Customizing UML for the development of distributed reactive systems and code generation to Ada 95

Michael Kersten, Jörg Matthes, Christian Fouda Manga, Stephan Zipser, Hubert B. Keller
Forschungszentrum Karlsruhe, D-76021 Karlsruhe, Germany, www.iai.fzk.de

Abstract

Distributed, reactive software systems, e.g. process control tools, can be modelled with the Unified Modelling Language (UML). Recently such UML models are used to generate source code automatically. Because of the complexity of UML, it is necessary to restrict the usage of its constructs by defining UML profiles to allow the automatic generation of source code. In this paper a UML profile for the design and implementation of distributed, reactive systems and an associated mapping to Ada 95 source code are introduced. Further, our experiences with the chosen approach are discussed.

Keywords:UML ,automatic generation of code, Ada.

1 Introduction

In our research group at the Forschungszentrum Karlsruhe we optimize industrial processes using methods like infrared thermography, fuzzy control and image processing. Since available process control tools have not been able to implement complex algorithms in a time effective and comfortable way we decided to overcome this situation with the development of the process control tool Inspect 2 [1].

First of all, the Inspect 2 system can be classified as a reliable, distributed, reactive system. Second, the area of research implies high maintainability requirements. To cope with this situation we decided to use a special development process which combines the good maintainability characteristics of UML with the advantages of Ada 95 of reliability aspects.

In cooperation with the Company Aonix, Karlsruhe, we defined a UML profile for distributed reactive systems and developed ACD (Architecture Component Development) templates for code generation to Ada 95 source code with the UML tool Software through Pictures (StP) 8.0[1].

ACD allows the user to define code generation templates for any implementation language using a script language. In theory it is possible to define own stereotypes and tagged values in StP, and to access these stereotypes by the code generator, and therefore almost any mapping between UML and the implementation language of choice can be defined. The details concerning the usage and implementation of the templates for the generation of Ada 95 source code can be found in [2]. The described UML profile and the associated templates developed togehter with Aonix in the Inspect 2 project are part of the latest release StP 8.3.

This report describes the UML profile and reviews our experiences with code generation from UML to Ada 95 based on StP.

2 The process control tool Inspect 2

In order to use Inspect 2 as a running example we will first of all give a brief description of its architecture (see Figure 1).

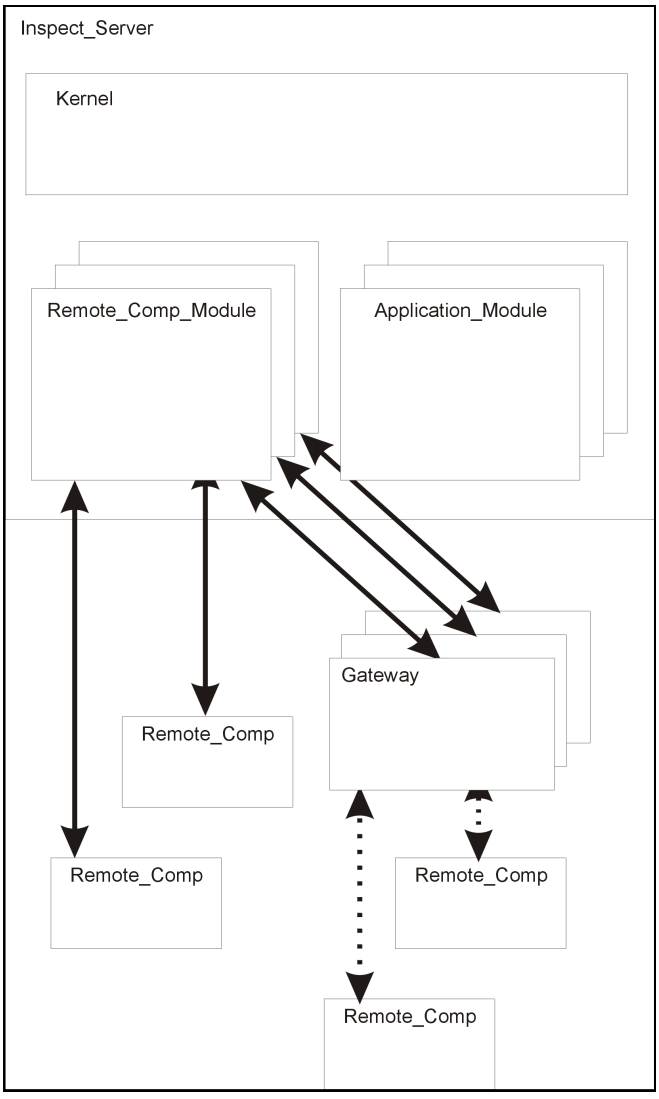

Figure 1 The structure of Inspect 2

Inspect 2 is a client server system with the Inspect server managing the collection, distribution and archiving of data.

