A_2 Concurrency Framework

Florian Negele, ETH Zurich

June 3, 2009

Abstract

This documentation describes the purpose, design decisions, functionality, and the rules for a proper usage of the generic A_2 Concurrency Framework that was designed to aid programmers of industrial control software to overcome the complexity of concurrent programs. It is subdivided into a comprehensive user manual and a complete reference guide.

Contents

1 Introduction

Well trained application programmers follow good advice if they model their programs on real life counterparts. Unfortunately those archetypes can seldom or almost never be directly mapped to the sequential nature of the imperative programming paradigms most often used today. In reality, they are characterized by great portions of autonomy and independence. Additionally driven by the wish to keep up with the current engineering progress in the field of multithreaded hardware, programmers therefore often try to achieve the same level of inherent parallelism imposed by these attributes. Regrettably, the complexity and care of handling concurrent programming arisen from this goal are often overwhelming for programmers. One of the main purpose of the A_2 Concurrency Framework is to help and assist programmers to overcome this burden.

1.1 Background

The A_2 Concurrency Framework is a programming framework and consists of several generic components bundled in modules. It is written in the programming language Active Oberon targeting the operating system A_2 developed and maintained by the group for Native Systems at ETH Zurich. The framework was designed, implemented and applied in a real-life industrial control software during the course of a CTI project which foster knowledge and technology transfers between companies and universities.

1.2 Focus

Although the broad genericity of the A_2 Concurrency Framework and the underlying concepts allows it to be used in many various contexts, this documentation and the terms used in it focus on practical topics typically found in industrial software applications. The intention is to provide a concrete yet non-trivial example of its use along the lines of a real-life domain-specific application.

2 Design

The A_2 Concurrency Framework provides several generic components bundled in a module called A2Sequencers. For a detailed reference of the this module, its components, and their usage see the description of the module `A2Sequencers` in the reference part, section 4 on page 11. This section describes the structure and concepts of the programming framework. For a discussion and examples on how to use its components in an application refer to the manual in section 3.

2.1 Control Loops

The most prominent of the processes within the context of industrial control software are the repetitive tasks of measuring, controlling and actuating. These endlessly performed tasks form the so-called *control loop*. Figure 1 shows the generic structure of a typical layout of such a control loop.

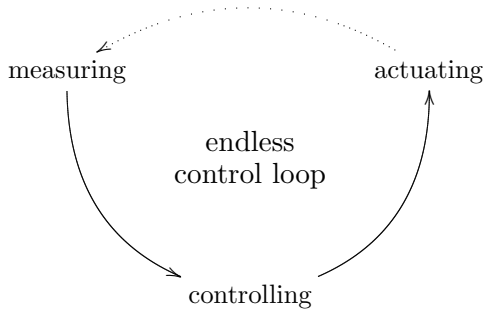

Figure 1: Generic control loop typically found in industrial control software

Although each of the three generic tasks is concurrent and runs in parallel to the others, they do not run completely independently but rather communicate by means of transferred data that has to be processed within the following stages:

Measuring: A new set of data is gathered by external measuring devices, so called *sensors*.

Controlling: The data is collected and processed by *controllers* in order to compute the current state of the application and to accordingly establish the target state by modifying control parameters.

Actuating: External devices are driven according to the updated control parameters by so-called *actuators*.

Eventhough the tasks perform their duties in parallel, they pass the data to be processed consecutively to each other. Therefore, the three stages through which a specific set of data passes throughout the whole processing, can be regarded as one run within an endless control loop. The next sections describe the abstractions and models obtained from this generic discussion.

2.2 Sequencers

The main idea behind the framework is to support a concurrent programming model that is transparent to the programmer by explicitly *sequentializing* parallel and concurrent processes. The two models provided by the A_2 Concurrency Framework to implement this idea are *sequencers* and *messages*. Naturally, tasks like measuring, controlling and actuating as described in section 2.1 can be implemented using sequencers. They are generic components running concurrently by maintaining an activity of their own (i.e. they are implemented as active objects).

For a detailed specification of the provided application interface with respect to sequencers see the description of the object `Sequencer` in the reference part, section 4.3 on page 15.

