# Shifting Temporal and Communicational Aspects into Design Phase via AADL and RTSJ

Thomas Driessen
Software Methodologies for Distributed Systems
University of Augsburg
Augsburg, Germany
Email: thomas.driessen@ds-lab.org

Bernhard Bauer
Software Methodologies for Distributed Systems
University of Augsburg
Augsburg, Germany
Email: bernhard.bauer@ds-lab.org

***Abstract*—By now, Model-Driven Development is a well-known approach in many domains. By (re)using standardized domain-specific models, productivity is increased and common errors are simultanously avoided. The Architecture Analysis and Design Language is a domain-specific modeling language for embedded, real-time and safety-critical systems. In our approach we utilize this modeling language, with its well-defined semantics, as source language for a mapping into real-time Java. The chosen subset of model elements enables system designers to create a system and subsequently generate a code framework that complies to the model in terms of structure, timing and communicational restrictions. In order to demonstrate the benefits of our approach, we model and generate the code framework for an existing autopilot and compare our results with the original software.**

## I. Introduction

It is common knowledge in Software Engineering that an early development error detection lowers the costs of its correction. This is especially the case if the system under development is an embedded or safety-critical one, whereby not only a system's software, but also its corresponding documentation or hardware is affected by changes.

In this context, "Model-Driven Development" (MDD) aims at shifting most aspects of a system's software implementation into earlier phases of the development, e.g., software design or system design. In this work, we thus concentrate on shifting the timing and inter-component communication aspects of a system's software from the implementation phase to the system design phase of a project.

Our approach uses the "Architecture Analysis and Design Language" (AADL) which offers – among other things – standardized semantics for timing and inter-component communication aspects of software components. In our approach, we utilize these semantics to define a mapping between the AADL and the "Real-Time Specification for Java" (RTSJ). RTSJ is an extension of "Java Standard Edition" (Java SE) for hard and soft real-time applications. By an implementation of this mapping, we generate AADL semantic-compliant RTSJ code which preserves the timing behavior and inter-component communication defined in an AADL model. Thus, a system designer is capable of designing and performing analyses regarding communication and timing almost completely during design phase, while resting assured that the implementation will reflect made design choices. Simultaneously, programmers are relieved of the monotonic and repetitive task of writing communication- and timing-related code.

This work is structured as follows: Section II introduces the chosen modeling language (AADL) and target language (RTSJ). Section III shows the chosen mappings for structural, temporal and communicational aspects. A use case for our approach is illustrated in Section IV, where an autopilot for a virtual quadrocopter is modeled and subsequently generated. Section V relates our approach to existing work. Finally, Section VI highlights the added value of our work.

## II. Basics

### A. AADL

AADL is a modeling language specifically designed for *"[...] the specification, analysis, automated integration and code generation of real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems"* [1]. Although the field of modeling languages is very crowded in this research area, AADL excels by defining an official, standardized semantics for all its model elements and properties. This makes it an ideal candidate for model-to-model or model-to-text transformations, as the semantics to be fulfilled by the target language are already predefined by the source language. The basic building blocks of AADL are:

- *Type declarations* which define the interface of a component that is visible to other components in terms of
- *features*, e.g., ports or access features that describe what kind of data is exchanged and how this exchange happens.
- *Implementation declarations* realize one *type declaration* and add information about the inner composition of the component in terms of subcomponents and their
- *connections* which can be used to describe access rights, delays or similar characteristics.

AADL provides predefined properties that are used to attach additional information to each of the model elements, e.g., for analysis or altering the behavior of a component.
Each *type declaration* and its corresponding *implementation declaration* can have a specific type, like *system*, *process* or *thread*, each with its own predefined semantic. For our approach, we utilized a subset that namely encompasses *threads*,

*data*, *data ports* and *port connections* with a subset of their possible properties.

### B. AADL-Subset

*1) Threads: Thread*s are the core concept in AADL for describing running software. *"A thread represents an execution path through code that can execute concurrently with other threads." [2]* The behavior of threads is standardized by AADL in terms of a timed automaton that defines different states which are entered through lifecycle events like *dispatch*, *start*, *completion* and *deadline*. A logical abstraction of this automaton is depicted in Figure 1. In this work we simplify the semantic of AADL in order to conform to the possibilities of RTSJ.

First, scheduler-specific events are not taken into account by this work, e.g., *dispatch* and *start* are considered the same, as the time of *dispatch* is not accessible in RTSJ by user code. Events that usually lead to a switch from *Active* state to *Executing* state or vice versa are summarized as a start event in our work.

Second, scheduler-specific states like *Awaiting Dispatch* and *Active* can not be represented in code, as a user in RTSJ is only aware of the *Executing* or *Recovery* state. In Figure 1, the additional states are merely depicted for conceptual completeness.

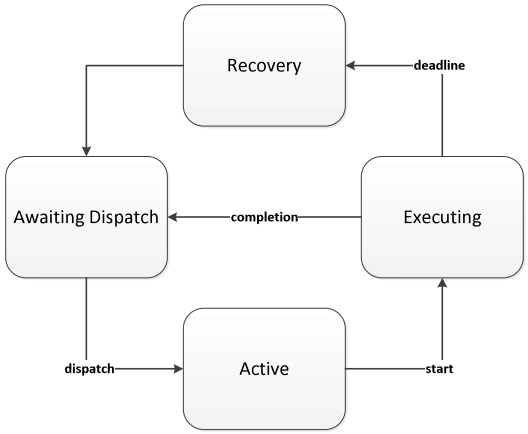


Fig. 1. Thread States

The behavior of a *thread* can be modified by associating properties with it, e.g., depending on the value of the property *Dispatch_Protocol* a *thread* is either considered as a *periodic*, *aperiodic*, *sporadic*, *background*, *timed* or *hybrid thread*, each with its own defined semantic meaning. In this work we only consider periodic *thread*s. Periodic *thread*s are usually started at the same time as the application and are running at a specified period. Therefore, they need a *Period* property which determines the *thread*'s amount of time between *dispatch* and *deadline*.