[1] Parts of the UML model, describing the communications subsystem, have also been mapped to Java source code

Additionally, the server is able to schedule various analysis and control tasks. The clients, communicating with the server via TCP/IP, are responsible for the measurement of process values and for the communication with other systems, e.g. the process control system. Special clients (Gateways) allow the communication with non TCP/IP based systems. Clients also act as distributed/remote human machine interfaces.

Each client communicates with the server via an own associated task (Remote_Comp_Module) within the server, which maintains the communication and caters for a reliable data transfer. The application logic of Inspect (i.e. data processing) is defined in Application_Module tasks. All modules can be customized by configuration and are scheduled by the Inspect server.

At present instances of Inspect 2 are used in the field of industrial combustion control and sensor networks.

3 The UML profile

As we already motivated in [3], an advantage in the use of UML is its possibility of customization. Since UML is a complex family of languages (see [4]) with a large amount of concepts, it is generally convenient to restrict these to the minimum required by the application domain. Otherwise problems in the mapping of UML diagrams to Ada 95 code are induced. Due to the fact that there are no complete formal semantics for UML, we decided to define a UML profile and give denotational semantics to it by mapping its concepts to the concepts of Ada 95.

For better understanding we do not use the UML notation for profiles (see [5]), but we define the concepts of our profile informally and illustrate them using parts of our running example Inspect 2. The structure of the profile is given in Figure 2. It uses class diagrams and statechart diagrams because they are able to describe distributed reactive systems completely. The structural and functional view of the system to be described is modelled by class diagrams. The behavioral view is modelled by statechart diagrams.

In the following we will describe the concepts of our UML profile and illustrate them using our running example Inspect 2 (see Figure 3).

3.1 Packages

In UML, the concept of Packages is used in the Model Management Part for the grouping of model elements (refer to [5], pp. 3-17). Since in standard UML subsystems are behavioral units rather than structural units (refer to [5], pp. 3-21) we introduce the stereotype subsystem for packages. In this sense in our profile a subsystem is a package which groups model elements which belong to structural parts of the system.

In our profile only packages with the stereotype subsystem are allowed. Subsystems may contain subsystems and for simplicity the total system is modelled by a subsystem too. In Figure 3 an example for a package with stereotype subsystem is shown: the package Inspect_Server.

