasks and memory allocation in order to make the system deterministic and suitable for high-integrity applications.

The Ada programming language provides *tasks* which implement the same functionality as POSIX threads. *Protected objects* are like C++ classes in that they expose an interface and contain private data. Furthermore, access to the procedures of protected objects are concurrency-safe. Thus, protected objects can be used to implement inter-task communication. In fact, the Ravenscar Profile stipulates that *only* protected objects be used to implement inter-task communication. Also stipulated by the profile is that all protected objects follow the priority ceiling protocol [8] in order to guard against deadlock and to avoid unbounded priority inversion.

As a consequence, a Ravenscar-compliant Ada runtime is guaranteed to be smaller and simpler than a complete Ada runtime. For our tests, we have used the Open Ravenscar Kernel [9], which is an open source kernel available from UPM.

C. AADL to Ravenscar Converter

The ARC tool is an open source Eclipse plugin that converts AADL system descriptions to Ada Ravenscar source code. In [5] we gave a mapping for translating the software components and entities of AADL—processes, threads, data, ports, connections etc.—into Ada Ravenscar source code. This allows us to automatically generate a *framework*[2] tailored to our system architecture. This framework contains *holes* for the functional code in the form of callback procedures of periodic and sporadic tasks.

In our original mapping, we proposed the transformation of AADL threads to Ada tasks, either periodic or sporadic, depending upon the thread's AADL properties. Data ports were transformed to protected objects we named *exchangers*. In essence, an exchanger is a protected object with two procedures `Get_Value` and `Set_Value` and an internal 1-place buffer of the same type as the data port it implements to accomodate the data. As an example, listing 1 gives the specification of an exchanger protected object generated automatically to implement a connection between two data ports of type `Integer`.

[2]A framework is the execution support for the system, such as tasks, protected objects for communication among tasks and stubs to access the generated communication infrastructure. For details on the code generation rules, refer to `http://aadl.enst.fr/arc/doc/`.

```
protected type Integer_Exchanger
               (Priority_P : System.Any_Priority) is
   procedure Set_Value (D : in Data_Type);
   procedure Get_Value (D : out Data_Type;
                        F : out Boolean);
   private
      pragma Priority (Priority_P);
      Data  : Integer;
      Fresh : Boolean := False;
end Integer_Exchanger;
```

Listing 1. Ada package specification of the `Integer_Exchanger`

This implementation is perfectly valid for applications where deterministic communication is not a requirement. E.g.: a moving map display where the display thread has a period of 500 ms and the data from the GPS is refreshed every 100 ms. A slight amount of non-determinism in this application would be neither noticeable nor critical.

D. Integration of DBX in Ada and ARC

We have implemented both types of DBX connectors—the stagger and stepper—as generic Ada packages that can be instantiated with the following information:

- The priority ceiling.
- The *step/stagger factor*, this is the value $r = P_{long}/P_{short}$.
- The data classifier (type) of the data ports that are to be implemented.

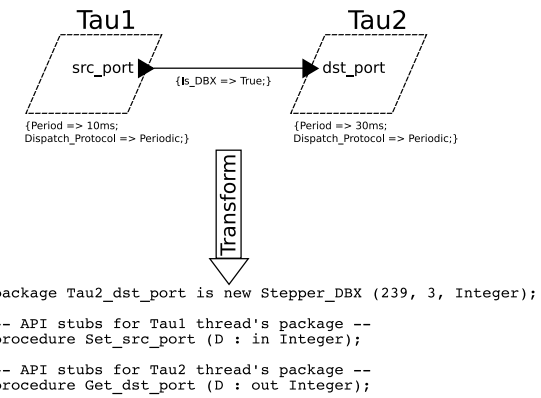
The specification of the generic package that implements a stepper DBX connector is given in listing 2. The generic package has only a protected object instance that serves as the exchanger. The specification for a stagger DBX connector is similar, the difference is in the `Set_Value` and `Get_Value` procedures' implementations.

```
generic
   Priority_P : System.Any_Priority;
   Factor_P : Integer;
   type Data_Type_P is private;
package Stepper_DBX is
   protected Stepper_DBX_Instance is
      procedure Set_Value (D : in Data_Type_P);
      procedure Get_Value (D : out Data_Type_P;
                           F : out Boolean);
   private
      pragma Priority (Priority_P);
      Back_Store : Data_Type_P;
      Front_Store : Data_Type_P;
      Invocation : Integer := 0;
      Fresh : Boolean := False;
   end Stepper_DBX_Instance;
end Stepper_DBX;
```