*2) Data Ports: Data port*s are specialized *feature*s that allow components to exchange data. They can be onedirectional (in, out) or bidirectional (in/out). Usually, the type of data that can be received/sent is given by a *classifier* which can be a primitive type from the *Base_Types* declared by AADL or a custom type specified by a *data type declaration* or *data implementation declaration*. AADL states that *data port*s can only receive/send one data element at a given point in time. Unlike other ports, for instance *event port*s that can buffer more than one event, *data port*s always store only one data element which is overwritten as soon as a new one arrives. In Listing 1, the *thread type declaration* `threadA` defines two *data port*s, `dataIn` and `dataOut`, with a specific offset to describe at what point in time a data element is received/sent at the respective *data port*. Usually, this offset (*Input_Time*/*Output_Time*) is given in relation to a lifecycle event (*IO_Reference_Time*) of the *thread* to which this *data port* belongs. Although negative offsets are allowed by AADL, we confine ourselves to positive ones.

```
thread threadA
  features
    dataIn : in data port Base_Types::Boolean
      {Input_Time =>
        ([Time => Start;
        Offset => 10ms .. 20ms;]);};
    dataOut : out data port Base_Types::Boolean
      {Output_Time =>
        ([Time => Completion;
        Offset => 10ms .. 20ms;]);};
end threadA;
```

Listing 1. Data ports with offsets

*3) Port Connections: Port connection*s are a more specific subtype of *connection*s which are only allowed between *port*s. Other possible *connection* types are *access connection*s, *parameter connection*s or the more general *feature connection*s or *feature group connection*s. As we restrict our approach to purely port-based communication, it is sufficient to explore *port connection*s and their relevant properties in detail. Listing 2 shows a *port connection* between two subcomponents with the classifier `threadA` which was defined in Listing 1. *Port connection*s between two *thread*s that are periodic and harmonic (same logical start time and period) can have three different values for their *Timing* property: *immediate*, *delayed* or *sampled*. Each timing has effects on the *Input_Time* and *Output_Time* of the connected *port*s. The semantic of each *Timing* is explained in detail in Subsection III-B. The default value for *Timing* is *sampled*.

```
process implementation processA.impl
  subcomponents
    sender : threadA;
    receiver : threadA;
  connections
    con1 : port sender.dataOut
    -> receiver.dataIn {Timing => Immediate};
end processA.impl;
```

Listing 2. Immediate port connection

*4) Data:* The *data* component is used to model user-specific data. *Data* can be used in the same way as classes or structs in common programming languages. In our approach we use *data* merely for modeling data types that can be used as classifiers for *data port*s. Other possible usage scenarios like shared data or local data are not part of this work. Listing

3 shows the usage of the user defined *data type declaration* `dataA` as classifier for the *in data port* `dataIn` of the *thread type declaration* `threadA`.

```
thread threadA
  features
    dataIn : in data port dataA;
end threadA;

data dataA
end dataA
```

Listing 3. Data as classifier for ports

### C. RTSJ

RTSJ was introduced in January 2002 as the first Java Specification Request (JSR-1) ever launched by the Java Community Process [3]. The specification is aiming for making Java SE usable in soft and even hard real-time systems which was previously impossible.

Real-time systems pose additional requirements on a system in terms of timeliness. A system not only has to deliver the right answer, but has to do this within a given amount of time. Otherwise, either the quality of the results is decreasing (soft real-time) or the whole system is rendered useless (hard real-time). For applications written in Java SE, those timing requirements were impossible to fulfill because of several language specific restrictions.

First, Java SE always comes with its built-in garbage collection that deallocates memory whenever it seems reasonable. As this deallocation can happen anytime, one cannot make exact predictions about the runtime behavior of a specific part of the code, as the garbage collector always could stop execution for deallocating memory.

Second, Java makes use of Just-in-Time compilation which recompiles code during runtime and is, like the garbage collector, unpredictable in terms of timing.

Other obstacles are a non-priority based scheduling of threads or the missing possibility of hardware access. RTSJ tries to bypass those obstacles by offering a new library for real-time compliant Java programming. RTSJ contains several classes that enable writing real-time capable software. The parts of the library that are of interest for our approach are described below.

*1) "Real-Time Thread" (RTT): Thread*s are the core part of a running software in RTSJ. RTSJ differs between two types of *thread*s, normal *thread*s and RTTs. Normal *thread*s are treated by the "Real-Time Java Virtual Machine" (RT-JVM) like a usual "Java Virtual Machine" (JVM) would do, whereas RTTs have one of at least 27 priorities that are taken into account by the real-time scheduler of the RT-JVM. A RTT is always considered more important than a normal *thread* and a RTT with a higher priority is always more important than one with a lower priority. Occasionally, this leads to preemption of *thread*s with lower importance if a *thread* with higher importance needs compute time. Preemption means the act of saving and stopping one *thread* in favor of a more important one and reestablishing the less important one once the more important *thread* has finished its work. This type of scheduling, based on the priorities given for each RTT, enables an exact prediction of the whole program in terms of timing.

*2) Timer and EventHandler:* A more abstract, but easier to use, concept compared to *thread*s are *Timer*s and *AsyncEventHandler*. RTTs are considered a rather low-level mechanism from a programmer's point of view. Thus, RTSJ provides two types of *Timer*s, *OneShotTimer* and *PeriodicTimer* that can be used to implement behavior that is either event-triggered or on a periodic base. A *Timer* must have either an *AsyncEventHandler* or a *BoundAsyncEventHandler* that encapsulates the logic to be executed once the handler is triggered. Both handler types are bound to a RTT in the background by the RT-JVM. Several *AsyncEventHandler* can share one RTT, but a *BoundAsyncEventHandler* is always bound to its own, unique RTT. If an *AsyncEventHandler* performs a blocking operation, for instance is waiting for another handler to finish its work, this might lead to a deadlock if both handlers are bound to the same RTT in the background. A solution for this problem is to always use *BoundAsyncEventHandler*s for blocking operations, as they are guaranteed to not share their RTT with other handlers.

