Custom Hardware Synthesis from UML

Michael Dossis

***Abstract*—The proliferation of extremely complex digital embedded and large computing systems have made the need for Electronic Design Automation (EDA) tools to accelerate their development and achieve higher overall productivity. Due to functional bugs that are found later in the design flow, as well as the iterative nature of such design methodologies, which delay a lot the development of their targeted products, industry and academia needs to invest to formal synthesis transformation techniques from high level specification formats such as UML. In this work high level and formal synthesis tools have been developed and integrated with electronic system level specification (ESL) techniques to automatically generate custom hardware circuits from UML. Because the transformations are formal the functional correctness of the generated implementations is guaranteed with respect to their UML specification. The proposed tools and methodology are proven usable with a number of real-world applications and benchmarks that are executed and synthesized into hardware implementations with the formal synthesis framework.**

***Index Terms*—UML, Formal Hardware Synthesis, EDA, High-Level Synthesis.**

I. INTRODUCTION

The recent years we witness an extremely high proliferation of complex embedded and advanced computing system digital integrated circuits (ICs) that constitute critical parts of the system's functionality. This complexity has burdened the work of the developing engineers and created a long design-to-product cycle. This long development time has often made the targeted products to miss the market window and has forced the engineers for including last-minute next version - related functionality into the current product version.

This high complexity of digital products and ICs can be realistically dealt only with fully automated and formal techniques and ESL design tools. These tools will generate provably-correct implementations only if they are based on formal and automated synthesis transformations. In order to raise the level of specification abstraction, high level specification formalisms such as UML are needed.

The major contribution of this work is a formal methodology and tools for automatically generate correct RTL VHDL/Verilog synthesizable implementations from system level UML diagrams. The generated implementations follow the first-time-right fashion and using this methodology last moment bug discovery and fixing are avoided. The underlying technology consists of the CubedC tools that were designed and developed by the author of this work. These tools take the automatically generated from the UML diagrams ADA code and formally and optimally transform it into a number of standalone hardware accelerator implementations. The generated accelerators are coded in IEEE std VHDL/Verilog (chosen by the user) and they are directly and with no manual alteration synthesizable into custom hardware using any academic or industrial RTL synthesizer. Apart from the type of the generated HDL code the user has in his disposal a number of custom hardware options, such as the location of large multi-dimensional objects in external memories and the choice between a massively-parallel architecture and a conventional FSM+datapath template. Moreover, the CubedC synthesizer can be guided with global or module-related resource constraints. These are fed into the embedded PARCS (Parallel, Abstract Resource – Constrained Scheduler) optimizer to generate optimal state machines for the accelerator control part.

The usability of the proposed UML-to-hardware design method is proven with a number of prototyped applications. Next section provides related existing work and bibliographical discussion. Section III presents the proposed design flow and CubedC tools. Section IV presents the UML set of diagrams used for our applications. Section V discusses experimental implementations from selected benchmarks. The last section draws useful conclusions and proposes future work.

Manuscript received July 20, 2014.

Michael Dossis, Department of Informatics Engineering, Technological Educational Institute of Western Macedonia, Kastoria, Greece

II. RELATED WORK AND BACKGROUND

A. UML for systems and hardware modeling

UML has been used extensively for software system level specification. It contains a number of diagram types and relations that define the functionality and other attributes of software system level descriptions. It is primarily a modeling language that has standardized the visualization of software systems. Since 1997 that was standardized it is managed by the Object Management Group (OMG). The most common UML diagrams include the class diagram, the packages, the object diagram, the component diagram, the activity diagram and the use-case diagram, state machine diagram and interaction diagram and other diagrams that are derivatives of these [1], [2]. UML diagrams fall into two major categories: the diagrams that represent structure and those that depict behavioral aspects [1], [2], [3].

There have been a few efforts to use UML as a modeling format for hardware and systems, such as Systems-on-Chip [4]. Nevertheless, none of these efforts were based on

automated methods using formal techniques such as the one of this work. Moreover, this paper's approach is based on formal transformations and it uses UML as the starting level of a automated synthesis tools methodology that is applicable to any type of custom digital circuits hardware.

B. Introduction to the CubedC Formal Framework

AI techniques, such as expert systems, inference engines and rule-based frameworks built with logic programming techniques have been long investigated and found very suitable for their use in safety-critical applications and industrial level product design systems [5]. Nevertheless up to date, there are not complete toolset reports that are based in AI methodologies and that deliver standalone hardware coprocessors which speed up their host environment tasks.

The toolset consists of the frontend compiler and the backend compiler. These two compilation phases communicate and exchange information by means of the Intermediate Predicate Format [6] (IPF) database. IPF[1] has been enhanced through many years of research, to capture the complete set of the source code algorithmic semantics (e.g. data typing, operators, complex control structure and hierarchy, interfacing, etc.).

C. High-Level Synthesis Related Work and Background

An early work that reported synthesis of algorithmic DSL code into hardware implementations is found in [7]. The use of FPGA accelerators to speed up their host computing system performance is found in [8]. The use of proprietary specification formats, and targeting of specific domains (e.g. DSP) or certain architecture templates are reported in [9]-[11]. The High-level Synthesis scheduling task has been studied in [5], [12]-[14]. In contrary to most of the available synthesis tools, the presented synthesis framework can process any arbitrary program code sets with as much complex control flow. There are no restrictions in the subset of the ADA language which is accepted by the frontend phase of the design framework synthesizer. Moreover, the input code can be hierarchical with as many subroutines as needed calling other subroutines.

Formal intermediate representation formats, can be found in [15]-[18]. Compiler-generators have been utilized to automatically generate large parts of compilers using formal input code syntax definitions [16]-[18]. Past circuit interface synthesis attempts along with the core functions generate protocol conversion circuitry for connecting multiple modules that have different and arbitrary communication protocols with the host environment [19]. High-level Synthesis of a small subset of the ANSI C language into efficient hardware can be found in [20].

Techniques to optimize data-flow expressions based on the Taylor Expansion Diagrams are described in [21]. An approach for reducing the power consumption of memory elements using dual power supply voltages is found in [22]. Rapid prototyping from SystemC models is reported in the SystemCoDesigner tool [23].