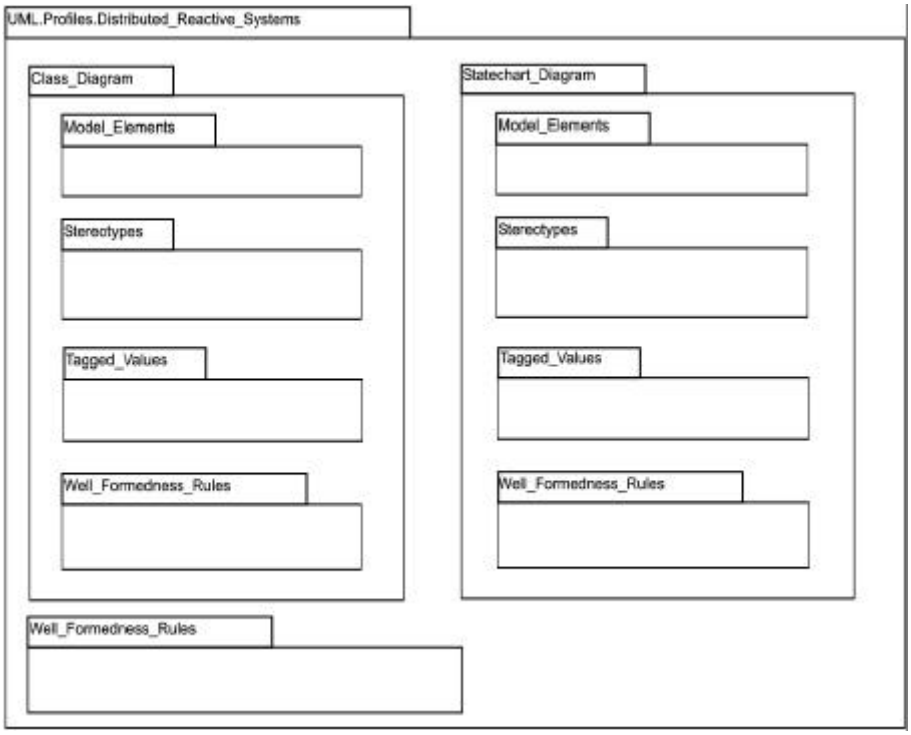

Figure 2 The structure of our profile

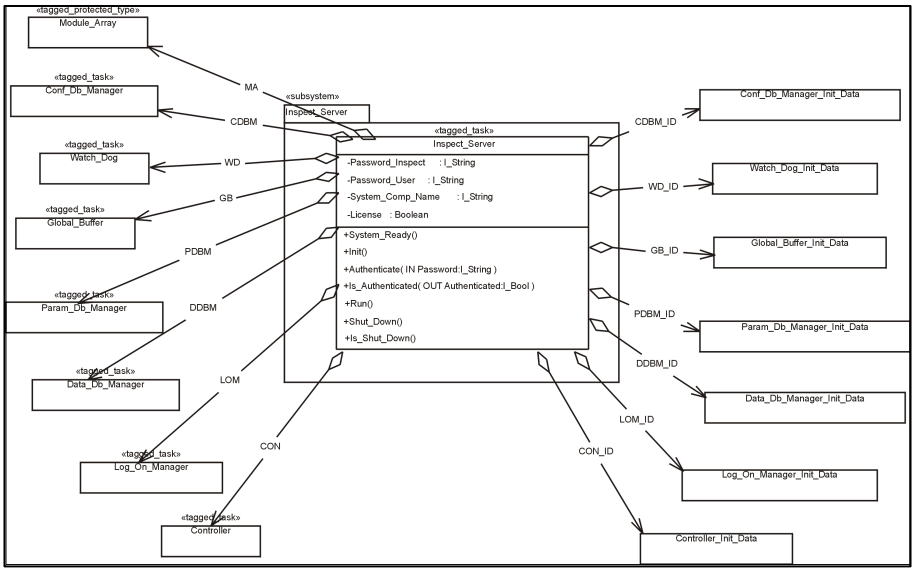

Figure 3 The structure of Inspect 2 as class diagram

3.2 Classes

In UML, Classes (refer to [5], pp. 3-34) follow the concept of abstract datatypes, i.e. they are container for typed attributes and operations over these. In our profile attributes and operations can have public or private visibility (as described in [5],p. 3-40 and 3-43). The protected visibility property is not allowed in our profile.

In order to model the concurrent behavior of distributed reactive systems we introduced the stereotypes Tagged_Protected_Type and Tagged_Task for classes:

A class with the stereotype Tagged_Protected_Type is a special case of a passive class. It is a storage class with a mechanism for the synchronization of concurrent access of multiple tasks.

The stereotype Tagged_Task defines an active class with a light-weight concurrent flow of control. In contrast to the standard passive classes our tagged_-task class has an own flow of control. To describe this flow of control, each Tagged_Task is associated with a statechart. For the connection of the class and the statechart diagram, the following well-formedness rules must hold:

- Each public method of the class must be used as an event in the associated statechart diagram and vice versa.
- Guard conditions in the statechart only refer to the attributes of the class diagram.
- The change of values of attributes of the class is modelled in the associated statechart diagram as actions.
- Public methods of the class are external events which allow communication between different classes.
- Private methods of the class allow local abstraction and the expression of a behavior which is hidden to the environment (other classes).
- Methods of classes are non blocking (i.e. execution is locally without any further entry call or accept statement). Otherwise they could cause deadlocks.

A further new introduced concept for classes with the stereotype Tagged_Protected_Type or Tagged_Task is the tagged value Timed_Call_Supported for public operations. These so-called timed operations have different semantics than normal public operations. The call of a normal public operation of a Tagged_Protected_Type or a Tagged_Task leads to the situation that the caller is blocked until the operation is finished. If in the case of a Tagged_Task the associated statechart has no outgoing transition in the current state, which is annotated with the corresponding event, the caller is blocked until the Tagged_Task reaches a state with this predicate. In the worst case deadlocks and livelocks can occur, even though the called method itself is non blocking. A simple and effective way of avoiding liveness problems is the usage of timed operations. In our profile they have the semantic that the caller of the operation is only blocked until either the operation call is accepted or the timeout of the operation is expired. All timed operations have an extra in-parameter for specifying the maximum waiting time and an extra out-parameter informing the caller if the call was accepted (and processed) or not. These two parameters are implicitly defined in the class. On the caller side both parameter values need to be attached.

In Figure 4 a fragment of the associated statechart diagram of the class Inspect_Server is shown.