*3) Real-Time Garbage Collection:* Although RTTs and the priority-based, preemptive scheduling of *thread*s allow predictions about the software's timing on a logical level, the real behavior of this software would still be unpredictable because of the garbage collector. Therefore, real-time garbage collectors are introduced by most of the RTSJ implementations in order to enable the software to run in a predictable way. Thus, programmers are still able to use the comfort of automatic memory management – and also prevent errors arising from this area – and still are able to write predictable, real-time capable software in Java.

*4) RTSJ Implementation:* The original specification is a work still in progress and is currently carried on by JSR-282 [4], specifying RTSJ 2.0 under the lead of aicas GmbH. The only RT-JVM supporting the newly developed features in RTSJ 2.0 is the JamaicaVM [5] from aicas GmbH, which is the implementation we are targeting.

## III. Mapping

In this section we present the three main parts of our mapping. First, we will show which structural characteristics of AADL are mapped onto the target language. Second, we present the timing-related features that can be automatically transformed into semantically appropriate RTSJ code. Third, we show how inter-component communication defined in AADL is transformed into inter-thread communication mechanisms in RTSJ.

### A. Structural Mapping

The structural mapping of our approach covers the AADL language constructs *type declaration*, *implementation declaration*, *subcomponent*s and the relationships *extends*, *realizes* and *refined to* that enable a user to structure components and refine them incrementally.

*1) Type and Implementation Declarations: type declarations* in AADL are representing interfaces in form of *features* that are provided by a realizing *implementation declaration*. In Java, the concept of an interface already exists. An interface defines method signatures which have to be realized by classes that implement the interface. In order to reuse this interface concept to reflect the *type declaration* concept in AADL, we translate each *feature* defined by a *type declaration* into corresponding method signatures. In our approach, we only regard *data port*s which can hold one data element at a time and therefore can be translated into simple `in` and `out` methods. This mapping is shown in Listing 4 and 5, where the *in/out data port* `dataIO` is translated into its corresponding `in` and `out` methods, each with the declared classifier `A` as parameter type.

```
process processA
  features
    dataIO : in out data port A;
end processA

data A
end A
```

Listing 4. Types in AADL

```
public interface ProcessA{
  void inDataIO(A data);
  void outDataIO(A data);
}
public interface A{}
```

Listing 5. Types as interfaces in Java

*Implementation declaration*s can *realize* exactly one *type declaration* and *extend* exactly one other *implementation declaration*. Multiple inheritance is forbidden by the standard. The extended *implementation declaration* has to *realize* the same *type declaration* as the extending one. One *type declaration* can have several *implementation declaration*s, each inheriting the *feature*s defined by the *type declaration*. Additionally, *implementation declaration*s can have *subcomponent*s which specify the inner composition of a component. When transferred to Java, *implementation declaration*s resemble abstract classes which have member variables that hold references to their subcomponents.

```
process implementation processA.impl
  subcomponents
    worker1 : thread threadA.impl;
    worker2 : thread threadA.impl;
end processA

thread threadA
  features
    dataIO : in out data port A;
end threadA

thread implementation threadA.impl
end threadA.impl
```

Listing 6. Implementations in AADL

Listing 6 depicts a *process implementation* `processA` of the *process type* defined in Listing 4 which inherits the *in/out data port* and adds two *thread* subcomponents, `worker1` and `worker2`. This *implementation declaration* is translated into a Java class as shown in Listing 7. The class `ProcessAImpl` implements the interface `ProcessA`, inherits the methods defined by `ProcessA` and implements them. The implementation of `in/out` methods is explained in detail in III-C3. Likewise, each subcomponent is represented by its own member variable, i.e., `worker1`, `worker2` and uses the declared *classifier* as type, i.e. `ThreadAImpl`. The member variables are initialized within the constructor, as all classes and interfaces can be seen as blueprints which have to be assembled by an outside entity.

```
public abstract class ProcessAImpl implements ProcessA{
  ThreadAImpl worker1;
  ThreadAImpl worker2;

  public ProcessAImpl(ThreadAImpl worker1,
        ThreadAImpl worker2){
    this.worker1 = worker1;
    this.worker2 = worker2;
  }

  @Override
  public void inDataIO(A data){
    ...
  }

  @Override
  public void outDataIO(A data){
    ...
  }
}

public interface ThreadA{
  void inDataIO(A data);
  void outDataIO(A data);
}

public class ThreadAImpl implements ThreadA{...}
```

Listing 7. Implementations as classes in Java

*2) Hierarchies and Refinements: Type declaration*s and *implementation declaration*s can be used to create an inheritance hierarchy of components by letting one component extending another or by an *implementation declaration* realizing a *type declaration*. The *extends* relation of AADL only allows single inheritance and demands the extending component to be of the same type, e.g., *system*, *process*, etc., as the one that is extended. Therefore, we can simply reuse the `extends` keyword from Java in order to map this relation. *Type declaration*s inherit all features from their parent *type declaration* which is mapped in Java by the child interface inheriting all method signatures from the parent interface. The same concept can be transferred to *implementation declaration*s that are classes instead of interfaces in Java. An *implementation declaration* extending another *implementation declaration*, inherits all features and subcomponents from its parent. In Java, a subclass extending a superclass inherits all non-private member variables, representing subcomponents and all methods representing features.