Listing 2. Ada package specification of the DBX connector

In our original work on ARC, we defined a property set `Ravenscar` in AADL to aid in code generation. To use AADL as a design and code generation vehicle for DBX connectors as well, we added a new property to the existing `Ravenscar` property set. The new property—of type boolean—is `Is_DBX`, and is applied to AADL connections and has a default value of `False`. If this property is set explicitly to `True` by the designer then instead of a normal exchanger, a stepper or stagger exchanger DBX is instantiated by ARC to correspond to the data ports' connection. The following conditions are tested before instantiating a DBX:

Fig. 5. An AADL system model with two threads and a deterministic data connection of type `Integer` being transformed to Ravenscar code through instantiation of a generic package.



1) The connection must be a data connection, not an event or event data connection and have the property association `Is_DBX => True`.
2) Both threads must have the property association `Dispatch_Protocol => Periodic`.
3) The period of one thread must be a perfect multiple of the period of the other, this can be verified by examination of both threads' `Period` property.

A stepper or a stagger DB exchanger is instantiated in case all three of the above conditions are met. Whether it is a stepper or a stagger DBX depends on whether the source thread has the shorter period or not. In addition to the instantiation of the DBX connector, stub procedures are generated in the functional packages of both threads which allow easy access to the DBX in question. On the side of the source thread is generated a stub procedure named `Set_<portName>`. On the side of the destination thread is generated a stub named `Get_<portName>`. These procedures simply call the `Set_Value` and `Get_Value` procedures of the actual exchanger. An example of such a transformation is given in figure 5.

This creates, in effect, a complete deterministic communication framework and generates an API to access these communication constructs. This aids greatly in implementing the entire system since the designer now needs to focus only on the functional (algorithmic) part of the system. Among the advantages of this approach are:

- Eliminates impact from system architecture to functional code. As all protocol information is embedded inside the connector thus the functional part need only interact with the tasking/communication framework through the provided API.
- Reduces programmer effort through the use of automatic code generation. This has an auxiliary consequence of reducing errors as well.
- Aids in the certification process for on-board software. This point will be explained in the next section when we present a verification of our connectors.

V. Verification

In this section we give details of the verifications we carried out on the stepper and stagger DBX connectors using LOTOS [10]. We use the CADP toolbox [11] in order to perform the verification. CADP stands for Construction and Analysis of Distributed Processes and is a toolbox that allows the compilation and analysis of LOTOS specifications. We use the LOTOS language in order to verify that our generated DBX connectors do not have any non-deterministic behavior and that the prescribed communication protocol is respected.

A. Overview of LOTOS

LOTOS stands for Language of Temporal Order Specifications. It is a process algebra inspired by CCS [12] and CSP [13] that uses the concepts of observable actions carried out by independant processes. Processes synchronize upon actions. Actions are taken over *gates* and may potentially involve an exchange of data between the gates of the synchronized processes.

The actions a LOTOS process can undertake are written in the form `a; b; c` which means the process engages in `a` followed by `b` and finally `c`. Actions of the sort `g?x` signify that an action is taken on *gate* `g` and offers `x` as the associated data. This action would need to be synchronized by a corresponding action of the sort `h!x` which signifies a gate `h` receiving data `x`.

B. Connector Specifications in LOTOS

In order to verify the correct behavior of our connectors, we implemented their specification in LOTOS. On running them in parallel with LOTOS blocks that behave as the threads on both sides of the dataflow, we were able to carry out a verification of correctness. It is possible to determine, given a certain communication configuration—i.e.: P_{short}, P_{long} and direction of dataflow—whether or not the automatically instantiated stepper or stagger exchanger will preserve determinism.

Figure 6 shows a graphical representation of the the specification of a stepper exchanger. From the `Start` state, it can engage in either a `GET_VALUE` or a `SET_VALUE` action. These correspond to a call to the exchanger's `Get_Value` and `Set_Value` procedures. The `SET_VALUE` action accepts incoming data, whereas `GET_VALUE` offers data available in the front store. It is obvious from the diagram that the `Inv` variable is incremented at each `SET_VALUE` action (and wraps around to 1 after `Factor` number of calls). The variable `Factor` represents $r = P_{long}/P_{short}$. The variables `Back_Store` and `Front_Store` are the same as those given in the pseudocode in section III and model the internal buffers. A similar diagrammatic representation for a stagger DBX is given in figure 7.

This block can be combined in parallel with blocks that specify the behaviors of the two threads in order to verify the correctness of the communication (figure 8). The two threads' behavior blocks should take into account the following hypotheses:

1) At the start of the mutual hyperperiod, it should be τ_s that should be launched first.

Fig. 6. The block defining behavior of a stepper exchanger.

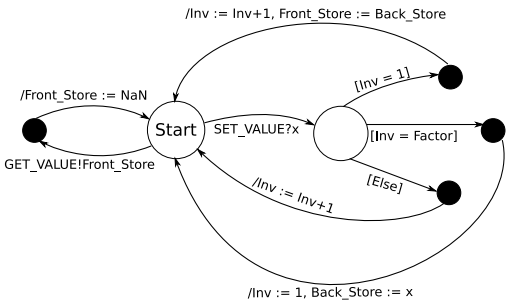
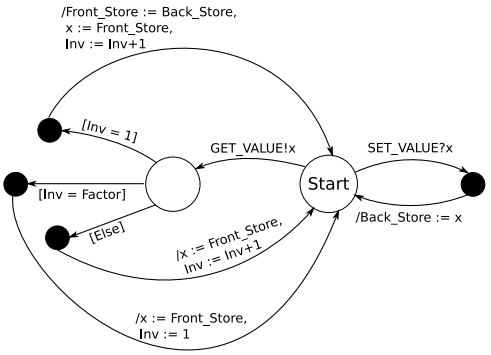


Fig. 7. The block defining behavior of a stagger exchanger.



2) There should be r (`Factor`) number of jobs of τ_s per hyperperiod, and 1 job of τ_l per hyperperiod.
3) At each job the source thread should engage in a `SET_VALUE` action and the destination thread in a `GET_VALUE` action.

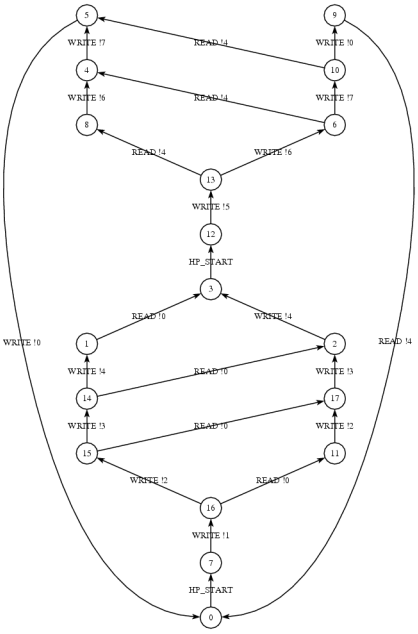
We can construct our source thread behavior block in such a fashion that the it writes a predefined pattern of data. If the destination thread does not receive the correct sequence of data, it raises a special `ERROR` action which if present, signals a non-deterministic run. The labelled transition system will also show a counter-example of the *run* that violated the rules.

We carried out verifications of both stepper and stagger DBX connectors for factors of 2, 3 and 4. We found no errors in all six scenarios. We used the CADP toolset to generate labelled transition systems from our LOTOS specifications. Figure 9 shows the labelled transition system generated for a stepper exchanger with a factor of 4 in parallel with two threads that use it to implement dataflow. The authors of [14] also use a process algebra to specify connectors, but they use it to detect deadlock in connector specifications, whereas we use it to verify determinism.

We also built test applications that we compiled onto the Open Ravenscar Kernel and ran on a simulator for the

Fig. 8. Three LOTOS blocks combined in parallel. Two are for threads and the third is for a DBX connector. The gates are shown on each block. One channel (connection between gates) is for the `Set_Value` interface and the other for `Get_Value`.

Fig. 9. Labelled transition system of a stepper DBX with a factor of 4 being used by two threads to implement dataflow. The absence of ERROR actions signifies that this system is not violating the determinism property.



ERC32 platform[3]. No errors were reported for stepper and stagger DBX connectors at factors of 2, 3 and 4. Source and destination thread periods were between 1 ms and 80 ms.

C. Application to Certification

DO-178B [15] is a widely accepted and applied standard for avionics software. It divides on-board software into one of five categories according to its safety requirements. These levels are given in table I. Level A software is the software of highest criticality, and it is at this level that all control software must be certified. The standard stipulates that Level A certified software must undergo Modified Condition/Decision Coverage (MCDC) testing. This code coverage method requires a test case to verify each condition that can affect the outcome of a decision. A compound conditional such as `if(a>0 && b<5)` would result in four MCDC test cases with $a \leq 0$, $a > 0$, $b < 5$ and $b \geq 5$.

Generating such test cases by hand is not feasible. Therefore, Level A certification usually requires instrumented compilers that identify and generate test cases. However, in case of moderate to complex software the number of test cases generated may be very large and thus testing becomes expensive. Our approach of verified exchangers for deterministic communication can reduce the cost of certification by providing verification artifacts in support of certification. This can potentially allow the certification authority to forego MCDC coverage of the exchangers as their behaviour has been formally verified and all conditions examined through the use of an exhaustive state-space search.

[3]The test applications, LOTOS specifications for the stepper and stagger DBX and different threads are available at `http://aadl.enst.fr/arc/doc/`. A detailed case study of deterministic dataflow control is also available at said URL in open source.

TABLE I
DO-178B/ED-12B SAFETY CRITICALITY LEVELS, COURTESY OF [17]

Software level	Failure condition	Outcome
Level A	Catastrophic	Death or injury
Level B	Hazardous/Severe-major	Injury
Level C	Major	Unsafe workload
Level D	Minor	Increased workload
Level E	No effect	None

Furthermore, the LOTOS code that was written by hand to verify the behaviour of the stepper and stagger DBX connectors may easily be generated automatically by ARC. It can traverse the entire system model, locate and identify DBX connections by their properties, and generate corresponding LOTOS specifications for the connectors and the threads which they are connected to. These automatically generated specificatioins can potentially replace the MCDC test cases for every DBX connector in the system, thereby significantly reducing certification cost. In fact, the UK defence software standard Def Stan 00-55 [16] promotes proof of correctness in the design rather than absence of errors. Research has shown that there is a correlation between both approaches [17]. Automatic generation of a process algebraic specification from AADL was also given in [18] where the authors use a process algebra to determine the schedulability of the system.

VI. CONCLUSION AND FUTURE WORK

We considered a problem in real-time control systems that is usually solved by synchronous methodologies and techniques. We proposed a light-weight and elegant solution to the problem for asynchronous systems. We integrated our proposed methodology into our existing tool.

We also provided verification of the correct behavior of our proposed solution. A path to verification of an arbitrary communication configuration using our approach is also given. We have also provided a justification for the use of this verification in lieu of testing for standard certification processes. Our methodology provides a robust and automatic manner for constructing connectors for co-located threads. This reduces programmer effort and the chance for bugs in the system.

Our ARC tool is based on the OSATE AADL parser [19]. The OSATE toolkit represents a parsed AADL tree in the form of an EMF model. This allows creation of completely model-driven tools. We also use an intermediate meta-model to represent the Ravenscar elements. Code generation is done by traversal of the Ravenscar meta-model. This will facilitate the writing of code generators for Ravenscar-compliant languages such as Ravenscar-Java [20].

Our future work will consist of a detailed case study involving the re-engineering of an existing reference avionics architecture to use the AADL/Ravenscar approach. Furthermore, a model traceability plugin is planned that will provide hypertext reporting between artifacts in AADL and the resulting generated source code. Such type of traceability documentation is very useful in certification processes.

Another interesting axis of research is the investigation into graceful degrade and/or fault tolerance of these connectors in case of overruns by tasks. We intend to investigate how to carry out a synergistic combination of the presented connectors together with the new temporal fault handling mechanisms introduced in Ada 2005.

REFERENCES

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[2] P. L. Butler and J. P. Jones, "Modular Control Architecture for Real-time Synchronous and Asynchronous Systems," in *Proc. SPIE Vol. 1964, p. 287-298, Applications of Artificial Intelligence 1993: Machine Vision and Robotics, Kim L. Boyer; Louise Stark; Eds.*, K. L. Boyer and L. Stark, Eds., Mar. 1993, pp. 287–298.

[3] L. Sha, M. H. Klein, and J. B. Goodenough, "Rate Monotonic Analysis for Real-Time Systems," *Computer*, vol. 26, no. 3, pp. 73–74, 1993.

[4] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*. New York, NY, USA: ACM Press, 2000, pp. 178–187.

[5] I. Hamid, B. Zalila, E. Najm, and J. Hugues, "A Generative Approach to Building a Framework for a Hard Real-Time System," in *31st Annual IEEE-NASA Goddard Software Engineering Workshop*, Baltimore, MD, 2007.

[6] SAE, *Architecture Analysis & Design Language (AS5506)*, September 2004, available at http://www.sae.org.

[7] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in high integrity systems," 2003.

[8] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[9] J. A. de la Puente, J. F. Ruiz, and J. Zamorano, "An Open Ravenscar Real-Time Kernel for GNAT," in *Ada-Europe '00: Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*. London, UK: Springer-Verlag, 2000, pp. 5–15.

[10] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Comput. Netw. ISDN Syst.*, vol. 14, no. 1, pp. 25–59, 1987.

[11] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis, "A Toolbox for the Verification of LOTOS Programs," in *ICSE '92: Proceedings of the 14th international conference on Software engineering*. New York, NY, USA: ACM Press, 1992, pp. 246–259.

[12] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.

[13] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[14] B. Spitznagel and D. Garlan, "A Compositional Approach for Constructing Connectors," in *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 148.

[15] RTCA and EUROCAE, *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, RTCS and EUROCAE, 1992.

[16] UK Ministry of Defence, *Requirements for Safety Related Software in Defence Equipment*, Def Stan 00-55.

[17] P. Parkinson and F. Gasperoni, "High-Integrity Systems Development for Integrated Modular Avionics Using VxWorks and GNAT," in *Ada-Europe '02: Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*. London, UK: Springer-Verlag, 2002, pp. 163–178.

[18] O. Sokolsky, I. Lee, and D. Clark, "Schedulability Analysis of AADL Models," in *IPDPS '06: 20th International Parallel and Distributed Processing Symposium*, April 2006.

[19] SEI, "Open Source AADL Tool Environment," http://la.sei.cmu.edu/aadl/currentsite/tool/osate.html, 2006.

[20] J. Kwon, A. Wellings, and S. King, "Ravenscar-Java: A High-Integrity Profile for Real-Time Java," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 5-6, pp. 681–713, 2005.

[21] M. Bordin and T. Vardanega, "Automated Model-Based Generation of Ravenscar-Compliant Source Code," in *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 59–67.