2.2.1 Messages

Besides their concurrent activity, each sequencer maintains a *message queue*. It keeps track of all messages sent to respective sequencer because this is the main way of communicating between the activities of the sequencers. Sequencers may call procedures of other sequencers and even wait for their result, which in both cases implicitly results in a new message being inserted into the message queue of the target sequencer. The code of the procedure associated with the message is not executed by the sequencer calling the procedure. Rather the activity of the target sequencer periodically checks incoming messages and calls the requested procedure on its own. The actual procedure call is therefore deferred to the active object owning the procedure.

Note: Programmers should be aware that there might be a delay between the actual execution of a procedure and its call which does not block and does return immediately.

Thus, procedure calls steming from different sequencers (and therefore different activities) are automatically decomposed into messages and therefore sequentialized behind the scenes. For this reason, programmers are no longer required to explicitly program the process synchronization on their own and can rather focus on more important aspects of their application.

2.2.2 Requests

Since messages are handled within the context of another active object, the actual procedure invocation is asynchronous. But for callers of procedures taking variable arguments or returning a value it is actually crucial to wait for the result. For these cases, the A_2 Concurrency Framework provides a special kind of messages called *requests*. Requests are handled like normal messages with the exception of the actual procedure invocations being synchronous.

For a detailed reference of the programming interface concerning messages and requests see the description of the object `Message` in the reference part, section 4.1 on page 11.

2.3 Properties

The A_2 Concurrency Framework provides *properties* in order to allow programmers to model the state of their application. The state of the program is often shared between and accessed by several different sequencers. E.g. sequencers implementing measuring devices (see 2.1) may store the data they read in properties. Sequencers implementing controllers on the other hand may find the need to read that data and process it further.

The flawless and concurrent access of the sequencers to the data stored in properties is handled by the framework behind the scenes. It uses a lock-free access scheme that allows several concurrent readers while synchronizing potential writers. Additionally, properties support the observer pattern which allows interested components (e.g. sequencers or logging facilities) to install (subscribe) so-called notification handlers which are called whenever the value of a property changes. Properties are therefore well suited as the model within a model view controller environment.

For a detailed list of all supported types of properties see the description of the object `Property` in the reference part, section 4.4 on page 17.

3 Manual

For each component in an application that physically or even logically has an activity of its own, this component must be implemented as an Active Oberon object extending the generic `Sequencer` object provided by the A_2 Concurrency Framework (see the description of the object `Sequencer` in the reference part, section 4.3 on page 15).

3.1 User-defined Activities

Sequencers are implemented as active objects and provide a default activity. Per default, it repeatedly waits in the first stage for messages sent to the sequencer and handles them accordingly in the second stage. The behaviour of the first stage can be user-defined by overwriting the procedure `Handle`. In this case, the sequencer will not wait for messages but periodically call this procedure.

Figure 2 shows a sample controller implemented using a sequencer and a user-defined activity. It maintains a private counter variable that is incremented with each implicit invocation of the `Handle` procedure.

```
TYPE Controller* = OBJECT (A2Sequencers.Sequencer)

   VAR counter: LONGINT;

   PROCEDURE Handle;
   BEGIN
     INC (counter);
   END Handle;

END Controller;
```

Figure 2: Sample controller implemented as a sequencer

Note how the access to the variable `counter` has not to be protected from concurrent access even in the presence of other sequencers resetting the counter as shown in the following section.

Rules:

- Application components with explicit or implicit concurrent behaviour must be implemented as objects extending the object `Sequencer`.
- The concurrent activity of sequencers may be user-defined. But it is important that the user-defined `Handle` procedure returns regularly in order to allow the framework to handle incoming messages.

3.2 Communication

The A_2 Concurrency Framework provides three distinct ways how sequencers should communicate with each other. Communication in this context means passing information between concurrent sequencers by means of lock-free properties or sequentialized messages.