In case of an *implementation declaration* realizing a *type declaration*, we will use the `implements` keyword in Java to map the semantic meaning. An realizing *implementation declaration* inherits all features defined by the *type declaration*. The same semantic meaning is given by a Java class that

is implementing an interface, whereby all methods declared in the interface are inherited and implemented by the class.

Problems arise when it comes down to the refine mechanism in AADL. The refinement of a component encompasses, among other things, the possibility to refine classifiers of *port*s by an extending *type declaration*. In order to reflect the classifier refinement of a *data port* in an extending and refining *type declaration*, we have to change the signature of its corresponding method in Java. Listing 8 shows a simple refinement of an *in data port*, where the classifier gets specialized by the extending *type declaration*. In Java, this would lead to an interface definition as shown in Listing 9. Declared as depicted, the overridden `in` method for `dataIO` is an invalid method signature. In Java, an overriding method must have the same signature, composed by method name and parameter types, as declared in the super type. As the overriding method changes the parameter types, it is not valid.

In order to restrict the possible types for refinement, AADL defines the property *Classifier_Substitution_Rule* which can be associated with a port. Currently, AADL defines three different values for this property, *Classifier_Match*, *Type_Extension* and *Signature_Match*. We only consider the former two. *Classifier_Match* – the default value – enforces the refined classifier to be exactly the same as the classifier in the extended component. *Type_Extension* allows the refined classifier to be a subtype of the one used by the extended component.

```
thread threadA
  features
    dataIO : in data port A
    {Classifier_Substitution_Rule=>Type_Extension};
end threadA;

thread threadB extends threadA
  features
    dataIO : refined to in data port B;
end threadB;

data A
end A

data B extends A
end B
```

Listing 8. Refinement of features in AADL

```
public interface ThreadA{
  void inDataIO(A data);
}

public interface ThreadB extends ThreadA{
  @Override
  void setDataIO(B data); //Compiler Error
}

public interface A{}
public interface B extends A{}
```

Listing 9. Erroneous refinement of features in Java

Addressing the previously mentioned problem of invalid method overriding, we decided to use Java's default implementation mechanism [6] to prevent an illegal use of the wrong method for a refined classifier of a *data port*. As depicted in Listing 10, the refined `in` method is defined in interface `ThreadB` and the now illegal `in` method, that is inherited from `ThreadA`, is per default throwing an `UnsupportedOperationException`. This way we conform to the semantic meaning of the AADL model, while only making minimal changes to the inheritance mechanisms in Java.

```
public interface ThreadA{
  void inDataIO(A data);
}

public interface ThreadB extends ThreadA{
  @Override
  default void inDataIO(A data) {
    throw new UnsupportedOperationException();
  }

  void inDataIO(B data);
}
```

Listing 10. Valid refinement of features in Java

### B. Timing concerned Mapping

In this section, we explain how the different semantics for timing of *thread*s, *data port*s and their *connection*s are mapped from AADL to RTSJ.

As already described in Section II-B1, *thread*s are the core concept for running software in AADL. As such they have a semantic defined by AADL that describes their timing behavior in detail. We restrict the *Dispatch_Protocol* of *thread*s in this work to *periodic*, so we will only explain the behavior of periodic *thread*s and properties altering it.

*1) Period, Deadline and Priority:* Periodic *thread*s must have a period, i.e., an interval at which they are executing code. The period can be given by the property *Period* in form of a number and a time unit, e.g., 200 ms or 3 sec. *Period* can be directly translated into RTSJ by using a `PeriodicTimer` for which a period can be given. A `Timer` is an event trigger and meant to be used in conjunction with the aforementioned `AsyncEventHandler`s, so the code to be executed is encapsulated in the `handleAsyncEvent()` method of the handler.

Another timing-related AADL property is the *Compute_Deadline* of a *thread* which states until when the computation has to be done at the latest. In RTSJ, an `AsyncEventHandler` can have so called `ReleaseParameters` which define a deadline and a `deadlineMissHandler` that is called in the case of a not fulfilled deadline.

Although not actually being a timing-related property, the *Priority* is nevertheless essential for every priority-based scheduler. It can be given for each *thread* and can be represented by `PriorityParameters` in RTSJ which again are associated with `AsyncEventHandlers`. Listing 11 and 12 depicts the mapping between a *thread implementation* defined in AADL, with the three mentioned properties associated, and a class in RTSJ representing an instance of this *thread implementation*.

```
thread implementation threadA.impl
  properties
    Dispatch_Protocol => periodic;
```

```
    Period => 200ms;
    Priority => 5;
    Compute_Deadline => 100ms;
end thread.impl;
```

Listing 11. Timing in AADL

```
public class ThreadAInstance extends ThreadImplA{
  private AsyncEventHandler handler =
          new InnerAsyncEventHandler();
  private Timer timer = new PeriodicTimer
          (null,new RelativeTime(200,0),handler);

  class InnerAsyncEventHandler
          extends AsyncEventHandler{

    public InnerAsyncEventHandler(){
      setDeamon(false);
      setSchedulingParameters
        (new PriorityParameters(5));
      ReleaseParameters rps =
        timer.createReleaseParameters();
      rps.setDeadline(new RelativeTime(100,0));
      setReleaseParameters(rps);
    }

    public void handleAsyncEvent(){
      //Logic
    }
  }
}
```

Listing 12. Timing in RTSJ

By default, all `AsyncEventHandlers` are treated as Daemons, i.e., background tasks that are only executed as long as the main thread in Java is running. In order to make an `AsyncEventHandler` a foreground task (which is executed independently from the main thread) we have to explicitly call `setDeamon(false)`. The `SchedulingParameters` of `ThreadAInstance`'s `InnerAsyncEventHandler` are used to reflect the *Priority* of `threadA.impl`. The *Dispatch_Protocol* and *Period* are directly translated into a `PeriodicTimer` with its period set to 200 ms. The handler for this timer is `InnerAsyncEventHandler` as it extends `AsyncEventHandler`. Finally, the `ReleaseParameters` are used to map the *Compute_Deadline*.

