DOI 10.1007/s00165-003-0027-0
BCS © 2004
Formal Aspects of Computing (2004) 16: 80–94

Formal Aspects of Computing

Integration of UML and VHDL-AMS for analogue system modelling

C. T. Carr, T. M. McGinnity and L. J. McDaid

Intelligent Systems Engineering Laboratory, University of Ulster, Northern Ireland, UK

Abstract. This paper details a new object-oriented methodology that permits a unified modelling language (UML) behavioural representation of analogue circuits at system level. The proposed method demonstrates a novel approach to the problem of behavioural representation of an analogue topology, by constructing a consistent set of rules for automated mapping of the UML model to a VHDL-AMS specification. The VHDL-AMS specification enables behavioural simulation of the UML model and the methodology is validated using an analogue subsystem level application.

Keywords: Analogue, Automated mapping, Circuits, UML, VHDL-AMS

1. Introduction

The current state of the art in electronic and computer systems design prohibits complete high level modelling of a comprehensive system involving software and both analogue and digital hardware. Typically, high level systems contain several subsystems as illustrated in Fig. 1. This system consists of a software partition, a digital partition and an analogue partition, typically with complex interdependencies. The software partition can be successfully modelled using existing unified modelling language (UML) techniques and thereafter implemented using a high level language [Mel99]. The digital partition can be successfully mapped from UML to a language such as VHDL [Axe01, Mcu99, Bad98]. The design of digital circuits and associated systems is well supported by sophisticated computer aided design (CAD) tools. However, analogue CAD tools are significantly less well developed. Many of the tools proposed by research into automated analogue design have never been prototyped at full system level, and certainly not commercially in industry. Most of these systems require a high level of knowledge of the analogue domain in order to use them efficiently, and this factor has led to under-utilisation of such systems. However, more recently VHDL-AMS (which is an extension to VHDL [Vhd99]), has emerged as a popular language for modelling analogue subsystems at a high level. An alternative is Verilog-AMS, an extension of Verilog. However VHDL-AMS or Verilog-AMS do not permit complete system modelling, i.e. they cannot model software elements. Currently therefore the analogue partition of Fig. 1 cannot be modelled at the same level of abstraction as the software or digital hardware partitions.

High level system modelling would benefit from a tool that could represent all the components of an electronic computational system and model their interactions. It has already been demonstrated that UML can be used to represent both digital and software subsystems and provide a mapping to the appropriate language for implementation [Mcu99, Jig00]. If a similar methodology existed to map analogue subsystems from UML to either

Correspondence and offprint requests to: C. T. Carr, Intelligent Systems Engineering Laboratory, University of Ulster, Magee Campus, Northland Rd, Derry, Northern Ireland, BT48 7JL, UK. Email: c.carr@ulst.ac.uk

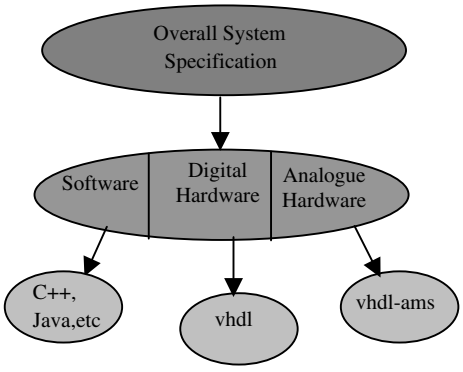

Fig. 1. System partitions

VHDL-AMS or Verilog-A, then an overall description of mixed signal systems could be realised. Also if UML could represent analogue systems, then it is possible that this high level model would be capable of representing a tight integration between hardware and software in embedded systems.

The authors will illustrate in this paper how UML can effectively represent a high level analogue topology and will further illustrate a procedure for mapping from UML to VHDL-AMS. VHDL-AMS has been selected instead of Verilog-AMS because VHDL-AMS is consistent with VHDL, which has already been used with UML in system modelling [Mcu99]. Previous work [Axe01] illustrated the principle of using UML as a modelling tool for complete systems, including analogue systems. However, this work presented only a UML representation and did not provide a technique for mapping to support simulation and analysis. There is clearly a requirement for a self-contained, high level system modelling technique to accommodate all three partitions in a unified representation, provide simulation capability and provide automatic code generation in the appropriate language. It is therefore plausible to attempt to represent an analogue subsystem in UML, with the objective of automatically generating VHDL-AMS from the UML model. This is the missing link for complete system modelling using UML. The work in this paper details an approach for modelling the analogue subsystem in UML and automatically generates VHDL-AMS from the models. This approach is currently being extended to mixed signal capabilities, which will realise the requirement for a unified representation for system modelling.

The remainder of this paper is organised as follows. Section 2 presents an overview of VHDL-AMS and discusses its relevance to the analogue domain. Section 3 gives an overview of the current status of UML. Section 4 describes the proposed methodology for modelling analogue system in UML; Section 5 applies this methodology to a simple example and shows how a UML model can be mapped to VHDL-AMS, and illustrates simulation results to justify the approach. Section 6 concludes the paper and discusses future extensions to the work.

2. VHDL-AMS

2.1. Hardware description languages

VHDL-AMS [Vhd99, Chr99] has emerged as the most popular hardware descriptive language for analogue behavioural modelling and has been accepted as an IEEE standard VHDL 1076.1-1 1999 [Vhd99]. These hardware descriptive languages (HDL) can be divided into digital, analogue and mixed signal HDLs, depending on the available language constructs. VHDL [Vhd02] is a digital HDL and is based on event driven techniques and a discrete model of time. Verilog [Ver95] is also a hardware description language used to design and document electronic systems. VHDL and Verilog both support the modelling of digital hardware at any abstraction level. Verilog-AMS [Vams03] is also a standard in mixed signal modelling and was standardised by Accellera for mixed signal behaviour in Verilog. Both VHDL-AMS and Verilog-AMS accommodate analogue and mixed signal modelling capabilities.

The main difference between these two HDLs is that VHDL-AMS is capable of automatically converting transient models (constructed of non-linear differential equations) into small-signal ac models while Verilog-AMS requires separately designed and implemented small-signal ac models to provide better compatibility with SPICE-based simulators. For simple models, this separate implementation of small-signal ac models is easily

accomplished. However for complex models, the implementation of separate ac models becomes very difficult. Such ac models are particularly difficult to design for power bipolar models where the internal operating state depends upon the presence of charge injected. Thus, VHDL-AMS is preferred over Verilog-AMS, particularly for more complex models.

2.2. Capabilities of VHDL-AMS

VHDL-AMS supports the description of a system of differential and algebraic equations (DAEs) where the solution varies continuously with time and, in addition, supports both structural composition and behavioural descriptions of analogue systems. VDHL-AMS can be used to model mixed signal systems, which means both event driven techniques and differential/algebraic equations are supported. Systems to be modelled using VHDL-AMS are lumped systems, that can be described by differential and algebraic equations; the solution of the equations describing the behaviour of the system may include discontinuities. Interactions between the discrete part of a model and its continuous part are supported in an adaptable and efficient way.

2.3. Modelling analogue systems

An analogue system in VHDL-AMS typically consists of many analogue components, which are described using an entity/architecture structure. The model consists of an entity and one or more architectures. The entity specifies the interface of the model to the outside world, including the description of the ports of the model, i.e. the points that are connected to other models and the definition of the generic parameters. The architecture contains the implementation of the model and may be coded using a structural style of description, a behavioural style, or a style combining both structural and behavioural elements. A structural description is a netlist representing a hierarchical decomposition of the model into appropriately connected instances of other models. A behavioural description consists of concurrent statements to describe event-driven behaviour and simultaneous statements to describe continuous behaviour. Concurrent statements include the concurrent signal assignment for data flow modelling and the process statement for more general event driven modelling. In VHDL-AMS, modelling can be either top–down or bottom–up. However, the work in this paper focuses on a top–down behavioural description for analogue subsystems, where the connectivity of the model is subsequently defined in a testbench. However subsequent work will focus on bottom–up modelling using the structural method which makes use of library models. The analogue aspects of a mixed signal system are more difficult to model than the digital aspects due to the continuous nature of analogue signal behaviour. Therefore there it necessary to be able to describe continuous systems in the time and frequency domain. The following are key extensions to VHDL to allow modelling of mixed-signal systems.

- Support for *interface quantities* (for modeling of signal flows) as abstract objects representing analogue waveforms and *terminals* (as connection points) for network connectivity.
- Support for *simultaneous equations* for description of explicit and implicit differential equations and *quantity* for the capture of physical variables to model analogue behaviour.
- Support for *break* statements to model discontinuities in analogue waveforms.

Architectures with quantity declarations and simultaneous equations lead to descriptions of continuous or discontinuous non-linear dynamic systems to allow true analogue or mixed-signal descriptions.

3. UML

3.1. Choice of UML as the system modelling tool

UML [Rum99, Bez99] is a general-purpose modelling language that is used to design a system graphically and textually and allows precise modelling of systems using different views and representations. It also enables automatic source code generation by mapping the UML models to high level languages, such as Java or C++. The language has emerged by common agreement among much of the computing community and was designed to include the concepts of the leading methods, so that it can be used as a universal modelling language. It was intended to supersede the models of OMT, Booch, and Objectory [boo99, Boo97]. The notation used in OMT, Booch, and Objectory, and other leading methods is similar to the UML notation. The UML consolidates a

set of core modelling concepts that are generally accepted across many current methods and modelling tools. Therefore, as a distillation of previous best modelling practices, and as the most widely used modelling language in the computing industry, UML is the obvious choice for the current work.

3.2. Capabilities of UML

The UML captures information about the static structure and dynamic behavior of a system. A system is modelled as a collection of discrete objects that interact to perform work that benefits an outside user. The static structure defines the kinds of objects important to a system and its implementation, as well as the relationships among objects. The dynamic behaviour defines the history of objects over time and the communications among objects to accomplish tasks; therefore UML allows a system to be modelled from several viewpoints, which allows it to be understood for many different purposes. The UML also contains organisational constructs for arranging models into packages, that allow software teams to partition large systems into workable pieces, to understand and control dependencies among the packages, and to manage the description of models in a complex development environment. UML is a very general notation, used primarily for software development but also in many fields and for many purposes. UML serves only as the abstract description of a system – it must be complemented using other languages. For most software systems these languages are usually traditional low-level languages, such as C/C++ or Java. UML has already been extended to cope with modelling real time systems [Dou99a, Dou99b] and work by McUmber [Mcu99] examines how object orientated modelling (specifically UML), can be used for embedded systems by developing a framework for generating VHDL specifications from a subset of UML models. The main objective was to enable developers to continue to use widely accepted development techniques and tools, both in terms of a modelling language (UML) and a target specification language (VHDL). McUmber has developed a set of formalisation rules that enable automated techniques to generate specifications from the individual UML notations. This latter work highlights how the digital domain is represented in UML, thus making is feasible to attempt extending the application of UML analogue domain.

3.3. Notation employed

UML is an extensive language with an emphasis on graphical notation, and contains a large number of concepts and diagrams to convey different aspects of a system. Many reviews of the language are available [Rum99]. Therefore, the following description is restricted to those aspects of the language pertinent to the concepts used in the remainder of this paper.

- Class or object model diagrams

Class diagrams describe the static structure of a system. The fundamental concept of object-oriented methods is the *object*. An object or class consists of a set of *attribute*s, or variables, which describe its internal state, and a set of *operations* that the object can perform, e.g. to update or query the state, or perform calculations. An object is an *instance* of a *clas*s, which can be thought of as a set of objects with similar properties, i.e. the same attribute and operation names, and the information about the properties is normally provided at the class level. At a given time, there may be several instances of a given class, and the instances may have different attribute values. Classes are shown as boxes with the name inside it. Alternatively, the box may be divided into three compartments, containing the class name, the attributes, and the operations, respectively.

- Relations or associations

It is possible to define *links* between objects i.e. classes, that allow objects communicate to each other. The links between objects are instances of *relations* between the corresponding classes. The relations can be a general *association*, shown by a line between the classes or aassociations represent static relationships between classes. Association names can be placed above, on, or below the association line. The relationship can be characterized by *aggregations*, indicating that an object is part of another object shown by a line with a diamond attached to the end connected to the 'owning' class. Finally the relations can be *generalizations*, which describe the relationship between classes similar to the subset relationship between sets; generalization are shown by a hollow arrow head attached to the end near the more general class. The class that is a specialization can be thought of as inheriting the features of the more general class, thus allowing information to be reused. Roles represent the way classes see each other and it is possible to add a *role* name to the association, indicating by what name an object refers

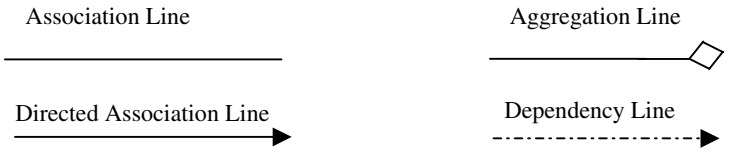

Fig. 2. Relations between classes

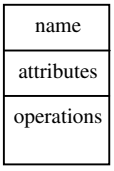

Fig. 3. A single class: used to represent one or more instances of the same component

to the other object in the relation i.e. the attribute x of the class related by the role r is referred by the name r.x. Figure 2 shows some of the different types of relations between classes.

- Multiplicity

For each class involved in a relationship there will always be a multiplicity for it, indicating the number of instances of one class linked to *one* instance of the other class. For example, one company will have one or more employees, but each employee works for one company only. Also a class can have many instances; it will have the same attributes and operations and can be used many times in the same system.

4. Methodology for modelling analogue systems in UML

4.1. Criteria for UML representation

In order to model an analogue subsystem, class diagrams were selected because they have the potential to represent the types of objects (i.e. components) found in such systems and the nature of the relationships that exists between components. Each component in the system has its own function in relation to the complete system; object classes can be used to represent different components in a system. Therefore, each object class in a system has different attributes and is capable of performing different operations. A collection of object classes are connected together to make up a class diagram. Consequently, class diagrams can be equated to analogue subsystems, due to the presence of many component classes within a system. If each component or analogue building block is treated as a class within a subsystem, then class diagrams within UML may be considered as adequate to represent a multi-component analogue subsystem and the associated connectivity.

The diagram in Fig. 3 shows a single typical class in UML with three sections. The first holds the name of the class, the second holds the attributes and the operations are described in the third section. In this work, the name of the analogue component is placed in the **name** section, the inputs and output ports and generic values are described in the **attributes** section and finally the behavioural description, represented as a transfer function, is placed in the **operations** section. This representation is limited to behavioural modelling of components, which can be modelled using mathematical relationships.

4.2. Mapping rules between UML and VHDL-AMS

Having selected which UML diagrams are used to model analogue system in UML, a comparison is now made between the constructs used in VHDL-AMS and the UML class. This is done to facilitate a mapping between the two representations. The graphical comparisons between class and the entity/architecture structures are shown in Fig. 4 and discussed in the points below.

- Structure: The structure of the UML class can be equated with the entity/architecture structure in VHDL-AMS and the name of the class corresponds to the name of the entity in VHDL-AMS.

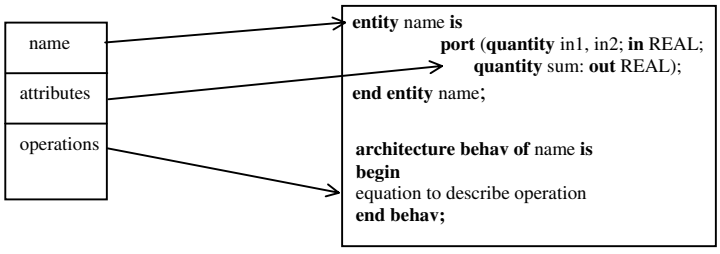

Fig. 4. Mapping from a UML class to an entity/architecture structure in VHDL-AMS

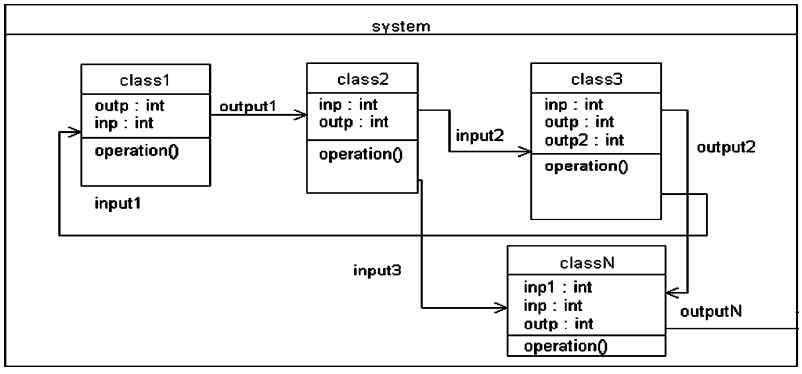

Fig. 5. A general system in UML with global inputs and outputs

- Attributes: The attributes section of the class (i.e. the input and output ports and generic values) are the same values that are entered in the port section in the VHDL-AMS entity structure.
- Operations: The operations section of the class, where the mathematical transfer function of the component is entered, holds the same information as the architecture section of the VHDL-AMS.
- Testbench: A testbench in VHDL-AMS is used to model test stimuli, in order to assess system performance and functionality. Subsequently local terminals of each component are mapped to new global terminal values so that the physical quantities being carried by the terminals define the connectivity of the overall system. The information used in the system class in UML can be equated with the information entered in the testbench in VHDL-AMS. In UML the overall system is defined in an overall class called 'system', as shown in Fig. 5 and the connections between classes show the global inputs and output ports of the system. These ports define the overall system connectivity, and are mapped from the local inputs/outputs of each component in the system. A new attribute is created in the system class for each component in the system and the local port and generic values are mapped to the global values in the system attribute for each component (an example of this is shown later in Fig. 9). The testbench is used to define system connectivity and test stimuli in this work; however a testbench is normally used solely as test stimuli and a structural architecture is used for connectivity. For simplicity the testbench is used for both in this work.
- Multiplicity: If the same component is instanced more than once in the same system then this must be accommodated for in the system attribute and multiple port and generic values need to be mapped according to the number of times the component is used.

4.3. Adapting UML for analogue modelling

The typical notation used to describe objects in a software system is not sufficient for modelling mixed signal systems and UML as a standard is not capable of representing the behaviour of analogue systems. Consequently to generate VHDL-AMS code, additional notation is required. It was therefore necessary for the authors to define a comprehensive set of keywords for UML, which will represent analogue subsystems at a high level of abstraction. The keywords are derived such that a general framework can be constructed.

Table 1. List of keywords

Keyword	Use
In(1... N)	Each analogue component has input and output ports and generic values. The keywords are used to differentiate between different types of entity values
Out(1... N)	
Generic(1... N)	
Rq	Rq and Et denotes real quantities and electrical terminals respectively, these define the types of terminal which exist in analogue components with respect to VHDL-AMS representation
Et	
Quantity	Defines the direction of current flow between terminals of a component
Across	
Through	
Equation	Refers to the behavioural representation of the component using a transfer function
Genericvalues=	Used for overall system specification
Noofinputs=	
Noofoutputs=	
Generic_map	Used to define mappings between local and global values of components
Port_map	
Multiplicity	Refers to the number of instances of any class in a system

- Keywords and how they are used in UML:

An analogue system is comprised of many interconnected components where each component has inputs, outputs, generic values, constants and a specific function within the complete system. The specific function of the component is equivalent to class 'operation' and can be described using a transfer function, i.e. equation. This generic composition of any component, has to be represented in UML. The following keywords have been defined to facilitate, and to enable a mapping to VHDL-AMS (Table 1).

- How each keyword is used in UML

For the purpose of this work, the Rhapsody tool [Rap97] was used; other UML tools are equally applicable.

1. In(1... N): The inputs to components are defined using the word 'in' followed by an integer from 1 to N, to allow representation of multiple inputs. The keyword "in" is only used if the port is of type "Rq", as electrical terminals do not have direction. Each class as shown in Fig. 2 can be selected, and a new attributed added for each input as shown in Fig. 6. The name of the input is entered in the name field. The keyword 'in' is placed in the description field to differentiate between the different types of attributes.

2. Out(1... N): The outputs have a predefined keyword 'out' followed by an integer from 1 to N, which again allows N outputs to be defined. The outputs are also entered as a new attribute, each time one is declared. Outputs are entered in exactly the same way as the inputs except that the word 'out' is entered in the description field (Fig. 6) to specify that it is an output. Again the keyword 'Out' is only required for ports of type "Rq" as electrical terminals (Et) have no direction.

3. Generic(1... N): The generic values (variables in the equation) are also entered in the attributes section of the class, and as before a new attribute is required for each new generic value. The name of the generic value is entered in the name section and in the description field in the dialog box in Fig. 6. It is specified as being a generic value by entering the keyword 'generic' followed by an integer.

4. Et and Rq: The keywords et and rq represent electrical terminals and real quantity respectively. These are the two types of ports that are used to represent the connectivity of electrical subsystems. When real quantities are used the direction is also specified using the keywords 'In' and 'Out'.

5. Quantity, Across and Through: These are the keywords reserved for defining an electrical terminal. ACROSS (e.g. voltage) and THROUGH (e.g. current) quantities are defined between terminals. Therefore, when entering information about electrical terminals these quantities must be specified. This information can be entered in the description box shown in Fig. 7.

6. Equation: Once all the information about the ports has been entered the operation of the component must be defined. This information is entered in terms of a mathematical equation. The equations are entered in the **operations** section of the features of each class, as shown in Fig. 7. The keyword 'equation' is used to indicate that a behavioural description of the component as a mathematical function has been entered. The mathematical function is entered in a system compatible with the standard VHDL-AMS mathematical library.

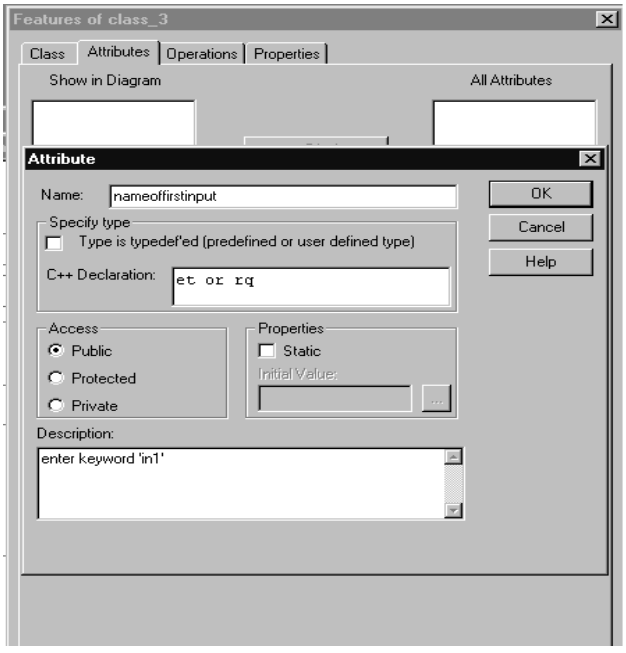

Fig. 6. A new attribute is created

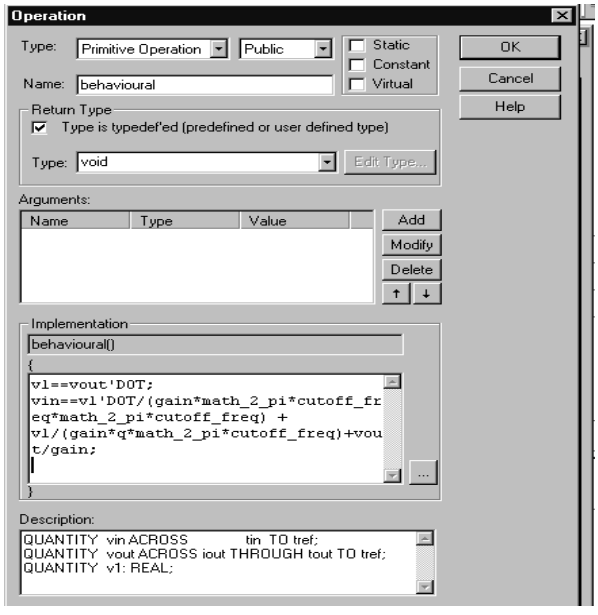

Fig. 7. Behavioural description of component

7. Genericvalues=, Noofinputs=, Noofoutputs=: These keywords are used to give an overview of the class and are entered in the overall class description field as shown in Fig. 8.

8. Generic_map and Port_map: The generic and port values entered in each class for each individual component are local to the component in order to describe its behaviour. When these components are connected together, as part of an analogue subsystem, the port names need to be globally defined within that system and the generic values need to be set a specific value; the **system class** handles this information. A new attribute is created for each class in the system, with the same name as the local class. This also contains the global

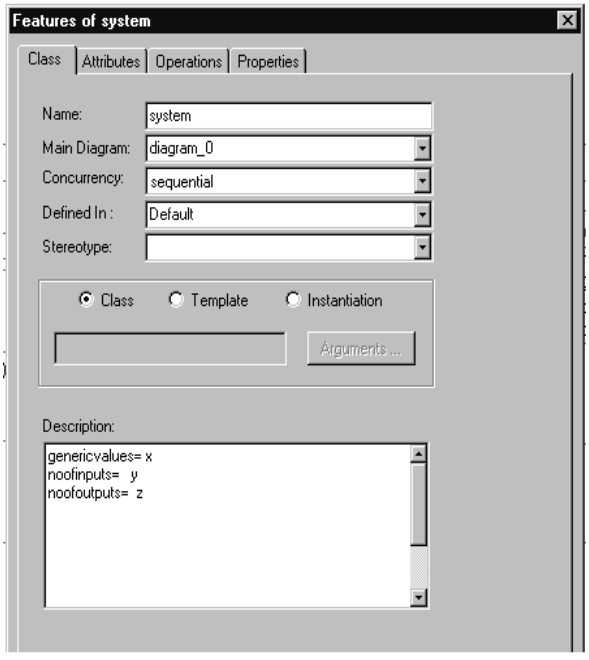

Fig. 8. Overview of the system

information about each class. Each new attribute contains the generic values and port mappings entered in the description field following the keywords generic_map and port_map respectively. This is shown in Fig. 9.

9. Multiplicity: Multiplicity refers to the number of instances of any class in one system, Moreover, systems could contain a number of identical components with different parameters and thus the class's behaviour only needs to be modelled once in the local class. However, in the system class the different parameters need to be entered for each instance of the class. This information is entered in the system class in the description field of each attribute, for each component as shown in Fig. 9.

4.4. Mapping methodology

In order to generate VHDL-AMS code from UML models, a set of rules for modelling analogue components in UML was defined. These rules were devised so that existing constructs in UML, specifically class diagrams, are utilised. Initally C++ code is automatically produced from the UML diagrams. For each class modelled in UML there is a '.cpp' and a '.h' file generated. The methodology adopted is to use these files to collect information about the system that is modelled in UML. The following algorithm in Fig. 10 shows how this was achieved.

- The first step in the algorithm is to read in the system file, which is generated from the system class in the UML representation. This file contains details about the system such as how many classes are in the system, the overall structure of the system and the names of the ports to each class. This information is obtained by searching the system file using the predefined keywords previously discussed.
- Information obtained from the system file is saved so that it can be mapped to the testbench file in VHDL-AMS and also to determine the number of entity/architecture structures required in VHDL-AMS.
- The next step involves reading in a template class file, which can be used as a model file to search for details on each class. The template file has general names, which are replaced with class names for each component.
- A new file is created for each class in the system; a search is then made for all the available information on each component.
- The number of classes found in the system is recorded from the system class file. A loop is entered at this point and each class file is the system is searched. The information obtained about the classes such as port and operation details are recorded, and repeated for each class in the system. A template VHDL-AMS file containing a general entity/architecture structure, in which the component details can be placed, is then read

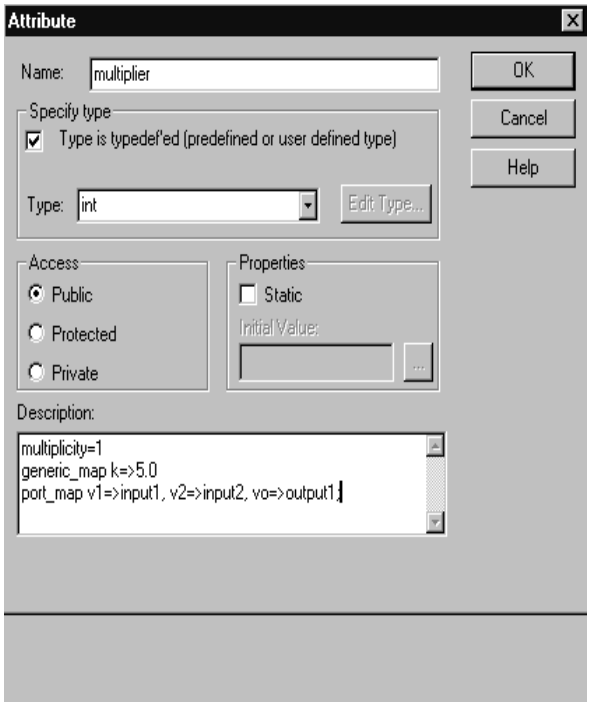

Fig. 9. Global value mappings for each class

in and populated with the specific data. Finally a new VHDL-AMS file is then outputted for each component with specific details about each class using the information previously obtained for each.

5. Example problem

5.1. Analogue subsystem

In this section a simple example system is used in order to illustrate the capability of this approach. Figure 11 shows a block diagram of an analogue subsystem containing four different components: a multiplier, a low pass filter and a summer. Sine waves are used as inputs. If this system is to be modelled in UML, three classes are required in the class diagram, one for each component. In addition there is a requirement to model the sinewave generator. The methodology discussed in the previous section is used to model this system.

The class diagram in UML, which represents the system in Fig. 11, is shown in Fig. 12. Figure 12 includes sinesources which are the stimulus for the system, which are required in order to generate a testbench for the system. Each component in the system is represented by a class within the class diagram. The sinesources, which are the stimuli are also represented using a class. The representation of component terminals and the constant values are defined in the attributes section, as shown. A transfer function, which defines the operation of each component, is declared in the operations section of each class and is equivalent to the architecture construct in VHDL-AMS. The mapping previously described is then used to automatically generate VHDL-AMS code from the class diagram. Using this technique, the VHDL-AMS code in Fig. 13 was automatically generated to describe the system in Fig. 11.

With reference to Fig. 12 (which is the UML class representation), the name of each sub system component has been placed in the first section and the port, i.e. the input and output terminals and constant values are entered in the second section. It is specified whether these are electrical terminals, real quantities or constant generic values. This information is subsequently mapped to the VHDL-AMS entity section of each component. These areas of code are marked 'ENTITY' in Fig. 13. The transfer function, which defines the operation of each component, is defined in the third section of each class in Fig. 12. In each class this function is called 'equation' and was entered in the operations form for each class. Equations maps to the architecture section of each com-

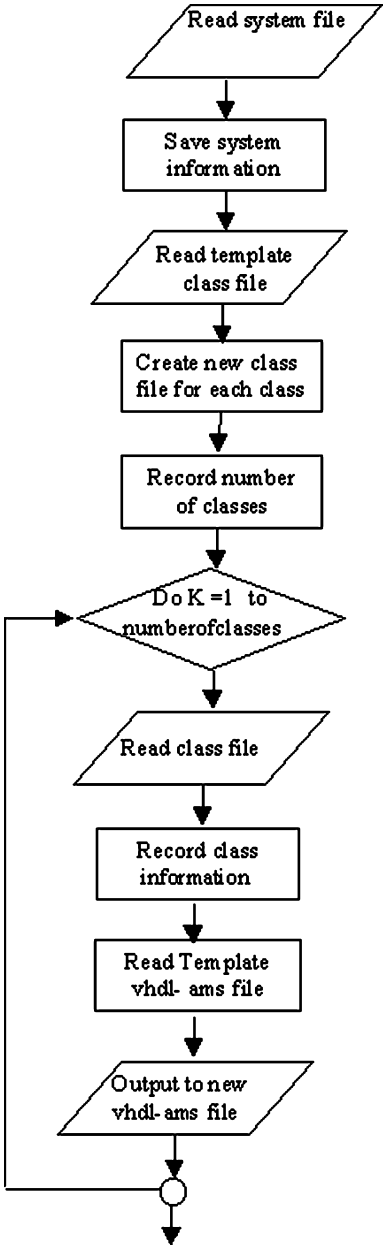

Fig. 10. Algorithm for mapping UML models to VHDL-AMS specification

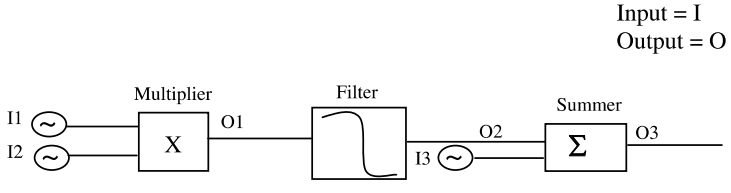

Fig. 11. Subsystem to be modelled

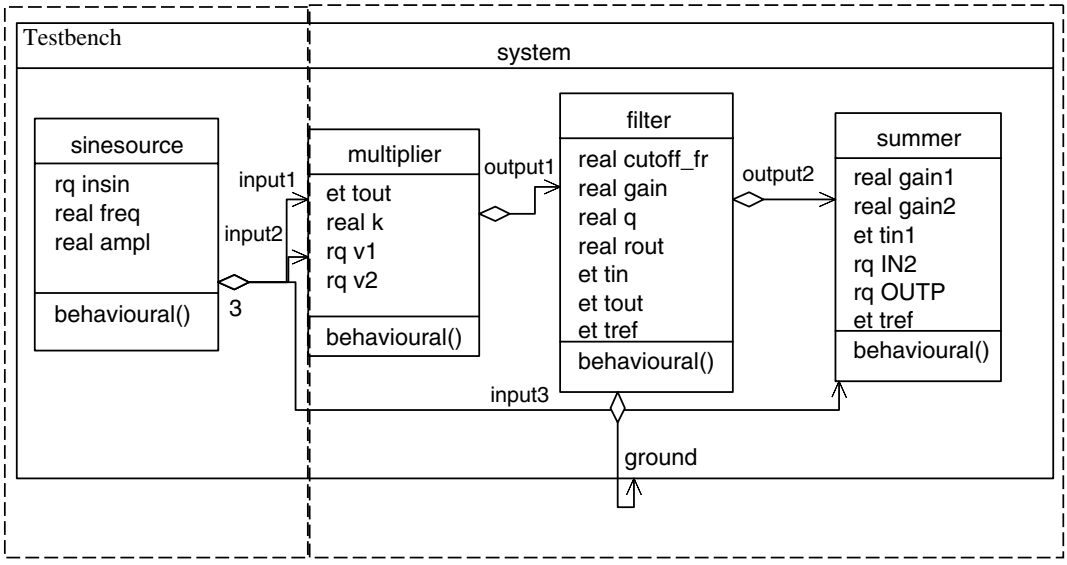

Fig. 12. Class diagram for example system in Fig. 11

ponent in the VHDL-AMS code in Fig. 13. The equations for the behaviour of the low pass filter, multiplier, summer are marked eq1, eq2, eq3 and eq4 respectively in Fig. 13. The code representing the connectivity of this system is shown separately in the testbench in Fig. 14. The connectivity of a system is usually shown in a structural architecture, however in the simple system discussed here it is included in the testbench. This code has been automatically generated from the connection details entered in the system class in the UML model.

5.2. Simulation results

When VHDL-AMS code is successfully mapped from a UML representation, the subsequent step is to simulate and eventually synthesise the code. A VHDL-AMS simulation tool called Hamster was chosen for the simulation [Www00]. The code in Figs. 13 and 14, which was generated from the UML representation of the example subsystem in Fig. 11, was used as input for the code simulator and the results are labelled in Fig. 15. Graphs 1, 2 and 3 in Fig. 15 show the three inputs to the system I1, I2 and I3 which produce the resulting output of the system. This output is the overall response from the system. In Fig. 16, the output from the VHDL simulator is compared to the same system simulated in MATLAB. Clearly it can be seen from these graphs that the response from the code simulator is almost identical to that generated by the MATLAB simulator. Any discrepancy in the two outputs is due to limitations in specifying parameter values in the MATLAB models i.e. the MATLAB models may be slightly different than the VHDL-AMS models. These graphs demonstrate that the mapping from UML to VHDL-AMS generates code is correct, and hence confirms that it is totally feasible to model an analogue system effectively using UML class diagrams.

6. Conclusions

In this paper a novel approach for the modelling of analogue subsystems at a high level using UML and a subsequent mapping to VHDL-AMS was presented. Previous research has demonstrated that a mapping from UML to VHDL in the digital domain is feasible. This work has extended these capabilities of UML further by developing a methodology to allow a behavioural representation of an analogue system in UML and a subsequent mapping from UML to VHDL-AMS. A specific rule-base is defined in order to enable an automatic generation of VHDL-AMS code from a UML representation. Class diagrams are selected, because they have the potential to represent the types of objects (i.e. components) in an analogue system and the nature of the relationship (i.e. connectivity) that exists in these systems. Each class is represented as a VHDL-AMS entity and architecture pair. An overall class defines system connectivity; and the parameters may also be entered in this class. The generated VHDL-AMS source code may subsequently be used to synthesise the analogue subsystem represented by the UML diagrams.

```
LIBRARY DISCIPLINES;
LIBRARY IEEE;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
USE IEEE.MATH_REAL.ALL;

    ENTITY filter IS
       GENERIC (
          cutoff_freq : REAL;
          gain        : REAL;
          q           : REAL;
          rout        : REAL);
        PORT (TERMINAL tin, tout, tref  : ELECTRICAL);

     END ;

    ARCHITECTURE behav OF filter IS

QUANTITY vin ACROSS       tin  TO tref;
 QUANTITY vout ACROSS iout THROUGH tout TO tref;
        QUANTITY v1: REAL;

    BEGIN
       v1 == vout'DOT;
       vin == v1'DOT/(gain*math_2_pi*cutoff_freq*math_2_pi*cutoff_freq) +
              v1/(gain*q*math_2_pi*cutoff_freq) + vout/gain;       --eq1
    END;

LIBRARY DISCIPLINES;
USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;

   ENTITY multiplier IS
            GENERIC( K : REAL :=1000.0);
                        PORT     (QUANTITY v1,v2 :IN REAL;
                                  TERMINAL tout: ELECTRICAL);
            END;

ARCHITECTURE behav OF multiplier IS
QUANTITY vout ACROSS iout THROUGH tout;

             BEGIN
             vout == K*(v1*v2);       -- eq2
            END ;

LIBRARY IEEE;
USE IEEE.MATH_REAL.ALL;

 ENTITY summer is
    GENERIC          (GAIN1 : REAL := 1.0;
                      GAIN2 : REAL := 1.0);
     PORT     (  QUANTITY  IN2, OUTP  : in  REAL;
                 TERMINAL tin1 , tref: ELECTRICAL);
  END summer;
ARCHITECTURE behav OF summer IS

QUANTITY IN1 ACROSS       tin1  TO tref;

    BEGIN
              OUTP == GAIN1*IN1 + GAIN2*IN2;      -- eq3

     END ;

 USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
USE IEEE.MATH_REAL.ALL;
LIBRARY DISCIPLINES;
LIBRARY IEEE;
ENTITY sineSource IS
  GENERIC  (ampl,freq : REAL);
   PORT(QUANTITY insin :out REAL);
END;

ARCHITECTURE behav OF sineSource IS
     BEGIN
   insin==ampl * sin (2.0*3.14* freq * now);      -- eq4

    END;
```

Fig. 13. Automatically generated VHDL-AMS code for example

```
LIBRARY DISCIPLINES;
LIBRARY IEEE;

USE DISCIPLINES.ELECTROMAGNETIC_SYSTEM.ALL;
USE IEEE.MATH_REAL.ALL;

ENTITY Test_bench IS
END Test_bench ;

ARCHITECTURE behav OF Test_bench IS

   QUANTITY INPUT1,INPUT2,INPUT3,OUTPUT3  : REAL;
TERMINAL OUTPUT1, OUTPUT2, GROUND : ELECTRICAL;

 BEGIN
 C1: ENTITY multiplier (behav)  GENERIC MAP  (K => 5.0)
                                          PORT MAP (v1 => INPUT1, v2 => INPUT2, tout => OUTPUT1);
 C2: ENTITY filter       (behav)   GENERIC MAP (cutoff_freq =>300,gain => 0.99,q => 1.5, rout=> 1.0e4)
                                  PORT MAP (tin => OUTPUT1, tout=>OUTPUT2, tref=>GROUND);
 C3: ENTITY summer    (behav)    GENERIC MAP  (GAIN1 => 1.0, GAIN2 => 1.0)
     PORT MAP (tin1=>OUTPUT2, IN2=>INPUT3, OUTP=>OUTPUT3, tref=> GROUND);
 C4: ENTITY sineSource (behav) GENERIC MAP (50.0,5000.0) PORT MAP (insin =>INPUT1);

 C5: ENTITY sineSource (behav) GENERIC MAP (25.0,1000.0) PORT MAP (insin => INPUT2);

 C6: ENTITY sineSource (behav) GENERIC MAP (90.0,5000.0) PORT MAP (insin => INPUT3);


END ARCHITECTURE ;
```

Fig. 14. Automatically generated code for the connectivity of the system

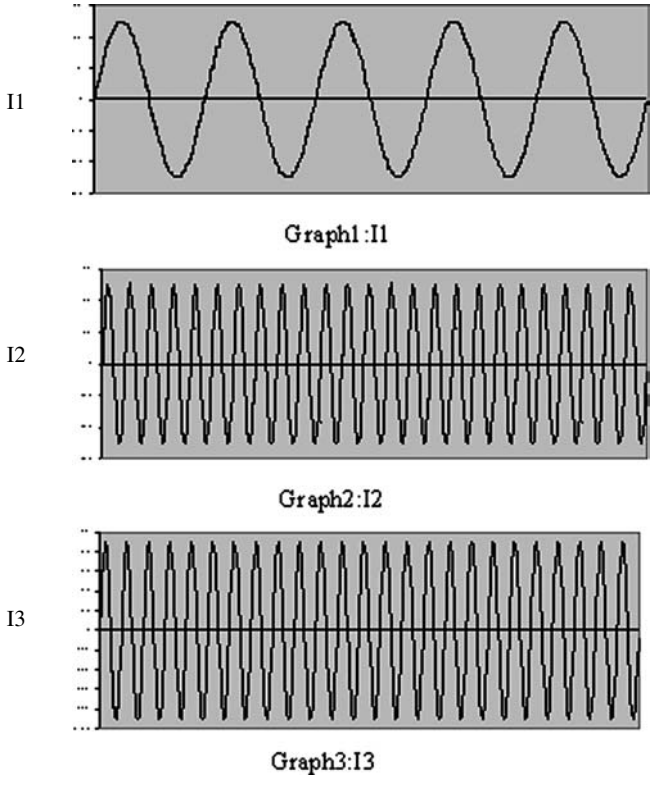

Fig. 15. Inputs to the system

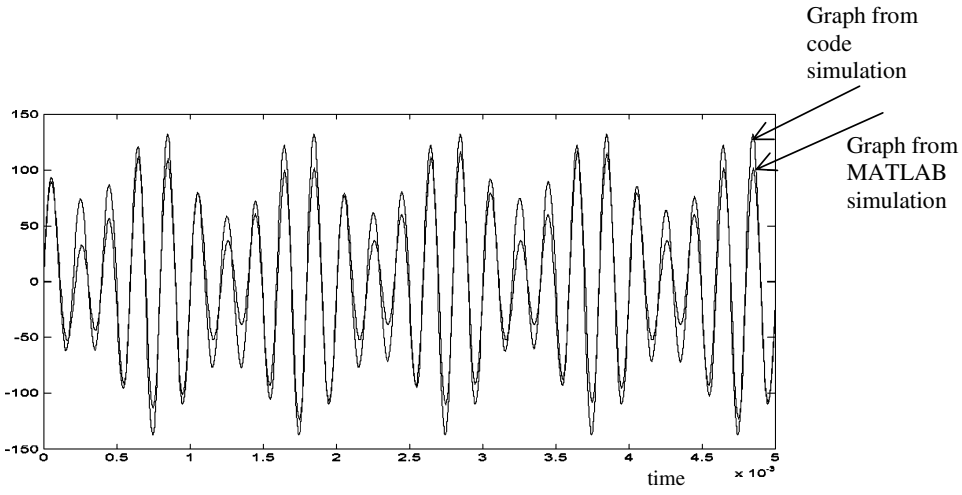

Fig. 16. Graph from MATLAB simulation and code simulation

An example analogue subsystem was modelled using class diagrams and subsequently mapped to VHDL-AMS. The resultant VHDL-AMS code was simulated using a VHDL-AMS simulator and the results were found to agree with those of a conventional circuit simulation. The results illustrate that this approach provides an effective means for mapping analogue subsystems and therefore mixed-signal solutions from a UML representation to a VHDL-AMS specification. This technique provides a basis for mixed signal representation in UML, which is the subject of a further publication.

References

[Axe01] Axelsson J (2001) Unified modeling of real-time control systems and their physical environments using UML. In: Proceedings, 8th annual IEEE international conference and workshop on the engineering of computer based systems-ECBS 2001. IEEE Computer Society, Los Alamitos, CA, pp 18–25

[Bad98] Badr I (1998) Object-oriented design of real-time systems. Circuit Cellar 101:26–31

[Bez99] Bezivin J, Muller P (1999) The birth and rise of a standard notation unified modeling language. In: UML'98. Beyond the notation. 1st international workshop selected papers. Springer, Berlin Heidelberg New York

[Boo97] Booch G, Rambaugh J, Jacobson I (1997) Unified modeling language user guide. Addison-Wesley, Reading, MA

[boo99] Booch G, Rambaugh J, Jacobson I (1999) The unified modeling language user guide. Addison-Wesley, Reading, MA

[Chr99] Christen E, Bakalar K (1999) VHDL-AMS: a hardware description language for analog and mixed-signal applications. IEEE Trans Circuits Syst II 46:1263–1272

[Dou99a] Douglass BP (1999) Real-rime UML. Addison-Wesley, Reading, MA

[Dou99b] Douglass BP (1999) Doing hard time. Addison-Wesley, Reading, MA

[Ver95] IEEE standard 1364 (1995) http://standards.ieee.org/. Cited 27 Feb. 2004

[Vhd02] IEEE standard VHDL reference manual, IEEE Std 1076 (2002) http://standards.ieee.org/. Cited 27 Feb. 2004

[Vhd99] IEEE standard VHDL analog and mixed-signal extensions, IEEE Std 1076.1-1 (1999) http://standards.ieee.org/. Cited 27 Feb. 2004

[Jig00] Jigorea R, Manolache S, Eles P, Peng Z (2000) Modeling of real-time embedded systems in an object-oriented design environment with UML. In: Proceedings, 3rd IEEE international symposium on object oriented real-time distributed computing, Los Alamitos, CA

[Mcu99] McUmber WE, Cheng BHC (1999) UML-based analysis of embedded systems using a mapping to VHDL, MSU-CPS-99-11. Michigan State University. In: Proceedings of IEEE high assurance software engineering, Washington, DC

[Mel99] Mellor SJ (1999) Automatic code generation from object models. C++ Report 11:6-30. SIGS Publications, http://www.sigs.com/. Cited 27 Feb. 2004

[Rap97] Rhapsody 3.0.1 MR1, copyright © 1997–2001. Rhapsody is a registered trademark of I-Logix Inc.

[Rum99] Rumbaugh J (1999) The unified modeling language reference manual. The Addison-Wesley object technology series. Addison-Wesley, Reading, MA

[Vams03] Verilog-AMS language reference manual, Ver 2.1. (2003) Accellera, http://www.eda.org/verilog-ams/htmlpages/lrm_draft.html

[Www00] http://www.hamster-ams.com/

Received May 2002
Accepted in revised form September 2003 by C. Delgado Kloos
Published online 13 March 2004