3.2.1 Message Passing

Each procedure of an object extending the generic `Sequencer` object must check wether it was called by the sequencer itself or by another entity. Figure 3 shows an expanded version of the previous controller that allows to reset its counter.

```
TYPE Controller* = OBJECT (A2Sequencers.Sequencer)

   VAR counter: LONGINT;

   PROCEDURE Reset*;
   BEGIN
     IF SequencerCalledThis () THEN counter := 0;
     ELSE AddMessage (Reset) END;
   END Reset;

   PROCEDURE Handle;
   BEGIN INC (counter);
   END Handle;

END Controller;
```

Figure 3: Sample controller with a resettable counter

As the procedure `Reset` may be called by other sequencers or even other instances of the same controller type, one has to make sure that the actual code of resetting the counter is executed by the correct sequencer instance. The procedure `SequencerCalledThis` retrieves wether the code can directly be executed since the procedure is actually called by the correct sequencer, or if the latter has to be driven to call this procedure using a corresponding message. In the latter case the message is appended to the message queue of the sequencer by calling the `AddMessage` procedure.

Note: A call to the procedure `Reset` as given in figure 3 does never block the caller. It is therefore not guaranteed that the counter of the controller as actually been resetted after returning from the procedure.

3.2.2 Request Disposal

In order to guarantee the counter to be reset the A_2 Concurrency Framework provides request messages that allow callers to be synchronized with the request. Figure 4 shows a slightly modified version of the previous controller that allows to reset its counter synchronously.

```
TYPE Controller* = OBJECT (A2Sequencers.Sequencer)

   VAR counter: LONGINT;

   PROCEDURE Reset*;
   BEGIN
     IF SequencerCalledThis () THEN counter := 0;
     ELSE AddRequest (Reset) END;
   END Reset;

   PROCEDURE Handle;
   BEGIN INC (counter);
   END Handle;

END Controller;
```

Figure 4: Sample controller with synchronous resetting

The difference of this version is the modified call to the `AddRequest` procedure. As its name foretells it appends a request instead of a message to the message queue of the sequencer causing the calling code to be blocked until the corresponding request has been completely handled and processed by the request. It is therefore guaranteed in this example, that the counter has been resetted after returning from a call to `Reset`.

Note: As appending requests to the message-queue blocks the caller one has to take care not to add requests that spawn further recursive requests as this opens up the possibility of two or more sequencers waiting on each other without any progress whatsoever.

Rules:

- Each procedure that might be called from within other instances of sequencer objects must contain a check for the correct execution environment.
- Procedures that return a value, take a variable parameter, or have to be processed before returning must be added to the message queue using requests.

3.2.3 Property Modification

As long as a variable is associated with one instance of a sequencer object, it is best to model it as a private field of that object and to synchronize the access to its value using messages and requests as desribed above.

If the variable reflects a global state of the application or is shared between several distinct entities (like e.g. customized interrupt handlers or the graphical user interface) it is good practice to store the value inside a property. They allow lock-free yet concurrent access to the value they store. Figure 5 shows further modified version of the previous controller example that stores its counter in a global property.

```
VAR counter: A2Sequencers.Integer;

TYPE Controller* = OBJECT (A2Sequencers.Sequencer)

   PROCEDURE Handle;
   BEGIN counter.Inc (1);
   END Handle;

END Controller;

PROCEDURE Reset*;
BEGIN counter.Set (0);
END Reset;
```

Figure 5: Sample controller with global counter

This controller uses a global property of type integer (see the description of the object `Integer` in the reference part, section 4.4.3 on page 19). The access to its value is provided by the procedures `Set` and `Incl` as given in this example. Note how access to properties needs no further attention from the programmer as the potentially concurrent access is handled by the framework.

Note: Only the access to the value is atomic, but not a sequence of accesses because of potential interferences of other active objects.

Rules:

- Properties can be used for global variables that store state shared between sequencers.
- Properties may not be used when a sequence of accesses must be atomic. Code called by messages is always atomic.

3.3 Summary of Rules

This section sumarizes all rules that must be obeyed in order to guarantee a safe usage of the A_2 Concurrency Framework.

- Application components with explicit or implicit concurrent behaviour must be implemented as objects extending the object `Sequencer`.
- The concurrent activity of sequencers may be user-defined. But it is important that the user-defined `Handle` procedure returns regularly in order to allow the framework to handle incoming messages.
- Each procedure that might be called from within other instances of sequencer objects must contain a check for the correct execution environment.
- Procedures that return a value, take a variable parameter, or have to be processed before returning must be added to the message queue using requests.
- Properties can be used for global variables that store state shared between sequencers.
- Properties may not be used when a sequence of accesses must be atomic. Code called by messages is always atomic.