*2) Input_Time and Output_Time at Data Ports:* Timing is not only important in the context of *thread*s and their execution time, but also for *port*s and the time when they are receiving or sending data. Based on *Input_Time* and *Output_Time* property values given for a *port*, there are basically two possibilities for *data port*s in AADL.

First, a *data port* receives/sends data at a given lifecycle event of the *thread* it belongs to, i.e., *dispatch*, *start*, *completion* or *deadline*, without any additional offset. In this case, the receiving/sending can be done by the *thread* itself.

Second, the *data port* defines its timing with a given offset in relation to one of the above mentioned lifecycle events. In this work we only consider a positive offset as sensible. In this second case, the *thread* is no longer able to do the receiving/sending on its own, but has to start a parallel task at the given lifecycle event. This parallel task then executes at the given offset and receives/sends data from/to a *data port*.

```
public class ThreadAInstance extends ThreadAImpl{
  private InDataPort<Object> dataIn;
  private InDataPort<Object> dataInWithOffset;
  ...

  private final void dispatch() {
    datain.receiveInput();
    new Handler(dataInWithOffset);
  }
  private final void start() {...}
  private final void compute() {...}
  private final void completion() {...}

  class InnerAsyncEventHandler
    extends AsyncEventHandler{
    ...

    public void handleAsyncEvent(){
      dispatch();
      start();
      compute();
      completion();
    }
  }
}

public class Handler extends BoundAsyncEventHandler{
  private InDataPort<Object> dataIn;

  public Handler(InDataPort<Object> dataIn){
    this.dataIn = dataIn;
    setSchedulingParameters
        (new PriorityParameters(5));
    Timer timer =
        new OneShotTimer
        (new RelativeTime(30,0),this);
    timer.start();
  }

  @Override
  public void handleAsyncEvent() {
    dataIn.receiveInput();
  }
}
```

Listing 13. Data port timing in RTSJ

In Listing 14, two exemplary *data port*s are given. First, an *in data port* with no explicitly defined *Input_Time* which is set to *Dispatch* and 0 ns offset per default. Second, an *in data port* with an *Input_Time* set to *Dispatch* and a positive offset between 30 ms and 40 ms. In Listing 13 the timings of those two *in data port*s are translated into the direct method call `datain.receiveInput()` and a parallel handler. The handler executes the same method with an offset of 30 ms which is the lower bound of the offset specified in the AADL model. The *Priority* of the handler is set to the same value as the *thread* that creates it, i.e. 5.

```
thread threadA
  features
    dataIn : in data port;
    dataInWithOffset : in data port
    {Input_Time => ([Time => Dispatch;
    Offset => 30ms .. 40ms]);};
  properties
    Priority => 5;
end threadA;

thread implementation threadA.impl
```

**end** threadA.impl

Listing 14. Data port timing in AADL

*3) Immediate, Delayed and Sampled Connections:* Although a specific *Input_Time* or *Output_Time* can be given for a *port*, the connection between two *port*s also has implicit effects on the timing aspects of a *port*. This implicit behavior is expressed by setting the *Timing* property of a connection between two *data port*s of two periodic *thread*s to *immediate*, *delayed* or *sampled*.

By default, all connections are set to be *sampled* as this has no effects on the timings given directly via *Input_Time* or *Output_Time*. The receiving *thread* would always receive the latest data from the sending *thread*.

For *immediate* connections, the *Input_Time* of the receiver is forced to be *start* as *IO_Reference_Time* and zero offset. The *Output_Time* for the sender is assumed to be *completion* as *IO_Reference_Time* and also zero offset, but can be overridden if a single value for *Output_Time* is given. An *immediate* connection enforces the receiver to be delayed until the sender completes execution. This ensures predictable communication within one dispatch frame, as depicted in Figure 2. In RTSJ this behavior is enforced by using a common synchronization object for the *immediate* connection, on which the receiving thread calls `wait()` as long as the call to `isDirty()` of the corresponding port returns `false`. The sending thread then wakes up the receiving thread after writing a new value to the respective port, by calling `notifyAll()` on the common synchronization object.

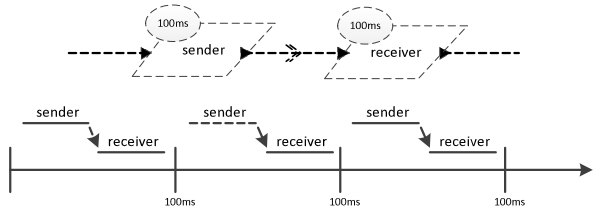


Fig. 2. Communication via Immediate Connection [2]

*Delayed* connections initiate the transmission of data at the deadline of the sender, thus having an *Output_Time* with *deadline* as *IO_Reference_Time* and zero offset. Accordingly, the receiver has an *Input_Time* with *dispatch* as *IO_Reference_Time* and zero offset as well. This way, the data is received at the next dispatch of the receiver following or equal to the sender's deadline as depicted in Figure 3.

### C. Communication-related Mapping

AADL defines several mechanisms concerning communication between components. Regarding our employed subset of AADL, we only consider port connections and the property *Classifier_Matching_Rule*. First, we will explain in detail how the mentioned property is semantically defined and then how the mapping of modelled *port connection*s into RTSJ code is done.