[1] The Intermediate Predicate Format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization

III. UML Design Flow and CubedC tools

The UML flow that we followed was based on the tools WinA&D from the company Excel Software [24]. WinA&D can support UML class diagrams and dictionary for regular package-based ADA specification code. The ADA package can be directly generated from interactive processing with the WinA&D UML diagrams front.

The generated ADA package from the WinA&D UML front is then passed onto the CubedC tools for hardware synthesis into RTL VHDL/Verilog [25]. Then the RTL code is implemented into any target ASIC or FPGA technology. The UML to hardware design flow is shown in Fig. 1.

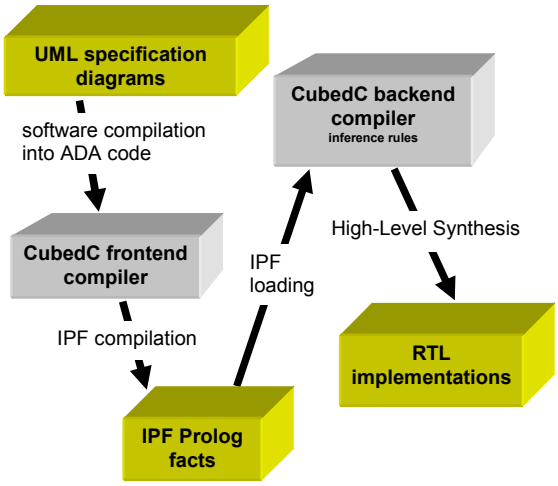

Figure 1: UML to hardware implementation design flow

The CubedC compiler consists of the frontend (ADA to IPF) and backend compiler (HLS tool). The HLS CubedC is based on formal rules which are constructed with a large set of inference engine Prolog statements [25], [26]. Therefore, and since the frontend compiler is constructed with automated compiler-compiler techniques the whole hardware synthesis flow is formal. Thus, the generated hardware implementations are provably-correct in regard to the input UML diagrams and ADA specification code.

At the moment, UML class diagrams and the dictionary are used to capture the functionality of the ADA targeted code. The code structure consists of an ADA package (library module of this language) and a number of global type declarations and subprograms (if necessary) to capture the semantics of the algorithm to be prototyped.

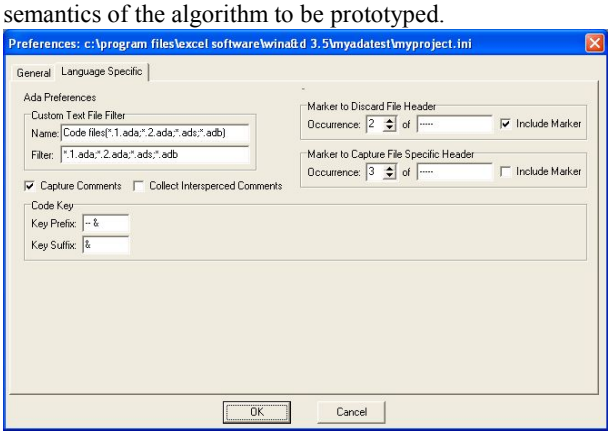

Figure 2: The WinTranslator project parameter configuration screen

The frontend UML translation tool of Excel Software is the WinTranslator. It needs to be configured for ADA code generation. The initial screenshot that contains this information about the tool configuration and starting of a new project is shown in Fig. 2. In order to guide the ADA code generation process several parameters are set in the tool as shown in Fig. 3.

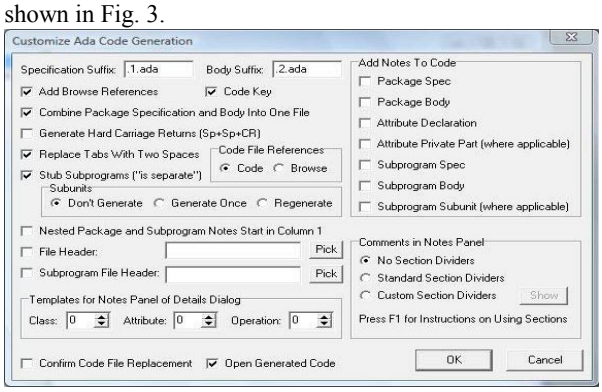

Figure 3: The ADA code generation customization screen

During generation of the ADA code, and if several ADA types need to be defined the screen in Fig. 4 shows how to define the types attributes for the ADA classes.

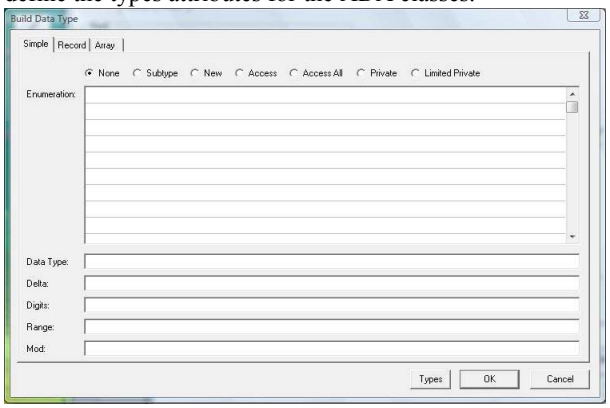

Figure 4: The ADA type attribute settings in WinA&D

For the selection of collection of types and their attributes the screen in Fig. 5 is used.

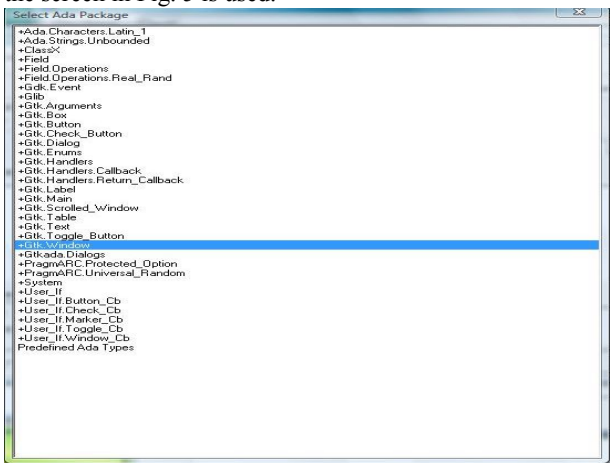

Figure 5: The types collection and attributes screen

By using the collection tool, and if the ADA language has been chosen, certain class relations between classes can be defined. Since the dictionary contains all the essential ADA constructs, for the certain program, using the class relation dialog we can add any program attributes by using the command merge. There can be more class instances in a UML diagram, in different diagrams in a project or multiple classes in a UML specification. In the same way, there can be multiple class relation instances in a UML diagram, in different diagrams or different ADA class definition windows.