4 Reference Guide

This section describes the generic user interfaces of all types defined by the A_2 Concurrency Framework. All types are defined in and available for users by importing the module `A2Sequencers`.

4.1 Messages

The A_2 Concurrency Framework defines the abstract type `Message` for messages which are used for asynchronous communication between sequencers (see section 4.3).

```
TYPE Message* = OBJECT

  PROCEDURE & InitMessage*;

  PROCEDURE Handle*;

END Message;
```

The procedure `InitMessage` is the initializer of the object and must be called by derived types within their own initializers. The abtract procedure `Handle` must be overwritten in derived types. It is called by the sequencer to which the message has been sent to. As soon as the sequencer is ready to process its messages, it will call the procedure `Handle` for each message in its message queue after removing it.

4.1.1 Utility Message Types

The A_2 Concurrency Framework defines handy message types for the most common procedure variable types while still providing the means for users to define their own variations. Sections 4.1.2 to 4.1.7 describe the interface of the message types predefined by the A_2 Concurrency Framework.

4.1.2 Procedure Messages

The message type `ProcedureMessage` can be used to defer a call to a procedure taking no arguments.

```
TYPE ProcedureMessage* = OBJECT (Message)

  PROCEDURE & InitProcedureMessage* (procedure: Procedure);

END ProcedureMessage;

TYPE Procedure = PROCEDURE {DELEGATE};
```

If the sequencer handles a message of this type, the procedure variable specified in the initializer `InitProcedureMessage` will be called.

4.1.3 Boolean Messages

The message type `BooleanMessage` can be used to defer a call to a procedure taking one argument of type `BOOLEAN`.

```
TYPE BooleanMessage* = OBJECT (Message)

  PROCEDURE & InitBooleanMessage* (value: BOOLEAN;
                        procedure: BooleanProcedure);
END BooleanMessage;

TYPE BooleanProcedure = PROCEDURE {DELEGATE}
                        (value: BOOLEAN);
```

If the sequencer handles a message of this type, the procedure variable specified in the initializer `InitBooleanMessage` will be called provided with the first parameter as argument.

4.1.4 Integer Messages

The message type `IntegerMessage` can be used to defer a call to a procedure taking one argument of type `LONGINT`.

```
TYPE IntegerMessage* = OBJECT (Message)

  PROCEDURE & InitIntegerMessage* (value: LONGINT;
                        procedure: IntegerProcedure);
END IntegerMessage;

TYPE IntegerProcedure = PROCEDURE {DELEGATE}
                        (value: LONGINT);
```

If the sequencer handles a message of this type, the procedure variable specified in the initializer `InitIntegerMessage` will be called provided with the first parameter as argument.

4.1.5 Real Messages

The message type `RealMessage` can be used to defer a call to a procedure taking one argument of type `REAL`.

```
TYPE RealMessage* = OBJECT (Message)

  PROCEDURE & InitRealMessage* (value: REAL;
                        procedure: RealProcedure);
END RealMessage;

TYPE RealProcedure = PROCEDURE {DELEGATE} (value: REAL);
```

If the sequencer handles a message of this type, the procedure variable specified in the initializer `InitRealMessage` will be called provided with the first parameter as argument.

4.1.6 Set Messages

The message type `SetMessage` can be used to defer a call to a procedure taking one argument of type `SET`.

```
TYPE SetMessage* = OBJECT (Message)

  PROCEDURE & InitSetMessage* (value: SET;
                        procedure: SetProcedure);
END SetMessage;

TYPE SetProcedure = PROCEDURE {DELEGATE} (value: SET);
```

If the sequencer handles a message of this type, the procedure variable specified in the initializer `InitSetMessage` will be called provided with the first parameter as argument.

4.1.7 String Messages

The message type `StringMessage` can be used to defer a call to a procedure taking one argument of type `ARRAY OF CHAR`.

```
TYPE StringMessage* = OBJECT (Message)

  PROCEDURE & InitStringMessage* (CONST value: ARRAY OF CHAR;
                        procedure: StringProcedure);
END StringMessage;

TYPE StringProcedure = PROCEDURE {DELEGATE}
                        (CONST value: ARRAY OF CHAR);
```

If the sequencer handles a message of this type, the procedure variable specified in the initializer `InitStringMessage` will be called provided with the first parameter as argument.