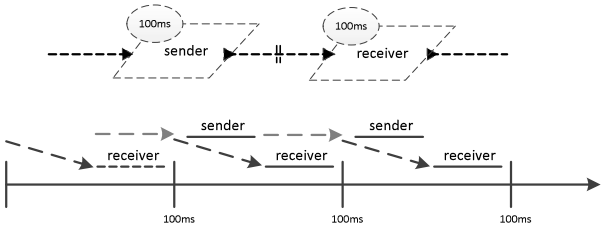


Fig. 3. Communication via Delayed Connection [2]

*1) Classifier_Matching_Rule:* The *Classifier_Matching_Rule* property defines how the classifiers of two connected *data port*s must conform to each other. Two possible values are currently of interest for our approach *Classifier_Match* and *Type_Extension*. Other possible values that are not covered by our work are *Equivalence*, *Subset* and *Conversion*. *Classifier_Match* is the simplest case, whereby the classifier of the source *port* has to match exactly the classifier given at the destination *port*. *Classifier_Match* is also the default value applying to every connection if not specified otherwise. The rule *Type_Extension* enforces the destination *port* to have a classifier that is either the same as the one of the source *port* or a subtype, e.g., a data type declared to be extending the source's classifier. By default, both possibilities are covered by the `extends` semantics of Java. The mapping declared in Section III-A automatically leads to valid Java code, regarding the classes used as parameter types in methods that represent *port*s.

*2) Port Connections: Port connection*s in AADL are always defined within a *component implementation declaration*. Each connection has a source port and a destination port. In Figure 4 a *process* with two *thread* subcomponents is depicted, where all components are connected via directed *port connection*s. For our mapping we identified two categories of *port connection*s within a given *component implementation declaration*. First, *port connection*s that have a subcomponent's port as destination, e.g., `downward` or `subcom_con1`. Second, *port connection*s that have a port of the component itself as destination, e.g., `upward`.

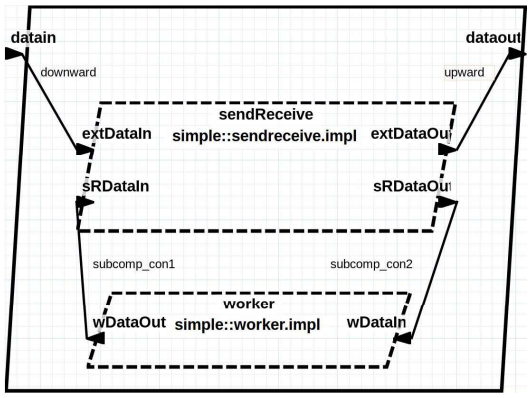


Fig. 4. Port connections within a component

In order to map these two categories of connections, we decided to generate a seperate class – a "ConnectionBroker" (CB) – for each component, that takes care of transmitting data over connections declared within that component. Listing 15 exemplarily shows a CB for the *process* depicted in Figure 4. The CB has a member variable (myself) for the *process* it manages the connections for, as well as for each subcomponent of myself, i.e., sendReceive and worker.

The actual transmission of data takes place in the method sendOnConnection(), where the name of the connection is passed as a unique identifier. If called, the method decides – based on the connection's name – which corresponding in/out methods of the component or one of its subcomponents have to be called. Depending on the chosen in/out method's parameter types, the transmitted data may have to be cast to the corresponding type.

In Listing 15, the case *"subcomp_con1"* is a representative of the first of the two above-named connection categories and represents the eponymous connection in Figure 4. As the destination port of this connection is sRDataIn of the *thread* subcomponent sendReceive, the corresponding method inSRDataIn() is called on the subcomponent member variable sendReceive. The port declares the base type Boolean as classifier, therefore the transmitted data is cast to this type.

The second category of connections is represented by the case *"upward"* in Listing 15. Also representing the eponymous connection in Figure 4, its destination is the *out data port* dataout of the component itself. Thus, the respective out method of myself is called. The given data is cast to Boolean, as declared by the *out data port* dataout.

```
public class ConnectionBroker{
  private SomeProcessImpl myself;
  private SendReceiveImpl sendReceive;
  ...

  public ConnectionBroker(Componet myself,
                  SendReceiveImpl sendReceive,...){
    this.myself = myself;
    ...
  }

  public void sendOnConnection(String connection,
                               Object data){
    switch (connection) {
      case "subcomp_con1":
        sendReceive.inSRDataIn((Boolean)data);
        break;
      ...
      case "upward":
        myself.outDataout((Boolean)data);
        break;
    }
  }
}
```

Listing 15. Connections in RTSJ

*3) In and Out methods:* As explained in Section III-C2, a CB is a sufficient possibility to manage the transmission of data within a component. In case of communication inside a subcomponent or outside of the given component, a CB calls the in/out methods of the respective component. The in/out methods take care of further routing the given data to their destination. The in methods of subcomponents handle this routing via their own CB, but in order to forward the given data over an out port of the component itself, a component has to use the CB of its parent component. Therefore, each component has – in addition to its own CB broker – a member variable parentBroker, as depicted in Listing 16. Below, the implementation of in and out methods of a generic component are explained in detail.

For in methods of a component, there are two possibilities. The first is, there are outgoing connections for the given *in data port* within the component. Thus, the method forwards the incoming data via the component's CB, as shown in the method inMethodForwarding() in Listing 16. The second is, there are no outgoing connections declared for the given *in data port* within the component. Then, the respective in method stores the incoming data within a designated port member variable, e.g. inPort, as depicted by the method inMethodFinal().