Aggregation relations as shown in Fig. 6, are defined between two class objects in order to represent ADA parent and children packages.

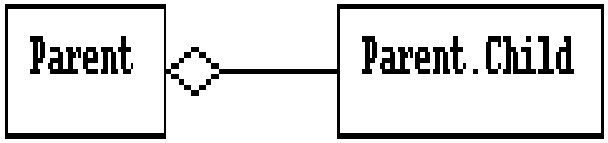

Figure 6: Aggregation relations between class objects to define package relations

An Aggregation Relation By Value as shown in the diagram of Fig. 7 determines that the ADA code of the child package will be integrated and embedded in the parent package. The particular relation is defined with the Export Control at the parent side, where the "+" (Public) signs denotes embedding in the Public part of the parent package, the "-" (Private) sign denotes embedding in the private part and "=" (Implementation) denotes embedding of code to the parent package body.

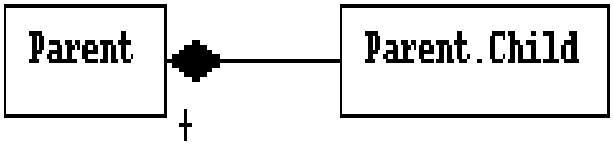

Figure 7: UML package relation with the type definition of the kind of the aggregation relation

A general ADA package is defined with the formal parameters in a dashed parallelogram in the class object side, as shown in Fig. 8.

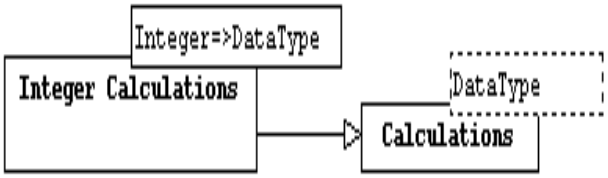

Figure 8: the class to object relation with the formal/actual parameters

In the same relation of this figure, the actual parameters are shown with the parallelogram connected to the respective class box. The particular relation of the general package shows the actual parameterized class to point to the formal class with the type of arrow as in Fig. 8.

The relations of type Dependence between class objects, generate With statements in the generated ADA code. These are library reference statements, and they are denoted with a dashed line arrow, as in Fig. 9. The Export sign at the left side

of the diagram of this figure, determines as to whether the With declaration belongs to the body or the declaration of the ADA package.

Similar diagrams apply to other generated ADA constructs as types, declarations, subprograms, variables, parameters and so on.

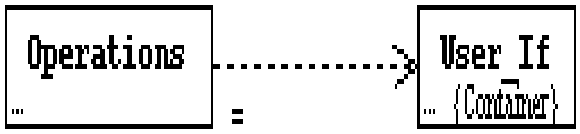

Figure 9: Type of body package Operations depends on package User_If

A. UML class diagram generation and ADA translation

WinA&D focuses on ADA code generation and relevant UML diagram translation and configuration. WinTranslator processes source ADA code and creates a number of dictionary entries that are inserted in WinA&D to generate UML class diagrams. In any case the object-oriented features of ADA such as named records, and type casting/extensions are not supported. Nevertheless these features are not yet supported in the CubedC synthesizer so this doesn't affect the present work. In our design flow, UML class objects model ADA packages, object relations model aggregation, inheritance and dependence relations between ADA packages.

When a new class object is added to the UML diagram, the corresponding dictionary entry is initialized in the class property window as it is shown in Fig. 10.

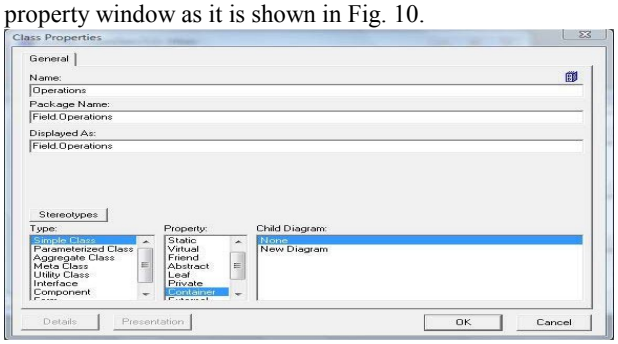

Figure 10: UML class object properties to be targeted to ADA

The package name is entered in the respective field as shown in Fig. 10 and automatically appended in the "Displayed As" field. For a child-package it is possible to delete the prefix from the Displayed field, and leave only the name that matches the UML diagram entry. The merge instruction adds a dictionary entry with the name of the package that it is entered as described above. The "Class Attributes and Operation" dialog window is used to enter the various algorithmic properties and features of a class. After entering all these details, appropriate dictionary entries are created for the UML diagrams.

Many fields in the dialog window of Fig. 10 are used to define the exact type of the ADA package that is modeled with the respective UML diagram class object. For example, the Type field can be set to either of the following: Single, Utility, Parameterized or Interface. All other types of classes are ignored during ADA code generation. The Simple class type models a package that can create instances of one or more class objects.

As mentioned above, the "Class Attribute And Operation" dialog window is used to enter many details and features of class objects. The class attributes fall in the following categories: State, Object, Type, Exception or Misc. The attribute type defines how the particular property will be mapped onto the generated ADA code detail using the WinA&D code generator. For example, the code generator places attributes of type State in record entries that are declared in the Public, το Private or Implementation section of the package code.

The class of type Utility is similar to the simple class, except that there is only one class object, the class itself. This class type is the most safe because it hides most of the class information since there is no public declaration of its data. During the code generation the parameter entries are placed in the package body along with the entries of the variable declarations.

The interface class type models a package of types that contains a list of type declarations but it doesn't contain any subprograms or parameters. For an interface class only a single package specification is generated, with no package body.

A Parameterized type of class, is used to create generic packages and instances. The parameterized class is similar to the simple class in the UML diagram, but with an extended parallelogram at the right upper corner of the class box, as shown in the example of Fig. 11.