4.2 Requests

The A_2 Concurrency Framework defines the type `Request` for requests which are used for synchronous communication between sequencers (see section 4.3).

```
TYPE Request* = OBJECT (Message)

  PROCEDURE & InitRequest*;

  PROCEDURE Handle*;

END Request;
```

The procedure `InitRequest` is the initializer of the object and must be called by derived types within their own initializers. The procedure `Handle` should be overwritten in derived types.

Note: It is crucial that objects deriving from the object `Request` make a super-call to the procedure `Handle` *at the end* of their own version of `Handle`.

4.2.1 Utility Request Types

The A_2 Concurrency Framework defines handy request types for the most common procedure variable types while still providing the means for users to define their own variations. Sections 4.2.2 to 4.2.4 describe the interface of the request types predefined by the A_2 Concurrency Framework.

4.2.2 Boolean Requests

The request type `RequestBoolean` can be used to defer a call to a procedure returning a value of type `BOOLEAN`.

```
TYPE RequestBoolean* = OBJECT (Request)

  VAR result-: BOOLEAN;

  PROCEDURE & InitRequestBoolean*
                      (procedure: ProcedureBoolean);
END RequestBoolean;

TYPE ProcedureBoolean = PROCEDURE {DELEGATE}
                      (): BOOLEAN;
```

If the sequencer handles a request of this type, the procedure variable specified in the initializer `InitRequestBoolean` will be called and its result will be stored in the variable `result`.

4.2.3 Integer Requests

The request type `RequestInteger` can be used to defer a call to a procedure returning a value of type `LONGINT`.

```
TYPE RequestInteger* = OBJECT (Request)

  VAR result-: LONGINT;

  PROCEDURE & InitRequestInteger*
                      (procedure: ProcedureInteger);
END RequestInteger;

TYPE ProcedureInteger = PROCEDURE {DELEGATE}
                      (): LONGINT;
```

If the sequencer handles a request of this type, the procedure variable specified in the initializer `InitRequestInteger` will be called and its result will be stored in the variable `result`.

4.2.4 Real Requests

The request type `RequestReal` can be used to defer a call to a procedure returning a value of type `REAL`.

```
TYPE RequestReal* = OBJECT (Request)

  VAR result-: REAL;

  PROCEDURE & InitRequestReal* (procedure: ProcedureReal);

END RequestReal;

TYPE ProcedureReal = PROCEDURE {DELEGATE}: REAL;
```

If the sequencer handles a request of this type, the procedure variable specified in the initializer `InitRequestReal` will be called and its result will be stored in the variable `result`.

4.3 Sequencers

The A_2 Concurrency Framework defines the type `Sequencer` modelling concurrent components communicating with each other:

```
TYPE Sequencer* = OBJECT
```

```
    PROCEDURE & InitSequencer;

    PROCEDURE Stop*;
    PROCEDURE Handle*;

    PROCEDURE Add* (message: Message; time: LONGINT);
    PROCEDURE Remove* (message: Message);

    PROCEDURE SequencerCalledThis* (): BOOLEAN;

  END Sequencer;
```

The procedure `InitSequencer` is the initializer of the object and must be called by derived types within their own initializers.

Sequencers do have their own activity (i.e. they are active objects) which can be stopped from execution for termination purposes by calling the procedure `Stop`. While the activity of a sequencer is running, the procedure `Handle` is called sequentally. This procedure can therefore be overwritten if a derived sequencer type has to supply its own activity.

Note: User-defined versions of the procedure `Handle` are not allowed to make super-calls. In order to guarantee the responsiveness of sequencers to handle messages, it is equally important that the procedure `Handle` does not block.

The procedure `Add` allows to append a message to the message queue of the sequencer. If the type of the message added to a sequencer is a request, the call to the procedure `Add` will block until the request was actually handled. The second parameter `time` allows to specify an absolute point in time before the message must not be handled by the sequencer. For delayed messages one can use the global procedure `Delay` which allows to specify relative point in times. If the message shall be executed as soon as possible, one can use the global constant `NoDelay` as argument to this parameter. The procedure `Remove` allows to remove messages from the message queue of the sequencer. This is useful for cancelling delayed messages.

The procedure `SequencerCalledThis` must be used called within every procedure that is directly called by other sequencers. It returns a Boolean value indicating wether the procedure was called by the sequencer itself in which case the procedure body should be executed. If it was not called by the sequencer itself, one has to skip the procedure body and add a corresponding message to the sequencer which causes the sequencer to execute this procedure again and therefore also its procedure body.

4.3.1 Utility Procedures for Messages

Sequencers additionally support some handy procedures to add certain types of messages. They are provided to release the programmer from the burden of manually define message types that are most often used.

```
PROCEDURE AddMessage* (procedure: Procedure);
PROCEDURE AddBooleanMessage* (value: BOOLEAN;
                    procedure: BooleanProcedure);
PROCEDURE AddIntegerMessage* (value: LONGINT;
                    procedure: IntegerProcedure);
PROCEDURE AddRealMessage* (value: REAL;
                    procedure: RealProcedure);
PROCEDURE AddSetMessage* (value: SET;
                    procedure: SetProcedure);
PROCEDURE AddStringMessage* (CONST value: ARRAY OF CHAR;
                    procedure: StringProcedure);
```

All these procedures `AddMessage`, `AddBooleanMessage`, `AddIntegerMessage`, `AddRealMessage`, `AddSetMessage` and `AddStringMessage` add the message of the specified procedure type to the message queue using the procedure `Add`. The messages are not delayed. Please refer to section 4.1.1 for a description of the different predefined message types.

4.3.2 Utility Procedures for Requests

Sequencers additionally support some handy procedures to add certain types of requests. They are provided to release the programmer from the burden of manually define request types that are most often used.

```
PROCEDURE AddRequestBoolean*
                    (procedure: ProcedureBoolean): BOOLEAN;
PROCEDURE AddRequestInteger*
                    (procedure: ProcedureInteger): LONGINT;
PROCEDURE AddRequestReal*
                    (procedure: ProcedureReal): REAL;
```

All these procedures `AddRequestBoolean`, `AddRequestInteger` and `AddRequestReal` add the request of the specified procedure type to the message queue using the procedure `Add`. The requests are not delayed. Please refer to section 4.2.1 for a description of the different predefined request types.

4.4 Properties

The A_2 Concurrency Framework defines an abstract type `Property` for different kinds of properties. Its generic interface looks like the following:

```
TYPE Property* = OBJECT

  VAR container*: OBJECT;

  PROCEDURE & InitProperty;

END Property;
```

The variable `container` isn't needed by the framework but is provided for applications to store further information about a property. The procedure `InitProperty` is the initializer of the object and must be called by derived types within their own initializers.

4.4.1 Common Behaviour of Properties

The values of the conrete property-types defined in the following subsections are not stored in the abstract object `Property`. But it actually defines a lock-free access scheme that allows several concurrent readers and one writer at a time to atomically access the value of the concrete property.

Every concrete property type additionally defines a procedure `AddHandler` which allows to install notification handlers. These handlers are delegates and called by the property each time the concrete value of a property changes (caused by executing the corresponding procedure `Set`).

4.4.2 Boolean Properties

Properties of type `Boolean` store values of type `BOOLEAN` .i.e. the values `TRUE` and `FALSE`.

```
TYPE Boolean* = OBJECT (Property)

  PROCEDURE & InitBoolean* (value: BOOLEAN);

  PROCEDURE Get* (): BOOLEAN;
  PROCEDURE Set* (value: BOOLEAN);

  PROCEDURE AddHandler* (handler: BooleanHandler);

END Boolean;

TYPE BooleanHandler = PROCEDURE {DELEGATE}
                      (property: Boolean; value: BOOLEAN);
```

The procedure `InitBoolean` is the initializer of the property which is used to set its initial value. The procedures `Set` and `Get` allow users to set and

retrieve the concrete Boolean value of the property. A notification handler can be installed by calling the procedure `AddHandler`. Its functionality as well as the characteristics of concurrent read and write accesses are common to all property types and are described in section 4.4.1.

4.4.3 Integer Properties

Properties of type `Integer` store values of type `LONGINT` which are 32-bit signed integers.

```
TYPE Integer* = OBJECT (Property)

  PROCEDURE & InitInteger* (value: LONGINT);

  PROCEDURE Get* (): LONGINT;
  PROCEDURE Set* (value: LONGINT);

  PROCEDURE Inc* (step: LONGINT);
  PROCEDURE Dec* (step: LONGINT);

  PROCEDURE AddHandler* (handler: IntegerHandler);

END Integer;

TYPE IntegerHandler = PROCEDURE {DELEGATE}
                      (property: Integer; value: LONGINT);
```

The procedure `InitInteger` is the initializer of the property which is used to set its initial value. The procedures `Set` and `Get` allow users to set and retrieve the concrete integer value of the property. The procedures `Inc` and `Dec` provide handy shortcuts to increment and decrement the integer value. They have the same semantics as the built-in procedures `INC` and `DEC` and behave exactly like as the `Set` procedure. A notification handler can be installed by calling the procedure `AddHandler`. Its functionality as well as the characteristics of concurrent read and write accesses are common to all property types and are described in section 4.4.1.

4.4.4 Real Properties

Properties of type `Real` store values of type `REAL`, which are single precision floating point numbers.

```
TYPE Real* = OBJECT (Property)

  PROCEDURE & InitReal* (value: REAL);
```

```
  PROCEDURE Get* (): REAL;
  PROCEDURE Set* (value: REAL);

  PROCEDURE AddHandler* (handler: RealHandler);

END Real;

TYPE RealHandler = PROCEDURE {DELEGATE}
                      (property: Real; value: REAL);
```

The procedure `InitReal` is the initializer of the property which is used to set its initial value. The procedures `Set` and `Get` allow users to set and retrieve the concrete floating point number of the property. A notification handler can be installed by calling the procedure `AddHandler`. Its functionality as well as the characteristics of concurrent read and write accesses are common to all property types and are described in section 4.4.1.

4.4.5 Set Properties

Properties of type `Set` store values of type `SET`, which are 32-bit or 64-bit sized bitsets depending on the wordsize of the CPU architecture.

```
TYPE Set* = OBJECT (Property)

  PROCEDURE & InitSet* (value: SET);

  PROCEDURE Get* (): SET;
  PROCEDURE Set* (value: SET);

  PROCEDURE Incl* (element: LONGINT);
  PROCEDURE Excl* (element: LONGINT);

  PROCEDURE AddHandler* (handler: SetHandler);

END Set;

TYPE SetHandler = PROCEDURE {DELEGATE}
                      (property: Set; value: SET);
```

The procedure `InitSet` is the initializer of the property which is used to set its initial value. The procedures `Set` and `Get` allow users to set and retrieve the concrete bitset of the property. The procedures `Incl` and `Excl` provide handy shortcuts to include and exclude individual set elements. They have the same semantics as the built-in procedures `INCL` and `EXCL`

and behave exactly like as the `Set` procedure. A notification handler can be installed by calling the procedure `AddHandler`. Its functionality as well as the characteristics of concurrent read and write accesses are common to all property types and are described in section 4.4.1.

4.4.6 String Properties

Properties of type `String` store values of type `ARRAY OF CHAR` whereas the array size is user-defined.

```
TYPE String* = OBJECT (Property)

  PROCEDURE & InitString* (CONST value: ARRAY OF CHAR;
                      length: LONGINT);

  PROCEDURE Get* (VAR value: ARRAY OF CHAR);
  PROCEDURE Set* (CONST value: ARRAY OF CHAR);

  PROCEDURE AddHandler* (handler: StringHandler);

END String;

TYPE StringHandler = PROCEDURE {DELEGATE} (property: String;
                      CONST value: ARRAY OF CHAR);
```

The procedure `InitString` is the initializer of the property which is used to set its initial value. The parameter `length` specifies the length of the internal character array to be allocated. The procedures `Set` and `Get` allow users to set and retrieve the concrete character array of the property. A notification handler can be installed by calling the prcedure `AddHandler`. Its functionality as well as the characteristics of concurrent read and write accesses are common to all property types and are described in section 4.4.1.

Index