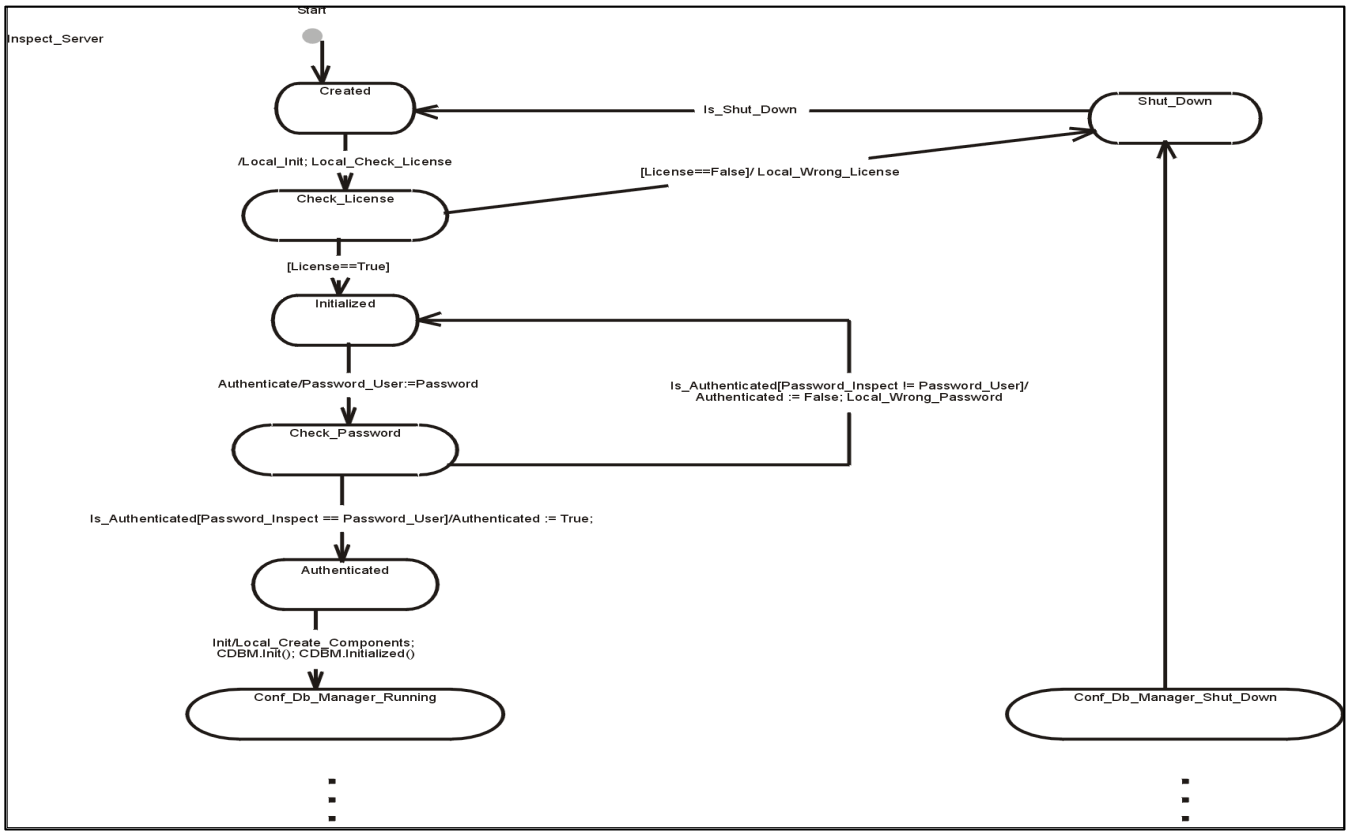

Figure 4 Fragment of the statechart of the class inspect_server

Between the states Initialized and Check_Password there are two interesting transitions. The transition from Initialized to Check_Password is triggered by the event Authenticate. After this event the attribute Password_User is set to Password.

Password is the value of the concrete parameter of the method Authenticate. The parameter is only defined in the class diagram (see Figure 4) and not in the statechart[2]. The transition from Check_Password to Initialized gives an example of a complex transition.

If the event Is_Authenticated occurs the guard condition [Password_Inspect!=Password_User] is evaluated and the transition takes place if the result is true (i.e. the password is wrong). In that case two actions are executed. The first action changes the classes attribute Authenticated to the value False. The second action is the execution of the local method Local_Wrong_Password. This is another interesting concept of our approach. The user may specify actions in the statechart which are not member of the associated class. These are mapped to locally defined Ada procedures in the body and represent an additional mechanism of local abstraction. The semantic of the local procedure is defined in the implementation and does not matter in the design. The use of this concept increases the clarity and understandability of the statechart diagram. It is important to avoid the definition of blocking local procedures in the code since this leads to deadlocks. Since local procedures are used as a concept of abstraction between the design and implementation level, we deliberately leave the avoidance of blocking in the responsibility of the developer. Thus our approach does not provide a method to ensure the non blocking definition of local procedures. We would therefore recommend not to use infinite loops and external operation calls. The latter can not always be considered, because the main purpose of local procedures is the use of operating system functions.

3.3 Associations

UML consists of a rich set of association types. In our profile only the following types of associations are necessary:

- generalization
- aggregation
- composition

For the use of the inheritance features, the generalization association (see [5], p. 3-79) is included in the profile. Additionally our profile includes the association type composition as described in [5] on page 3-74.

The most important association type in our profile is the aggregation association (see [5], p. 3-79). In contrast to standard UML the only restriction is the deprecation of association classes for the aggregation.

For the aggregation our profile contains the stereotype Owner. The so-called ownership aggregation models a close binding between the associated classes. One side of the association has the role of the owner and is responsible for the creation and destruction of the class on the other side.

3.4 Well-formedness rules

In order to avoid liveness problems we decided to use only hierarchic visibility between classes.

4 Mapping the concepts to Ada 95

4.1 Packages

UML Packages with the stereotype Subsystem are mapped to folders in the project-folder.

4.2 Classes

The classes of the profile are mapped to Ada 95 packages with the following content:

- a record with one component for each attribute of the class and one component for each aggregation of the class,
- a public procedure for each public operation of the class,
- a private procedure for each private operation of the class.

4.2 Tagged_Protected_Type

The classes with stereotype Tagged_Protected_Type are mapped to an Ada 95 package containing the following elements:

- a record with one component for each attribute of the class and one component for each aggregation of the class,
- a private protected type Class_Name_Pt with one entry for each class operation (with isomorphic signatures),
- a public procedure Operation_Name for each class operation with a signature isomorphic to the class operation (In the procedures body the corresponding entry of the protected type Class_Name_Pt is called.),

a public procedure Timed_Operation_Name for each class operation with tagged value Timed_Call_Supported, which has a signature isomorphic to the class concatenated with the additional parameters for timing. These are (in:Time:duration,out:timed_out:boolean) (In the body of the procedure the corresponding entry of the protected type Class_Name_Pt is called as a timed operation.).

4.3 Tagged_Task

The classes with stereotype Tagged_Task are mapped to an Ada 95 package which includes the following elements:

- a record with one component for each attribute of the class and one component for each aggregation of the class,

[2] by convention the parameters of classes event can be omitted if the name of the event is unique in the class

- a variable Current_State of an enumeration type Class_Name_Event_Type containing all states of the associated statechart diagram,
- a private task type Class_Name_Task with one entry for each event in the statechart diagram. The entries have a signature isomorphic to the corresponding class operation.
- a public procedure Operation_Name for each class operation with a signature isomorphic to the class operation (In the procedures body the corresponding entry of the task type Class_Name_Task is called.),
- a public procedure Timed_Operation_Name for each class operation with the tagged value Timed_Call_Supported which has a signature isomorphic to the class concatenated with the additional parameters for timing. These are (**in**:Time:duration,**out**:timed_out:boolean) (In the body of the procedure the corresponding entry of the task type Class_Name_Task is called as a timed operation.),
- a statechart implementation using select-statements depending on the value of the variable Current_State,
- a private procedure for each action in the statechart diagram with a signature isomorphic to the action.

4.4 Code examples

In the section above we described the mapping of the UML concepts to Ada 95. For a better impression to the reader we will give a detailed code example. First, we will show the Ada specification file of the Inspect_Server_Pkg, whose UML representation was discussed in section 3[3].

In the specification the package Inspect_Server_Pkg is introduced:

```
package Inspect_Server_Pkg is

type Inspect_Server is new
ACD_Runtime.Active_Base_Class.ActiveLimitedInstance
with private;

type Inspect_Server_Cptr is access all In-
spect_Server'class;

type Inspect_Server_Ptr is access all Inspect_Server;

...
```

It consists of the derived type Inspect_Server with a pointer Inspect_Server_Ptr and a class wide pointer Inspect_Server_Cptr. The type ACD_Runtime.Active_Base_Class.ActiveLimitedInstance is an empty tagged record type which represents the type active class in the profile and was adopted from the standard ACD code generation templates. We found it usefull, because it enables the user to define global properties of active classes, if necessary. In the private part the type is redefined.

[3] For better readability we reformatted the automatically generated file an removed irrelevant comments.

Next, the type Inspect_Server_Event_Type defines the set of events of the class Inspect_Server:

```
type Inspect_Server_Event_Type is (
      Is_Authenticated_Event
    , Is_Shut_Down_Event
    , System_Ready_Event
    , Shut_Down_Event
    , Init_Event
    , Authenticate_Event
    , Run_Event
    );
```

The correspondance between events, class operations and entries of the task are shown in the body. The next parts in the specification are the constructor and destructor operations:

```
--Constructor Operations-----------------------------------------

   procedure Initialize (Acc_This : Inspect_Server_Cptr);

   function Create return Inspect_Server_Ptr;

--Destructor Operations------------------------------------------

   procedure Finalize (Acc_This : in out  In-
   spect_Server);

   procedure Free (Acc_This : in out  In-
   spect_Server_Cptr);
```

Next, the class operations are specified. Since the mapping principle is the same for each operation, we show only two examples:

```
--Operations-----------------------------------------------------

   procedure System_Ready(Acc_This : access In-
   spect_Server);

   -- Timed_Call_Supported:

   procedure Timed_System_Ready(Acc_This : access
   Inspect_Server; Timeout : duration; Timed_Out : out
   Boolean);

   procedure Init(Acc_This : access Inspect_Server);

   -- Timed_Call_Supported:

   procedure Timed_Init(Acc_This : access In-
   spect_Server; Timeout : duration; Timed_Out : out
   Boolean);
```

The reader may have recognized that two versions of each operation are defined. This reflects that for these operations the tagged value Timed_Call_Supported is set in the UML model. The timed versions of the procedures have prefix Timed_ attached. In the private part of the specification, the static part of the associated statemachine of the active class Inspect_Server is defined:

```
private

package Inspect_Server_State_Machine is

--State Type----------------------------------------------------

   type State_Type is (

         ST_Authenticated

       , ST_Check_Password

       …

   );

 end Inspect_Server_State_Machine;
```

This is the type Inspect_Server_State_Machine which models the set of states of the statemachine. The transitions of the statemachine are defined in the body. Since each active class has an own thread of control a task type is necessary:

```
task type Inspect_Server_Task(Acc_This : access Inspect_Server'class) is

--State Machine Operations---------------------------------------

   entry Take_Is_Authenticated_Event(Authenticated :
   out Types.I_Bool);

   …

   entry Take_Run_Event;

end Inspect_Server_Task;
```

For each event of the statemachine the task type has a corresponding entry. To increase the readability of the code, the prefix Take_ is attached to the entries. The example above shows only two entries. As stated earlier, the type Inspect_Server is redefined in the private part:

```
type Inspect_Server is new
ACD_Runtime.Active_Base_Class.ActiveLimitedInstance
with

record

Tsk : Inspect_Server_task(Acc_This => Inspect_Server'access);

--Attributes----------------------------------------------------

Password_Inspect : Types.I_String;

…

 --Relations----------------------------------------------------

WD_Part : Watch_Dog_Pkg.Watch_Dog_Cptr;

MA_Part : Module_Array_Pkg.Module_Array_Cptr;

…

end record;

end Inspect_Server_Pkg;
```

The first component of the tagged record is of the type of the previously defined task type. This shows the encapsulation of the task in the tagged record which is the central concept of the active class implementation. The second component shown in the example is the attribute Password_Inspect. The other attributes are not shown. After that, two examples of relations to other classes are shown. The prefixes WD and MA are the names of the relations in the class diagram (refer to Figure 3). By convention each relation attribute is named by the relation name followed by _Part.

After this short overview of the Ada specification of the Inspect_Server_Pkg we will have a look to the associated Ada body file:

```
package body Inspect_Server_pkg is

   package body Inspect_Server_State_Machine is

   end Inspect_Server_State_Machine;
```

The code fragment above shows the implementation of the statemachine type. Since it is an enumerated type the most information is still defined in the Ada specification.

More interesting is the implementation of the Inspect_Server_Task:

```
task body Inspect_Server_Task is

   This : Inspect_Server'class renames Acc_This.all;

   --State Machine-------------------------------------------------

      current-
      State:Inspect_Server_State_Machine.State_Type
      := Inspect_Server_State_Machine.ST_Start;

         use Inspect_Server_State_Machine;
```

For readability reasons the access to self (Acc_This.all) is renamed by the keyword This. After that, the variable currentState is declared and set to the initial state.

Next, the local procedures are defined. As shown in Figure 4, local procedures implement the actions of the UML statecharts which model the behavior of active classes. For example we show a fragment of the implementation of the local procedure Local_Create_Components, which creates all components of the active class Inspect_Server:

```
procedure Local_Create_Components is

begin

--Create Module_Array

This.MA_Part := Module_Array_Pkg.Create.all'access;

--Create Watch_Dog in the Pkg

Watch_Dog_Pkg.The_Watch_Dog_Ptr :=
Watch_Dog_Pkg.Create.all'access;

This.WD_Part := Watch_Dog_Pkg.The_Watch_Dog_Ptr;
…

end Local_Create_Components;
```

The following code fragment shows the implementation of the statemachine. For a good understanding a comparison of the implementation with the UML statechart diagram (refer to Figure 4) is usefull. The core of the implementa-

tion is a case statement over the variable currentState enclosed in a loop. The case statement branches (via when-statements) over each possible state of the statemachine. Since the variable currentState enumerates over all states of the statemachine it is an invariant of the implementation so in the focus of the loop statement exactly one when-expression evaluates to true. This implies that the order of the when-statements is inessential. We only show the states Initialized and Check_Password in our example because they include all relevant concepts of our statemachine implementation:

```
loop
  case currentState is
    …
    when Inspect_Server_State_Machine.
    ST_Check_Password =>
    -- Activity:
    null;-- user defined code to be added here
    select
      when ( This.Password_Inspect =
      This.Password_User )
      or ( This.Password_Inspect /=
      This.Password_User ) =>
      accept
      Take_Is_Authenticated_Event(Authenticated :
      out Types.I_Bool) do
          if This.Password_Inspect =
          This.Password_User then
          Authenticated := True;
          currentState := ST_Authenticated;
          elsif This.Password_Inspect /=
          This.Password_User then
          Authenticated := False; Lo-
          cal_Wrong_Password;
          currentState := ST_Initialized;
          end if;
          end Take_Is_Authenticated_Event;
      end select;
    when Inspect_Server_State_Machine.
    ST_Initialized =>
    -- Activity:
    null; -- user defined code to be added here
    select
          when ( TRUE ) =>
          accept Take_Authenticate_Event(Password
    : in Types.I_String) do
              if TRUE then
                  This.Password_User:=Password;
                  currentState := ST_Check_Password;
              end if;
          end Take_Authenticate_Event;
    end select;
    …
  end case; -- end state case
  end loop;
```

Assume the variable currentState has the value ST_Check_Password. Thus Inspect_Server_State_Machine.ST_Check_Password evaluates to true and the other when-expression evaluates to false. Then the code after the first when-statement is executed. Because there is no entry action defined in the state Check_Password, there is a null-statement after the comment *–Activity*. Next, the implementation of the outgoing transitions of the state Check_Password is shown. Both are triggered by the event Is_Authenticated (for both guard conditions Take_Is_Authenticated_Event is accepted). When this event occurs (the operation Is_Authenticated is called) the guards are evaluated in the if-elsif-combination. If the if-branch evaluates to true, authenticated ist set to true and the next currentState is set to ST_Authenticated. Otherwise authenticated is set to false and the next currentState is set to Initialized.

Assume the current state is Initialized, there is one possible outgoing transition. In contrast to the transitions above this one has no guard condition. In the select-statement the expression **when** (TRUE) is used which is constantly true.

Whenever the event Take_Authenticate_Event occurs (the operation Authenticate is called), the guard **if** TRUE evaluates to true and the action code is executed. This means, that the attribute This.Password_User is set and the next value of the variable currentState is set to ST_Check_Password.

5 Conclusion

The above described UML profile for distributed reactive systems was developed at the Forschungszentrum Karlsruhe for internal use. We used it for the development of the process control tool Inspect 2 which we used as running example in this paper.

A measurement of the systems complexity[4] did not take place yet, but we can provide some estimates in order to get an idea of the projects effort. The UML model is decomposed into 19 subsystems with 31 class diagrams and 13

[4] Complexity measurement of software is an interesting but elaborate scheme, in particular in the domain of the object-oriented analysis and development. We use the notion of complexity in its natural sense, since the use of a formal complexity measure is no benefit for the reader for the understanding of our presented approach.

statechart diagrams. These are mapped to nearly 150 packages of Ada 95 source code.

Since the project was the first using the presented approach there was an overhead in the early project phases. This was caused by the need of defining the UML profile and the associated mapping to Ada 95. Another cost factor was the analysis of the existing design tools, their installation and test. Additionally the customization of the Ada 95 templates for code generation increased the start up costs.

After finishing the projects foundations we have to point out that most of the effort of the implementation phase was drawn in the earlier design phase. Even some effort of the documentation was done previously.

During the development of the system there was one very important philosophy in the developers mind: whenever a change could be made in the design instead of the implementation it was made in the design. This strong focus on the design phase implied a good understanding of the system to be developed (which reduces logical faults) and a very tight relationship between the design and implementation model.

The common coding errors could be minimized. Therefore the implementation phase was very short. Additionally the testing phase[5] was decreased extremely. This fact was induced by the excellent debug capabilities of the approach. Whenever a logical failure occurred during the testing phase it was easy to find the error because its area could be determined using the design model and specifically the statechart diagram.

An analysis of the errors found in the testing phase leads to the following error classes:

1) about 70% of the errors found were logical errors and therefore design errors,

2) about 20% of the errors found were runtime errors, caused by forgotten initializations in the manually implemented code,

3) about 10% of the errors found were memory leaks.

The error classes two and three could be reduced in future projects to a minimum by refining the code generation properties. Class two errors will be reduced automatically by increasing the amount of automatically generated code (currently about 80 percent). The reduction of errors of class three can be done by the integration of constructor and destructor method implementations in the code generation templates. In the present templates only the standard constructors and destructors are generated automatically. Since the needed information is statically available in most cases improvements are possible for less effort. Only the first class errors cannot be addressed by increased use of automated code generation and require deeper further investigations. To reduce these errors the combination of the chosen approach with design level simulations and formal methods appears to be helpful. Another very promising circumstance is the fact that the work on the UML standard will go on. Influenced by the precise UML group (see e.g. [6]) the Object Management Group (OMG) plans the new standard UML 2.0 (see e.g. [7]). This will come with complete meta-modelling semantics. Concluding it can be said that the combination of UML as modelling language and Ada 95 as implementation language in conjunction with the ACD code generation leads to higher software quality. Our concrete example, Inspect 2, has been successfully running in industrial applications all over the world in round-the-clock operation for six month.

The future plans of our research group can be divided into different fields. One of these is the further development of Inspect 2 with the creation of new application modules for different application domains.

Another field ist the advancement of the present development approach. To increase the debugging capabilities, one actual task is the definition of a watch-dog-concept which allows the tracing of transitions in a log file. Therefore the code generation templates are modified.

A Further, long term activity, is the development of formal methods for the quality assurance of UML design level models. Actually, the investigation of the theoretical foundation of that work is in progress. That needs the definition of formal meta-modeling semantics of the UML profile. The main objectives of our formal methods are the proof of liveliness and timing properties on the design level.

Another very promising subject in the field of the UML-based analysis and development is test automation. For the definition of test cases sequence diagrams are well suited. The definition of templates for the ACD code generation seems to be a facile venture. Since our resources are restricted we can not address this subject in our research group.

[5] Since our approach has a strong focus on the design phase, we use the term testing phase in an informal way.

6 Acknowledgement

The developement of Inspect 2 was supported by the "HGF-Strategiefond"-projects "NOX" and "ELMINA".

7 References

[1] C. Fouda, H.B. Keller, M. Kersten, J. Matthes, S. Zipser and T. Krakau (2002), *Systemhandbuch Inspect 2.1*, technical report, Forschungszentrum Karlsruhe.

[2] W.D. Heker (2002), *Generating Ada95 with StP/UML*, technical report, Aonix GmbH, available via stp-support@aonix.de or StP 8.3.

[3] M. Kersten and H.B. Keller (2001), Die Problematik der Abbildung von UML-Modellen auf Konstrukte der Programmiersprache Ada 95, in: P. Dencker et.al., editors, *Ada und Softwarequalität: Ada Deutschland Tagung 2001*, München, Ottobrunn, Shaker Verlag.

[4] S. Brodsky, T. Clark, S. Cook, A. Evans, and S. Kent (2000), *Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modelling Approach*, technical report, The precise UML Group, http://www.cs.york.ac.uk/puml/mmf.

[5] OMG (1999), *OMG: Unified Modeling Language Specification*, technical report, OMG, http://www.omg.org.

[6] Clark et.al. (2001), *Initial Submission to OMG RFP's : ad/00-01-01(UML 2.0 Infrastructure), ad/00-09-03(UML 2.0 OCL)*, technical report, Precise UML Group, http://www.cs.york.ac.uk/puml/papers/uml2submission.pdf.

[7] OMG (2001), *Request for Proposal: OMG Document: ad/00-09-01 UML 2.0 Infrastructure*, technical report, OMG, http://www.omg.org.

Dr. Hubert B. Keller is head of the research groups "Innovative Process Control" and "Intelligent Sensor Systems" at the Institute for Applied Computer Science, Research Centre Karlsruhe.
Research interests: real time systems, software engineering, machine intelligence, intelligent sensor and process control systems.

Dipl.-Inf. Michael Kersten was part of the research group "Innovative Process Control" at the Institute for Applied Computer Science, Research Centre Karlsruhe from 1999 to 2002. Now he is working at the department of computer science at the Carl von Ossietzky university Oldenburg.
Research interests: design and analysis of object oriented embedded realtime systems.

Dipl.-Ing. Jörg Matthes is part of the research group "Intelligent Sensor Systems" at the Institute for Applied Computer Science, Research Centre Karlsruhe since 2000.
Research interests: data analysis in sensor networks.

Dipl.-Ing. Christian Fouda Manga is part of the research group "Intelligent Sensor Systems" at the Institute for Applied Computer Science, Research Centre Karlsruhe since 2000.
Research interests: safe and secure interlinking of fieldbuses to Ethernet.

Dipl.-Ing. Stephan Zipser is part of the research group "Innovative Process Control" at the Institute for Applied Computer Science, Research Centre Karlsruhe since 2000.
Research interests: optimization of industrial combustion processes.

Email: keller|matthes|fouda|zipser@iai.fzk.de
Michael.Kersten@informatik.uni-oldenburg.de