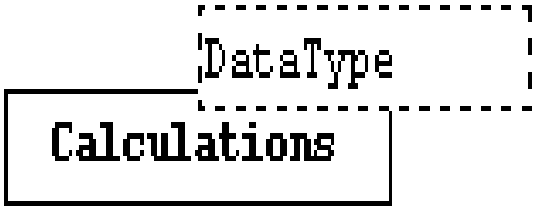

Figure 11: Parameterized class Calculations

The extended parallelogram contains a list of parameters of the class, which are defined in the Parameters field of the Class Properties dialog. This field is like the name in the Displayed as field and it is used only for the diagram appearance. After entering the parameterized class in the dictionary the Class tab of the "Class Attributes And Operation" dialog is used for the definition of the parameter list which is required for the code generation.

This type of class models a generic package. This class may contain properties and operations that are declared in the generic package. During the code generation, and for every parameterized class both the package interface as well as the main package body files are processed and modified accordingly with the UML diagram definitions.

B. Class Attributes

As soon as the UML class diagram is designed and added in the dictionary, detailed information about every class can be entered and defined. Using the Details button on WinA&D

"Class Attributes And Details" dialog window appears and it becomes ready for editing. The Attributes tab of this dialog is then used to define most of the properties of an ADA package except for the subprograms. The Operations tab of the dialog can be used to define the ADA subprograms.

In order to define an attribute list the attribute names are entered in the respective field at the bottom of the dialog window and using the buttons New or Insert. In order to enter details of every attribute a double click on its name shows the "Attributes Details" dialog window, that is used if the ADA language has been selected and provides information related to this language, as shown in the snapshot screen of Fig. 12.

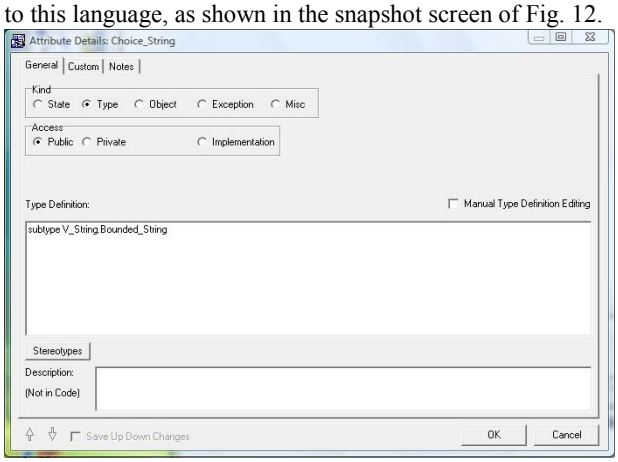

Figure 12: The attribute details dialog window for the ADA type Choice_String definition, from the WinA&D UML frontend.

The kind of attribute can be selected amongst the following choices: State, Type, Object, Exception and Misc. By doing so the window fields change as we select the attribute type.

C. State Attributes

The object instance in an object-oriented design has a state that is defined from the State Attributes. For an attribute of type State the Access field contains the choices: Public, Private, Protected and Implementation. The available choices depend on the object type. As an example the Simple class allows for Public, Private or Protected state attributes, while a Utility class allows only Implementation State attributes. An Interface class does not have any state so it doesn't allow for State attributes.

The kind of access to every State attribute, applies to all the State attributes throughout the whole class set. During the code generation all the State attributes are created in a package entry. The defined attribute can be either a component or a discriminate of the package entry using the State Type selection buttons.

Using the Class Properties dialog we can define the class type and then we can transfer it to the dictionary with the Merge command. The value that is stored in the dictionary determines which choices will appear in the Attributes Details dialog window. If we change the class type in the dialog Class Properties we have to use subsequently the Merge command so that we update the dictionary. The Data Type and Value fields are used to define the component features. The Data Type field is associated with a choice menu with available type names.

We can change the choices in this menu using the Types button, which shows every time a list of all the packages that we have defined in our UML WinA&D project. This process is depicted in the choice window of Fig. 13.

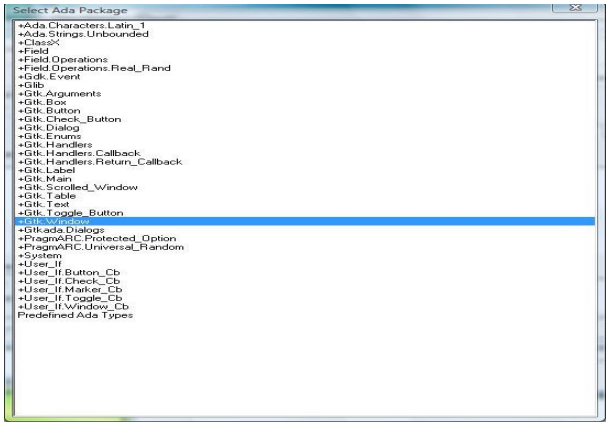

Figure 13: Select Ada Package list of package types to select one

D. Type, Object and Exception Attributes

If we select Types Attributes we can define the visibility using the Access button as follows (after the code generation):

- Public declaration is the one that is included in the public part of the package declaration.
- Private declaration is the one that appears in the private part of the package declaration.
- Implementation declaration means that it appears in the main body of the package.

The Type Definition field if activated shows the Build Data Type dialog window, with an example shown in Fig. 14.

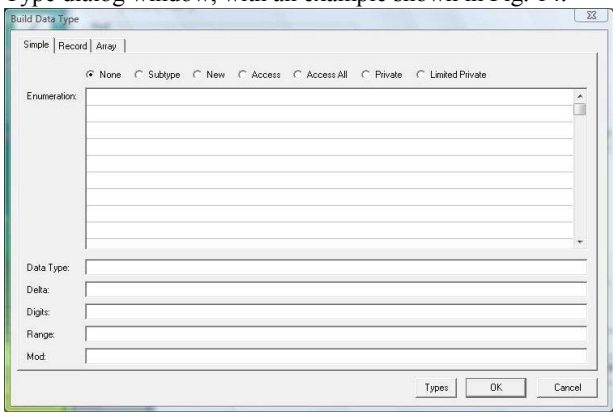

Figure 14: Build Data Types dialog window for the construction of ADA type definitions

In order to declare a public or private type we can set the Access selection as Public. In the Build Data Type dialog we can select Private or Limited Private. Thus, the type declaration is public but its implementation is private.

The Build Data Type dialog window has three panels: Simple, Record and Array, which determine the kind of type we want to create. With the OK button only the data we entered in the selected panel is used and the data of other panels are deleted. The Simple panel is used to define all the types and subtypes of ADA except the composite types such as records and arrays. When the type is based on existing subtype it can be selected from the respective menu. In order to change the types that appear in the menu the Types button

is used and a package is selected.

The Record panel is used to define ADA records. It has two sections, one for discriminate definition and one for definition of the record's elements. In every section, the Type field reports defined types in it, or in other packages that we have already defined. In order to make the window with the available types appear, first a cell needs to be selected in the Type column and then using right click. The Array panel contains a matrix for the definition of one or more array dimensions, where of course we need to select the type of the array's elements.

The Object property in the Attribute Details dialog, is used to define variables, constants, deference constants and named numbers as determined by Object Type button. In the same dialog window, the exception property is used to define ADA exceptions for the particular package.

E. Miscellaneous Attributes

The Misc attribute provides a field with free text in order to define various declarations. In this field we can define Representation clauses, Pragmas, imports or other element types of the package that don't fit in any of the previous categories. During the code generation the text in the Declarations field is mapped directly in the ADA code that is created by the tool.

F. Class functions and subprograms

In order to define a list of functions we type the name of each procedure and function in the bottom part of the Operation panel of the Class Attributes And Operation dialog and we use the New or Insert buttons. The various details of each subprogram are entered using a double click on its name. By doing so the Operation Details dialog appears which provides information related to the ADA language.

If the subprogram is a function then the type of the returned data (results) is defined in the Return Type field. A new type can be written or one of the existing types can be selected from the selection menu. In order to modify the list of available types the Types button is used and then a package is selected.

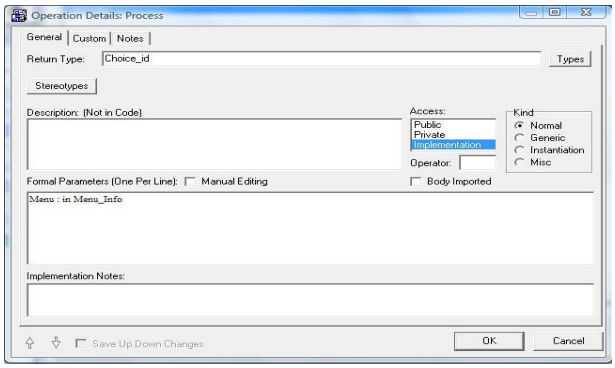

Figure 15: The subprograms definition dialog window

The definition of operations and subprograms is done via the Operations Details dialog window as shown in Fig. 15. In order to access the Build Subprogram Argument dialog window we select the Arguments field. In order to modify the list of parameters and arguments we select the selection box Manual Argument Ending. The Build Subprogram Arguments dialog allows for quick definition of the arguments in the lines of the matrix of Fig. 16. If a field in the column Data Type is selected the available type list appears with the right click. Many operation types can be defined in the Operations Details dialog window.

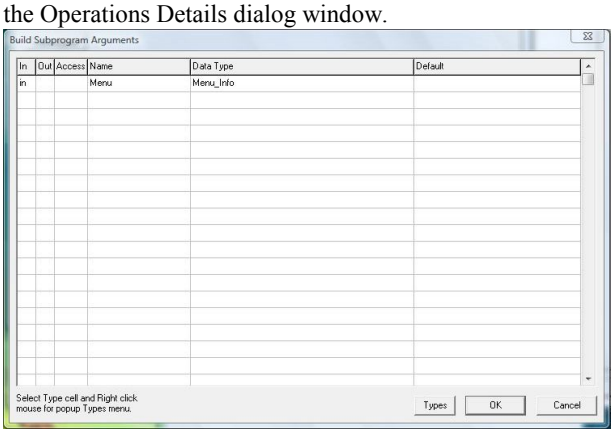

Figure 16: Building of Ada subprogram parameters using the Build Subprogram Arguments dialog window

The Generic choice is used to create a generic type subprogram and subsequently we define its arguments in the Parameters field of the στο πάνελ Special panel which then appears. In order to create an instance of a generic subprogram we create an operation of type In station and we use the respective panel to define the subprogram name and its actual parameters. The Package button allows to select a package so that the selection menu of the field Generic Subprogram shows all the generic subprograms of this package. It is important to note here that the subprogram is the building block of the ADA algorithm that generates a module of digital coprocessor in the targeted VHDL or Verilog code.

The Body Importer selection box is used to define an operation which uses a Pragma entry instead of the main body of a subprogram. ADA allows to produce an implementation for operators such as "+" and "=" but these characters are now allowed in the name of a dictionary entry. In order to define an operator we give it a name using alphanumerical characters, such "Plus" or "Equal" and then we write its actual name in the Operator field of the Operation Details dialog window.

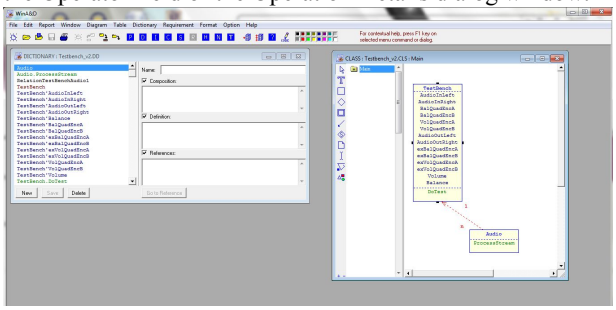

Figure 17: AudioControl UML diagram entries

IV. UML Diagrams for our Testcases

A. Audio Control

For this application a program is constructed which implements volume control and balance of a sound stream in real-time. For control Rotary Encoder components are used. Fig. 17 shows the UML diagram of AudioControl. In this figure both the dictionary entries and the class diagrams of the

Audio Control application are shown. The following part of ADA code is automatically produced with the code generator of WinA&D.

```
Package AudioControl is

procedure ProcessStream
(AudioInLeft, AudioInRight : in INTEGER; BalQuadEncA, BalQuadEncB : in BOOLEAN;VolQuadEncA, VolQuadEncB : in INTEGER;AudioOutLeft, AudioOutRight : out INTEGER);

end AudioControl;

package body AudioControl is

  procedure ProcessStream
(AudioInLeft, AudioInRight : in INTEGER;
BalQuadEncA, BalQuadEncB : in BOOLEAN;
VolQuadEncA, VolQuadEncB : in INTEGER;
AudioOutLeft, AudioOutRight : out INTEGER) is
Volume, Balance : INTEGER;
   begin
      if BalQuadEncA then
         if BalQuadEncB then
            if Balance < 100 then
               Balance := Balance + 1;
            end if;
         else
            if Balance > 0 then
               Balance := Balance - 1;
            end if;
         end if;
      end if;

      if VolQuadEncA then
         if VolQuadEncB then
            if Volume < 100 then
               Volume := Volume + 1;
            end if;
         else
            if Volume > 0 then
               Volume := Volume - 1;
            end if;
         end if;
      end if;
   AudioOutLeft := ((100 - Balance) / 100) * (Volume / 100) * AudioInLeft;
   AudioOutRight := (Balance / 100) * (Volume / 100) * AudioInRight;
      end ProcessStream;

end AudioControl;
```

The generated ADA code is then automatically and formally transformed and optimized, using the CubedC tools, into synthesizable VHDL RTL, which in turn is implemented on FPGA hardware.

B. MPEG

This application implements image compression that complies with the MPEG-1, ISO/IEC 11172 : 1993 standard. This image compression utilizes realistic methods of compression to significantly reduce the percentage of the data which is required for the regeneration of the image. It reduces or rejects on certain frequencies and spectrum areas that the human eye doesn't completely realize. Moreover, it exploits temporal and spatial deficiencies of the human eyes to achieve even greater data compression. Fig. 18 shows the UML diagram of the MPEG application.

The generated ADA code is too long to include in this work. It is synthesized using the CubedC tools into hardware implementations, the generated RTL is simulated to prove the first-time-right argument and then it is placed and routed on either ASIC and/or Xilinx FPGA technologies.

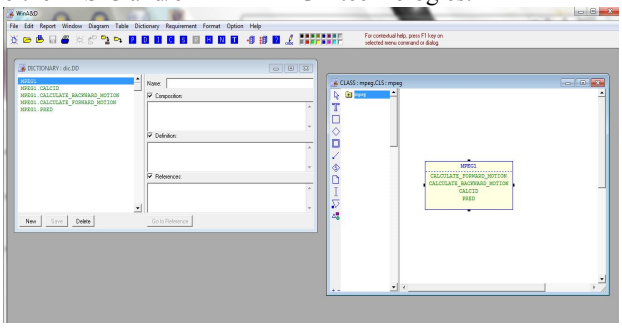

Figure 18: The MPEG-1 video compression engine UML specification

All the testcases in this work comply with the latest standards of UML and recent specifications and developments of OMG [27]-[35].

C. Computer Graphics Application

This UML specification defines an application that given the start and end coordinates it draws on the screen a straight line, using the digital differential analyzer (DDA) algorithm. The calculation of the pixel coordinates uses only integer numbers.

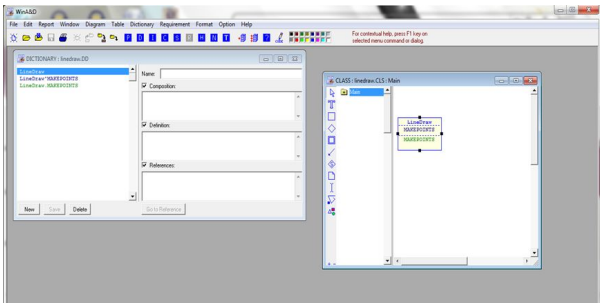

Figure 19: The line drawing application specification in UML diagrams

The DDA application specification in UML diagrams from the WinA&D screenshot is shown in Fig. 19. As in the other applications, in this case the generated ADA code is formally transformed and optimized into hardware and then implemented in FPGA technology using commercial backend tools. For reasons of space economy the application code is not included in this work.

D. Picture Edge-detection using Cellular Neural Networks

The authors of [36] developed and implemented a formal design flow for HLS of cellular neural network (CNNs) applications in hardware. The CNNs process images in real-time. The applications here target edge detection, intermediate tones and morphological image processing, and these can be used for medical, military or commercial surveillance applications.

After the UML is constructed it is used to automatically generate ADA code which in turn is transformed and optimized into hardware using the CubedC and backend RTL synthesis tools. In every case the generated implementation was simulated to prove the author's argument and cross-checked with high level verifications of the ADA algorithm and the rapidly prototyped hardware

implementation in FPGA hardware.

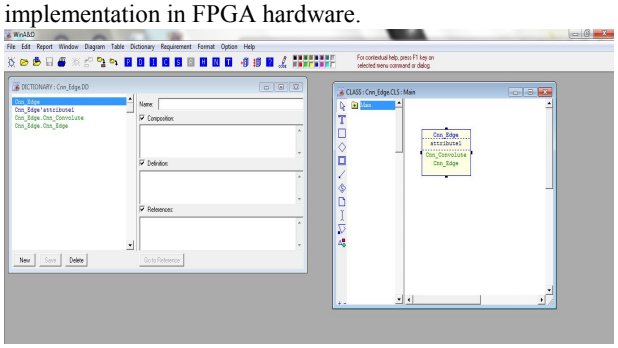

Figure 20: Image processing with CNNs application in UML diagrams

A snapshot of the UML diagram specification of the CNN edge detection application is shown in Fig. 20.

V. Experimental Results and Implementation Statistics

A. Simulation of Generated RTL Implementations

We observe rapid nature of our methodology from is a nested loops benchmark, which includes nested loops of 2 levels and it derives from a civil engineering application. In this test's statistics, from the very compact program code of the nested loops benchmark ADA model (155 lines of code), the hardware compiler optimizes about 100 initial FSM states and the produced optimized schedule is coded in about 2000 lines of optimized lines of synthesizable VHDL code.

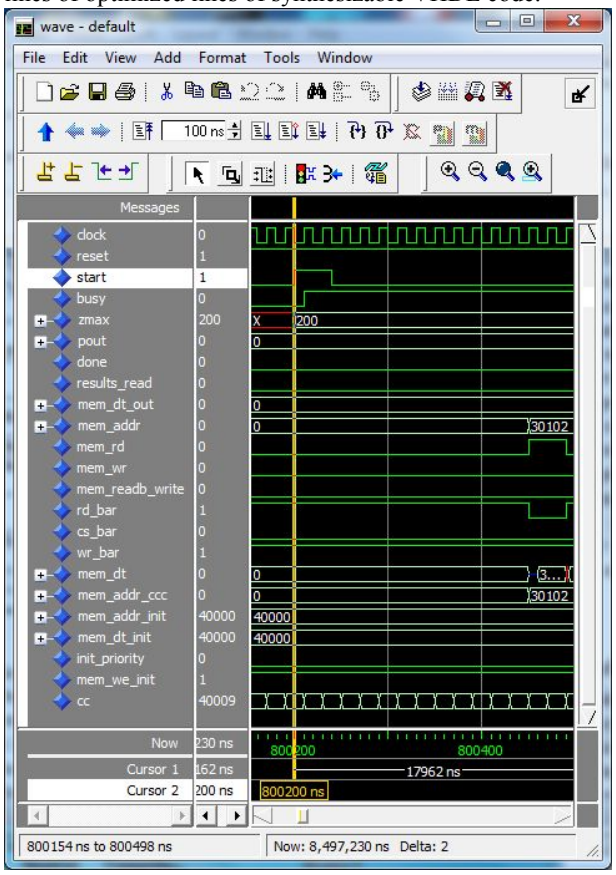

Figure 21: Simulation of the Start event of the n.l. test

The rapid increase in number of VHDL lines in combination with the large number of FSM states (100) is something particularly cumbersome to be dealt with manually, even by the most experienced hardware engineers and RTL code designers. All our tests were simulated both at the ADA compiled coprocessor package plus testbench model, as well as at the RTL simulation level and the results from both verification levels were compared and they found to coincide. The Start event of the initial nested loops schedule is visible at the simulation waveforms of Fig. 21. The reader can observe the Start pulse as well as the rising of the Busy signal which completes the handshake synchronization with the controlling computing environment (e.g. a host processor). At the bottom of this screen snapshot, there is an ever-updating clock cycle counter (cc), which at the Start rising time shows 40009 clock cycles. This is due to the preceding memories initialization with the array data.

A few clock cycles after the Start event there is some activity starting, as it can be observed from Fig. 21, on the memory interface signals, with names that usually have the prefix "mem_". The memory interface signals are used to fetch from and store data on external, shared memories. The read and write strobes (mem_rd, rd_bar, etc.) are shown which control the memory read and write cycles.

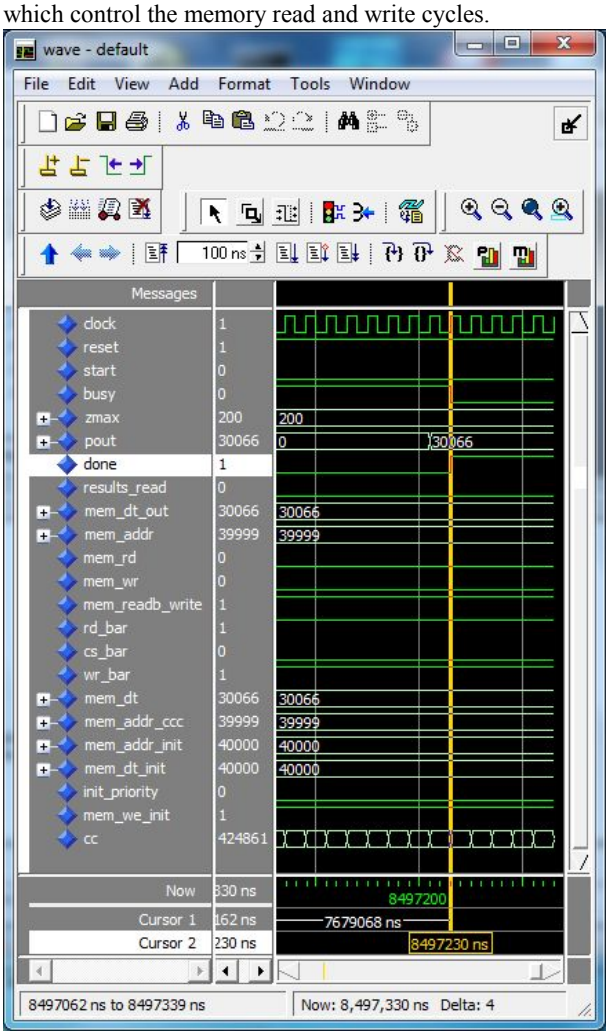

Figure 22: Simulation of the Done event of the n.l. test

The Done event, which signals the completion of the coprocessor function and the production of the final result on output "pout", giving the particular value of 30066, is shown in Fig. 22. This particular value was produced from the ADA code verification by using the same values for the design's

array components. In Fig. 22, the Busy signal goes low at 8497230 ns, and allows the external environment to conclude the handshake with a rise on the results_read input. Since the Start even appears at cc=40009 and the Busy event is on 424861 cc, it means that the unoptimized RTL completes the coprocessor calculations in 384852 cc, which, using a clock period of 20 ns it gives a total execution time of 7697,040 us, or about 7.6 ms.

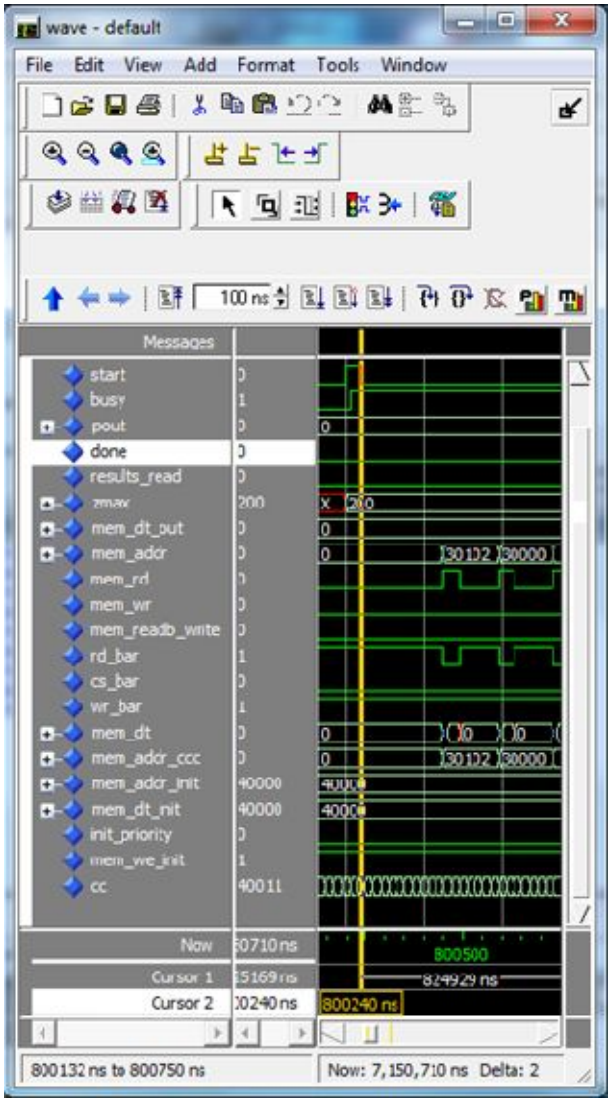

Figure 23: The Start event of the optimized (with PARCS) coprocessor

Of course the ADA verification, using compile and execute methods is much less verbose, portable, compact and easy. Considering this, and taking into account the formal features of the CCC transformations we can deduce that we can rely on the high-level behavioral code verification for a vast plethora of designs, without the continuous headache of detailed, heavy and slow RTL or even more gate simulations.

So far simulation data from the raw initial schedule of the massively parallel coprocessor were discussed. Fig. 23 and Fig. 24 give the Start and the Done events of the PARCS-optimized coprocessor. Fig. 23 shows a Start time of 800200 ns and Fig. 24 shows a Done time of 7150710 ns.

This gives us a PARCS processing time of 6350,510 us, which is about 6.3 ms. This is a reduction in processing time of about 17%. This is a significant improvement over the initial scheduling in a very control-intensive design with a large number of nested control constructs such as for loops and ifthen blocks.

To confirm, once again that all these simulations (unoptimized and not) produced the same results, between the massively-parallel simulations and the FSM+datapath simulations, as well as compared with the verification trials on the ADA compiled & executable specifications.

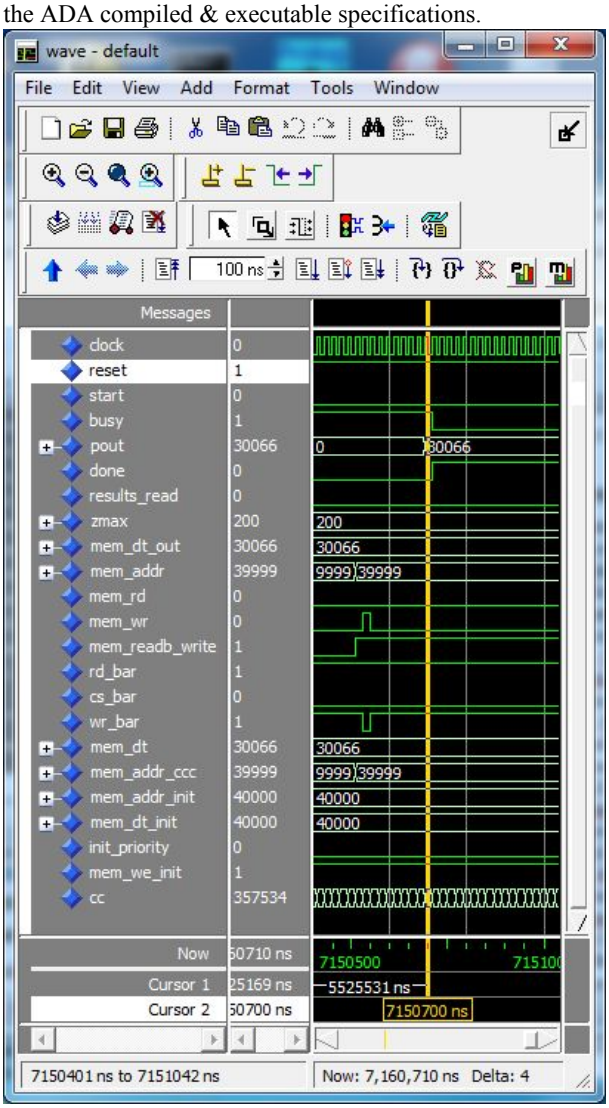

Figure 24: The Done event of the optimized (PARCS) coprocessor

B. Benchmarks and Implementation Statistics

A large number of design benchmarks were executed through the design framework. Nevertheless, five of these tests are analyzed here. They include a FIR filter from the DSP domain, the classical high-level synthesis benchmark which is a differential equation solver, an RSA design from cryptography, a test with nested for loops, and a large MPEG video compression design. An important conclusion is a dramatic explosion in the number of code lines and characters as we move from the very compact ADA code to the generated RTL models. From the experiment statistics in Table I, it seems that the behavioral ADA model is sometimes more than 10 times smaller and more compact than the generated VHDL. This is only one of the indices of the reduction in design

complexity for the users of the presented methodology, against those that insist in the conventional RTL hardware design.

TABLE I. STATISTICS OF THE FIRST FOUR DESIGN BENCHMARKS

	FIR filter	Differ. eq. solver	RSA mod exp	nested loops
number of subprograms in the source code	**2**	**1**	**5**	**6**
number of lines of source code	**62**	**31**	**226**	**155**
number of characters of source code	**1021**	**457**	**3690**	**2741**
number of VHDL entities produced	**2**	**1**	**5**	**6**
total number of VHDL lines	**373**	**200**	**1519**	**2000**
total number of characters in VHDL code	**11190**	**3663**	**47595**	**78928**
initial schedule's FSM # of states	**17**	**20**	**16**	**97**
optimized # of states	**10**	**13**	**11**	**81**
front-end compiler run time (secs)	**0,06**	**0,05**	**0,11**	**0,06**
back-end compiler run time (mins)	**0,03**	**0,02**	**0,25**	**10,41**

TABLE II. PARCS STATE REDUCTION STATISTICS FOR THE MPEG TEST

Module	Initial schedule states	PARCS parallelized states	State reduction rate
MPEG 1st routine	**88**	**56**	**36%**
MPEG 2nd routine	**88**	**56**	**36%**
MPEG 3rd routine	**37**	**25**	**32%**
MPEG top routine (with embed. mem)	**326**	**223**	**32%**
MPEG top routine (with external mem)	**462**	**343**	**26%**

Moreover, it is shown that the execution time of the synthesizer does not exceed about 10 minutes even for the most complex of these benchmarks. