

Figure 2. Non-determinism in data-flow

riority 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 data-flow. 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 data 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 . However, in the next hyperperiod, τ_l finishes just before the 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—which sacrifices freshness of data for determinism—is widely used in industry. All data-flow values are taken from the last job of the previous mutual hyperperiod of both threads involved. This eliminates non-determinism due to scheduling decisions by the RTOS.

This is available in Simulink[®] as “rate transition” blocks. The Simulink[®] execution model collapses all periodic rates into a timer interrupt. This procedure interleaves the execution of the various functional blocks according to their periods, thus all rates must be integer multiples of the “base rate”. This creates an offline schedule and data transfers can be placed at well-defined temporal points.

We present a method to implement this type of data-flow in an RTOS environment with true multitasking including sporadic threads. The protocol and its properties are explained in Section 2. Our proposed solution to implement the protocol is given in Section 3. The details of implementation as well as the tool support developed are given in Section 4. The verifications carried out are detailed in section 5. Section 6 presents our conclusions and discussion of future work.

2 Deterministic Protocol and its Properties

In this section we will formulate the mathematical properties of the data-flow. However, first we need to explicitly state the assumptions and hypotheses that can be made upon the system and the consequences thereof:

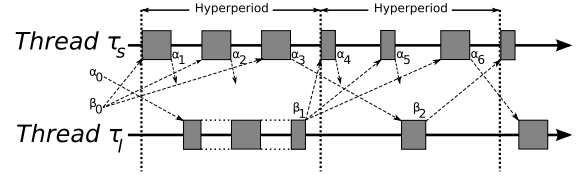


Figure 3. Protocol to ensure determinism

- The system is hard real-time. Periodic tasks are guaranteed to be dispatched at the start of each period (dispatching means being put in the ready state, not necessarily being given the processor)
- The release of tasks themselves is synchronous, i.e.: two periodic tasks with coincidental release times are dispatched concurrently (even though the higher priority task will execute first)
- Deadlines are met. Otherwise 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 data-flow and the destination thread reads from the data-flow at each job
- The scheduler is preemptive and priority based
- Priority assignment is rate monotonic [14], i.e.: τ_s has higher priority than τ_l . Thus, at the start of each mutual hyperperiod P_{long} , τ_s will be *executed* before τ_l , even though both were made ready simultaneously

2.1 Protocol

Given the above assumptions we can obtain a mathematical model for the deterministic data-flow protocol. For a data-flow from a high-frequency to low-frequency thread every r^{th} write is used. Similarly, for a dataflow from a low-frequency to high-frequency thread every write 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 data-flow α . For the example of Figure 3 ($P_{long} = 3 \times P_{short}$), the first few data-flows are:

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

We can generalize this into the following equation:

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

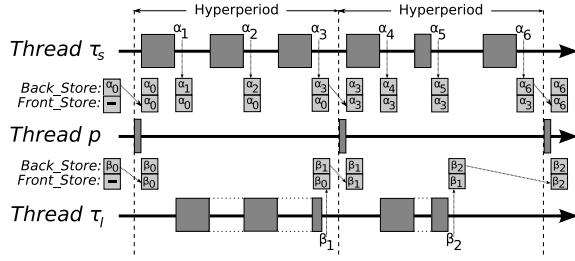


Figure 4. A suboptimal implementation

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 per hyperperiod for $r = 1$.

2.2 Suboptimal Implementation

A suboptimal—yet correct—implementation would be to introduce an extra thread to assure the protocol. The data-flow would be implemented via a double buffer. The source thread would write to the back buffer and the destination thread would read from the front buffer. A protocol thread with priority greater than both the source and destination threads and period 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 (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 rate monotonic assignment

3 Deterministic Bridge Exchangers

We propose a connector based approach for the implementation of the data-flow protocol. From [9]:

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 the clear separation between components and communication mechanisms, two orthogonal concerns of a system. Data-flow presents an ideal case where the components are the participating threads. By subsuming the functionality of the protocol into a connector 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 data-flow from a high-frequency to a low-frequency thread. We call the connector that implements a data-flow from a low-frequency to a high-frequency thread a *stagger deterministic bridge exchanger* (stagger DBX). Both types of connectors have an internal double buffer of the same type as the data-flow, called the *back store* and *front store*. They also provide concurrency-safe procedures `Set_Value/Get_Value`.

3.1 Stepper DBX

This connector implements a deterministic data-flow from a high-frequency thread τ_s to a low-frequency thread τ_l . τ_s computes a new value and calls `Set_Value` on the connector on each job, which is read by τ_l on each job via a call to the connector's `Get_Value` procedure. 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 data-flow protocol, a stepper DBX embeds buffer handling logic into `Set_Value`:

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 pseudocode for a stepper DBX `Set_Value` is shown as Algorithm 1. We call this connector a stepper exchanger since it *steps over* a certain number of inputs to the data-flow, only every r^{th} data value is copied into the back store.

Algorithm 1: `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  $\leftarrow$  Invocation + 1
  if Invocation = 1 then
    Front_Store  $\leftarrow$  Back_Store
  end
  if Invocation = r then
    Back_Store  $\leftarrow$  Data
    Invocation  $\leftarrow$  0
  end
  return
end

```

3.2 Stagger DBX

This connector implements a deterministic data-flow from a low-frequency thread τ_l to a high-frequency thread τ_s . τ_l computes a new value and calls `Set_Value` on the connector on each job, which is read by τ_s on each job via a call to the connector's `Get_Value` procedure. In each mutual hyperperiod P_{long} , there will be $r = P_{long}/P_{short}$ invocations of `Get_Value` and one invocation of `Set_Value`. Also, at the start of each P_{long} , it will be `Get_Value` which will be invoked first, as τ_s has a higher priority. Thus, a stagger DBX embeds buffer handling logic into `Get_Value`:

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 `Get_Value` returns the data in the front store
3. Every invocation of `Set_Value` copies the data provided by the source thread into the back store

The pseudocode for a stagger DBX `Set_Value` is shown as Algorithm 2. We call this connector a stagger exchanger because it *stagger*s on a data-flow value a certain number of times before getting a new one.

Algorithm 2: `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  $\leftarrow$  Invocation + 1
  if Invocation = 1 then
    Front_Store  $\leftarrow$  Back_Store
  end
  if Invocation = r then
    Invocation  $\leftarrow$  0
  end
  Data  $\leftarrow$  Front_Store
  return
end
```

4 Implementation and Tooling

We integrated the stepper and stagger DBX into ARC [7], our code generator that transforms system architectures specified in the AADL [13] architecture description language to Ada Ravenscar [2] source code¹. Below we provide an introduction to both AADL and the Ada Ravenscar before moving on to an explanation of how the deterministic bridge exchangers are implemented in our tooling.

¹Available at <http://aadl.enst.fr/arc/>

4.1 AADL

The Architecture Analysis and Design Language [13] is a new architecture description language targeted specifically to 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 subprograms) 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 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 are given separately.

Among the various types of interfaces that can be put on components in AADL, are data ports. Two connected data ports on different components behave like a data-flow channel. A value written on the source port becomes visible on the destination port. Data ports also have associated *qualifiers*, which are AADL data components that specify 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 may also define project and/or tool specific properties.

4.2 The Ravenscar Profile

The Ada language runtime provides constructs which are generally part of the operating system. These include *tasks* and *protected objects*. The Ravenscar Profile [2] is a restriction of the Ada runtime's tasking constructs, it aims to ease the schedulability analysis of the system. It also restricts dynamic creation/destruction of tasks and memory allocation in order to make the system deterministic and suitable for high-integrity applications.

Ada tasks implement the same functionality as POSIX threads. Protected objects are like C++ classes in that they expose an interface and contain private data. Furthermore, procedures of protected objects are concurrency-safe, thus they 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 (on co-located tasks). Also stipulated by the profile is that all protected objects follow the priority ceiling protocol [15] in order to guard against deadlock and to bound priority inver-

sion. As a consequence, a Ravenscar-compliant Ada runtime is smaller and simpler than a complete Ada runtime. We use the open source Open Ravenscar Kernel [4].

4.3 AADL to Ravenscar Converter

ARC is an open source Eclipse plugin that converts AADL system descriptions to Ada Ravenscar code. In [7] we gave a mapping for translating the software components and entities of AADL—processes, threads, data, ports, connections etc.—to Ada Ravenscar. ARC generates a framework² tailored to the system architecture that has *holes* for functional code in the form of callback procedures of tasks.

In our original mapping, we proposed the transformation of AADL threads to Ada tasks. A data port is transformed to protected objects we named *exchangers*, 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 accommodate the data. Listing 1 gives the specification of an exchanger generated automatically to implement a connection between two data ports of type *Integer*.

```
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. Integer_Exchanger spec

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.

4.4 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 parameters:

- The ceiling priority for its protected object
- The *step/stagger factor* $r = P_{long}/P_{short}$
- The data classifier (type) of the data ports involved

The specification of the generic package that implements a stepper DBX connector is given in Listing 2. The generic

²See <http://aadl.enst.fr/arc/doc/>

package only has a protected object instance that serves as the exchanger. The specification for a stagger DBX connector is similar.

```
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. Stepper DBX spec

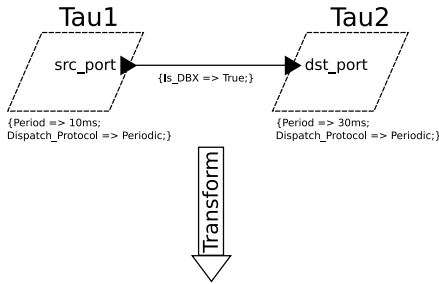
In our original work on ARC, we defined a property set called *Ravenscar* in AADL to aid in code generation. Now we add a new boolean property *Is_DBX* (applied to AADL connections) to this property set. If this property is *True* then instead of a normal exchanger, a stepper or stagger exchanger DBX is instantiated to correspond to the data ports' connection. The following conditions are tested before instantiating a DBX:

1. The connection must be a data connection
2. Both threads must be periodic
3. The period of one thread must be a perfect multiple of the period of the other

Whether a stepper or a stagger DBX is instantiated 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. On the side of the source thread a stub procedure named *Set_<portName>* is generated. On the side of the destination thread a stub named *Get_<portName>* is generated. These procedures simply call the *Set_Value* and *Get_Value* procedures of the actual exchanger. One such transformation is shown in Figure 5.

This creates a complete deterministic communication framework and also generates an API to access it. This aids the designer since he now has to focus on only the functional part. Among the advantages of this approach are:

- Eliminates impact from system architecture to functional code. All protocol information is embedded in the connector. The functional part need only interact with the generated API



```

package Tau2_dst_port is new Stepper_DBX (239, 3, Integer);
-- API stubs for Tau1 thread's package --
procedure Set_src_port (D : in Integer);
-- API stubs for Tau2 thread's package --
procedure Get_dst_port (D : out Integer);

```

Figure 5. Code generation from AADL with a stepper DBX between two tasks

- Reduces programmer effort and errors through the use of automatic code generation
- Aids in the certification process for onboard software. This point will be explained in the next section

5 Verification

In this section we give details of the verifications carried out on the stepper and stagger DBX connectors with LOTOS [1] using the CADP toolbox [5]. We use LOTOS to verify that our generated DBX connectors do not have any non-deterministic behavior and that the prescribed communication protocol is respected.

5.1 Overview of LOTOS

LOTOS (Language of Temporal Order Specifications) is a process algebraic formal description technique inspired by CCS [10] and CSP [8]. It uses the concept of observable actions carried out by independent 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 with a corresponding action of the sort $h!x$ which signifies an action on *gate* h receiving data x .

5.2 Connector Specifications in LOTOS

In order to verify the correct behavior of our connectors, we implemented their specification in LOTOS. On run-

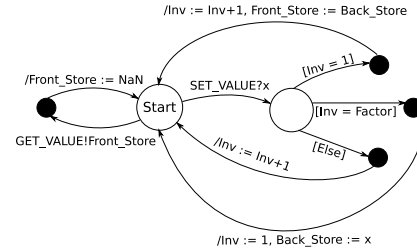


Figure 6. Stepper exchanger LOTOS spec

ning them in parallel with LOTOS blocks that behave as the threads on both sides of the data-flow, 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 data-flow—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 3 and model the internal buffers. A similar diagrammatic representation for a stagger DBX is given in Figure 7.

These blocks can be combined in parallel with blocks that specify the behaviors of the two threads in order to verify the correctness of the communication. The two threads’ behavior blocks should take into account the following:

1. At the start of the hyperperiod, τ_s is launched first
2. There should be r (*Factor*) jobs of τ_s per hyperperiod, and 1 job of τ_l per hyperperiod
3. At each job the source and destination threads should engage in a *SET_VALUE/GET_VALUE* action respectively

We construct the source thread behavior block so that it writes a predefined pattern of data (modulo a certain number of hyperperiods to avoid a state space explosion in LOTOS). If the destination thread does not receive the expected sequence of data, it engages in a special *ERROR* action. In this case, the offending run will be visible in the labelled transition system.

We carried out verifications of both stepper and stagger DBX connectors for factors of 2, 3 and 4. We found no

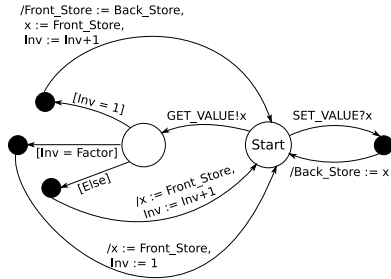


Figure 7. Stagger exchanger LOTOS spec

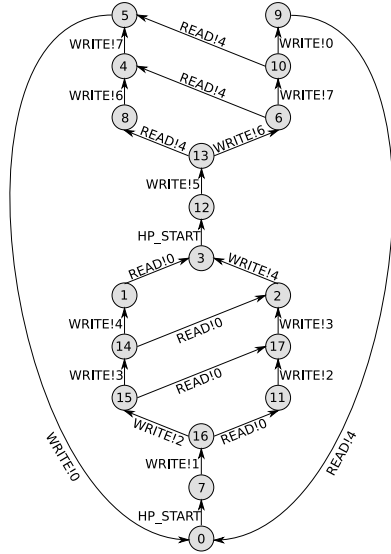


Figure 8. Generated LTS

errors in all six scenarios. We used the CADP toolset to generate labelled transition systems from our LOTOS specifications. Figure 8 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 data-flow (state 0 is the start state). The authors of [17] 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 ERC32 platform³. 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.

5.3 Application to Certification

DO-178B [12] is a widely accepted and applied standard for avionics software. It divides onboard software into

³Test applications, LOTOS specifications for the stepper and stagger DBX and threads are available at <http://aad1.enst.fr/arc/doc/>

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

Table 1. DO-178B Safety Criticality Levels [11]

one of five categories according to its safety requirements. These levels are given in Table 1. Level A software is of highest criticality, and it is at this level that all control software must be certified. The standard stipulates that Level A certified software 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 condition 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. Level A certification thus requires instrumented compilers to identify and generate test cases. However, for complex software the number of test cases generated may be very large and thus testing becomes expensive. Verified exchangers for deterministic communication can reduce this cost 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/decisions examined thanks to the exhaustive state-space search. In fact, the UK defence software standard Def Stan 00-55 [18] promotes proof of correctness in the design rather than absence of errors. Research has shown that there is a correlation between both approaches [11].

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 they are connected to. Translations of AADL to process algebraic specifications have also been used to determine schedulability of the system [16].

6 Conclusion and Future Work

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

We provided verification of the correct behavior of our proposed solution. A path to verification of an arbitrary

communication configuration using our approach is given as well. 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.

The DBX connectors and their implementation are a necessary step towards the acceptance of the AADL language as a viable tool for control system design. With these connectors, code generated from AADL models to run on Ravenscar-compliant kernels is faithful to the semantics of data ports as given in the standard. They also alleviate the problem of data inconsistency in case a thread has more than one incoming data port. Such a thread may be pre-empted by the source thread after it has read one port but before it has read the others. The source thread may then write a new set of data on the ports causing the thread in question to read the new data for the unread ports when it resumes. However, this is irrelevant with DBX as the source thread writes in the *back store* while the destination thread continues its calculations with values read from the *front store*.

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.

There is also the possibility of extending the DBX connectors by replacing the internal front and back stores with circular queues. In such a case, operations that copy the back store to the front store would simply “promote” the queue. This would ease in transcribing complex difference equations to AADL models and generating code for them. The designer would not have to explicitly implement buffers in functional code to access values sent over data ports in previous jobs.

Another interesting path 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] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.
- [2] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems, 2003.
- [3] P. L. Butler and J. P. Jones. Modular Control Architecture for Real-time Synchronous and Asynchronous Systems. In K. L. Boyer and L. Stark, editors, *Proc. SPIE Vol. 1964*, p. 287-298, *Applications of Artificial Intelligence 1993: Machine Vision and Robotics*, Kim L. Boyer; Louise Stark; Eds., pages 287–298, Mar. 1993.
- [4] 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*, pages 5–15, London, UK, 2000. Springer-Verlag.
- [5] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590, pages 158–163, jul 2007.
- [6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [7] 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.
- [8] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.
- [9] 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*, pages 178–187, New York, NY, USA, 2000. ACM Press.
- [10] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
- [11] 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*, pages 163–178, London, UK, 2002. Springer-Verlag.
- [12] RTCA and EUROCAE. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. RTCS and EUROCAE, 1992.
- [13] SAE. *Architecture Analysis & Design Language (AS5506)*, September 2004. available at <http://www.sae.org>.
- [14] L. Sha, M. H. Klein, and J. B. Goodenough. Rate Monotonic Analysis for Real-Time Systems. *Computer*, 26(3):73–74, 1993.
- [15] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [16] O. Sokolsky, I. Lee, and D. Clark. Schedulability Analysis of AADL Models. In *IPDPS '06: 20th International Parallel and Distributed Processing Symposium*, April 2006.
- [17] 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)*, page 148, Washington, DC, USA, 2001. IEEE Computer Society.
- [18] UK Ministry of Defence. *Requirements for Safety Related Software in Defence Equipment*. Def Stan 00-55.