For out methods, there is only the possibility of forwarding the given data via the parentBroker. An out method resembles a broadcast as it sends the given data on all outgoing connections that are declared within its component's parent component. This is done via the sendOnPort() method of the parentBroker. This method works similar to the sendOnConnection() method, as shown in Listing 15. Merely the unique identifier is different, as several subcomponents of the parent component might have the same name for their *out data port*s. Thus, the concatenation of the components name and its *out data port*'s name is used.

```
public class Component{
  private ConnectionBroker broker;
  private ConnectionBroker parentBroker;
  private InDataPort<Object> inPort =
              new InDataPort<Object>();

  public void inMethodForwarding(Object data){
    broker.sendOnConnection("con1", data);
    ...
    broker.sendOnConnection("conX", data);
  }

  public void inMethodFinal(Object data){
    inPort.setFWData(data);
  }

  public void outMethodForwarding(Object data){
    parentBroker.sendOnPort
              ("compName+portName", data)
  }
}
```

Listing 16. In and out methods in RTSJ

## IV. Use Case

The following use case is based on an autopilot, developed by students of the practical course "Avionik Praktikum" at the University of Augsburg. The goal of this course was to write a working autopilot for a quadrocopter that is simulated in X-Plane [7]. The autopilot software is running on a

separate device – a Raspberry Pi 2 with special autopilot hardware from Erle Robotics S.L. [8] – and communicates with the simulation via UDP messages. The software is written completely in Java, respectively RTSJ, and is running within the JamaicaVM [5] from aicas GmbH on a real-time Linux. The current state of the software encompasses seven components which are loosely coupled via a software bus. There are four basic components – `PitchController`, `RollController`, `HeadingController` and `AltitudeController` – that control the pitch, roll, heading and altitude of the quadrocopter. These basic components translate higher level commands – `PitchCommand`, `RollCommand`, `HeadingCommand` and `AltitudeCommand` – from the `PositionController` and `FlightMissionExecutionController` into `ThrottleRequest`s for each of the quadrocopter's engines. Those `ThrottleRequests` are aggregated via the `MixThrottlesController` into one `ThrottleCommand` which is then sent back to the simulation. The whole communication is handled via messages that are sent over the `Bus` component which broadcasts every message to all registered components and also back to the *Simulation*. Figure 5 depicts this architecture and shows all messages being exchanged between the components.

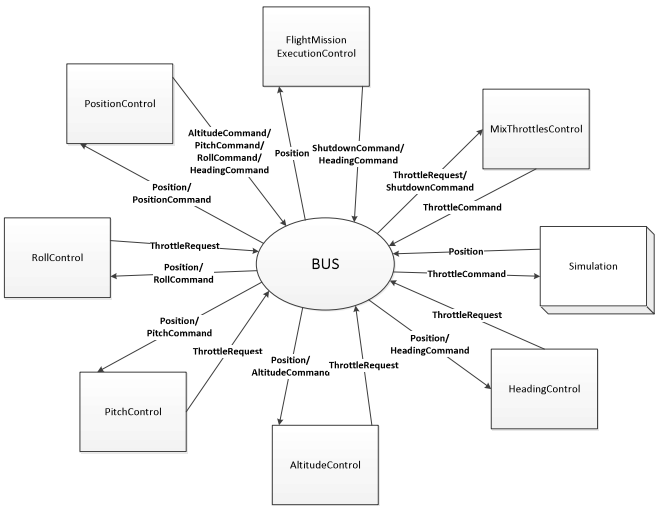


Fig. 5. Architecture of the Autopilot

Each component is currently implemented as a RTT. All *thread*s have a *Period* of 10 Hz or 100 ms, and are hence *periodic*. Thus, the messages are *sampled*, no *immediate* or *delayed* connection exists between the components. Given this starting point, we modeled the same autopilot with our approach, using the defined subset of AADL elements of Subsection II-B.

We decided to represent the messages that are sent over the bus via one common super *data type declaration*. This *data type declaration* `BusMessage` is then realized by several *data implementation declaration*s which represent the more specific messages and values of these messages as *data* subcomponents, e.g., the message `Position` is mapped onto a *data implementation declaration* that contains four *data* subcomponents: `altitude`, `latitude`, `longitude` and `loiter`. The first three subcomponents use *Basic_Type::Float* as classifier and the last one *Basic_Type::Boolean*.

Now, each component is mapped onto a *thread* in AADL, whereby the four basic controllers – `Pitch-`, `Roll-`, `Heading-` and `AltitudeController` – are modeled via AADL's extension and refinement mechanisms. The common parent *thread type declaration* is `BasicController` that defines the features `position` and `command` as *in data port*s and `throttleRequest` as an *out data port*. Afterwards, we create four *type declaration*s for for pitch, roll, heading and altitude, extending `BasicController` and refining the `command` *in data port* to its corresponding message type, e.g., `PitchCommand`, `RollCommand`, etc. The remaining controllers are modeled as separate *thread type declaration*s, as they do not share common features.

We were not able to map the `Bus` component, because the defined subset of AADL only allows *data port*s for communication. Thus, we connected each sending component with its receiving counterparts, resulting in the explicit communication architecture depicted in Figure 6 in contrast to the implicit communication architecture in Figure 5.

After finishing the modeling we generated the *system implementation declaration* depicted in Figure 6. As we do not consider AADL model elements like *process*, *device* or *system* yet, only seven *thread*s with their corresponding inheritance hierarchy were generated together with their features and timing properties. The messages were also generated and were used as parameter types for the in and out methods of each *thread*'s *port*s. The only work left to be done by a programmer is writing the control logic for the designated methods of each controller and assemble those threads by instantiating them in a main class. The generated code takes care of timing and communication aspects of all *thread*s without any influence by the programmer.

## V. Related Work

In the context of AADL, a lot of work was done regarding code generation for different target platforms like [9] or [10]. [9] focuses on AADL and Simulink [11] for modeling architecture and behavior and then generating the corresponding Ada and SPARC code. [10] is a stand-alone AADL model processor that supports code generation, targeting C real-time operating systems and Ada for native and Ravenscar targets. However, as we decided to use RTSJ as target language, we concentrate on work that has the same target language or at least Java without the real-time capabilities of RTSJ. To the best of our knowledge, we are aware of three different approaches, namely [12], [13] and [14].

[13] focuses on medical applications and a generation of AADL system models into Java code for a given reference platform. The generated code has to be compliant with

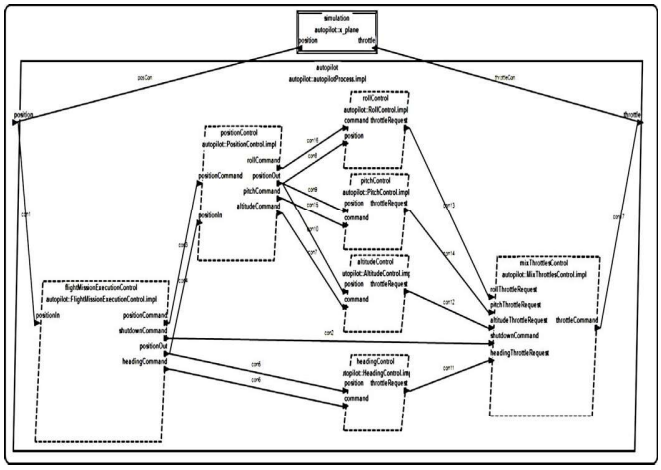


Fig. 6. Architecture of the Autopilot in AADL

the specified publish/subscribe mechanisms of the underlying middleware. As reference platform, they use an open source "Medical Application Platform" (MAP) called "Medical Device Coordination Framework" (MDCF). Although some of the presented generation mechanism are relevant for our work, the highly specialized target platform contradicts our more general-purpose approach, as we use plain RTSJ without any further assumptions about the underlying platform.

[12] indeed uses RTSJ as target language, but focuses on the partitioning of systems defined in AADL for ARINC 653 compliant systems and how this partitioning can be maintained in the generated code. Moreover, a major part of the work deals with communication between partitions and is not concerned with a generalized generation approach.

[14] in contrast is very similar to our approach regarding their goal of a general mapping between AADL and RTSJ. However, their work is simplifying most of the aspects that our work investigates in detail. To allege an example, they simplify the Data Port communication to always happen either at dispatch, start or deadline. The properties *Input_Time* and *Output_Time* are completely ignored as well as timings dictated by the *Timing* property of data port connection. Threads are not forced to run consecutively if a Data Port connection between them is marked as immediate. Another distinction to our work is the targeted version of RTSJ. While they are targeting version 1.0.2, we use mechanisms from version 2.0 which facilitates the realization of semantics determined by AADL.

## VI. Conclusion

In this work we presented a mapping approach from an AADL subset to RTSJ which maintains the semantics given by the AADL standard. This approach enables developers of real-time, performance-critical systems to shift structure, timing and communication-related concerns into design phase. Hence, they are able to perform analyses regarding communication and timing during design phase, while resting assured that the implementation will reflect their design choices. The application of our approach is shown via the implementation of an autopilot for quadrocopters. For this purpose the software of the quadrocopter is modeled in AADL and is then generated by our implementation. The usecase shows three advantages of our approach over an implementation without code-generation:

- The speed-up of development by letting the programmer focus on application logic instead of writing recurring code concerned with timing and communication.
- A less error-prone transition from the design of a system to its implementation.
- The possibility of an earlier detection of timing- or communication-related errors in the system.

Taken some steps further, this approach can lead to the possibility of simple Java developers, writing real-time systems, designed by one competent real-time system designer. In our further research we will investigate a broader subset of AADL which encompasses *event (data) ports*, *aperiodic* and *sporadic threads* and also the error model annex of AADL. Especially the last one is of interest in order to integrate safety-related aspects like error-propagation into our existing approach, by exploiting Java's exception- and RTSJ's asynchronous-transfer-of-control (ATC) mechanisms.

## References

[1] Architecture Analysis and Design Language. Accessed 09. June 2016. [Online]. Available: http://www.aadl.info/aadl/currentsite/

[2] P. H. Feiler and D. P. Gluch, *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[3] JSR 1: Real-time Specification for Java. Accessed 09. June 2016. [Online]. Available: https://jcp.org/en/jsr/detail?id=1

[4] JSR 282: RTSJ version 1.1. Accessed 09. June 2016. [Online]. Available: https://jcp.org/en/jsr/detail?id=282

[5] JamaicaVM. Accessed 09. June 2016. [Online]. Available: http://www.aicas.com/cms/en/JamaicaVM

[6] Default Methods. Accessed 09. June 2016. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

[7] X-Plane 10. Accessed 09. June 2016. [Online]. Available: http://www.x-plane.com/desktop/home/

[8] Erle Brain 2, a Linux brain for robots and drones. Accessed 09. June 2016. [Online]. Available: https://erlerobotics.com/blog/product/erle-brain-v2/

[9] M. Bordin, C. Comar, E. Falis, F. Gasperoni, Y. Moy, E. Richa, and J. Hugues, "System to software integrity: A case study," in *Embedded Real-Time Software and Systems 2014*, , FR, 2014. [Online]. Available: http://oatao.univ-toulouse.fr/10939/

[10] Open AADL. Accessed 09. June 2016. [Online]. Available: http://www.openaadl.org/ocarina.html

[11] Simulink: Simulation und Model-Based-Design. Accessed 09. June 2016. [Online]. Available: http://de.mathworks.com/products/simulink/

[12] Y. Wang, D. Ma, Y. Zhao, L. Zou, and X. Zhao, "Automatic rt-java code generation from aadl models for arinc653-based avionics software," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. IEEE, 2012, pp. 670–679.

[13] S. Procter and J. Hatcliff, "Robby: towards an AADL-based definition of app architectures for medical application platforms," in *Proceedings of the International Workshop on Software Engineering in Healthcare. Washington, DC*, 2014.

[14] B. Jean-Paul, C. Raphaël, C. David, F. Mamoun, and R. Jean-François, "A mapping from aadl to java-rtsj," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. ACM, 2007, pp. 165–174.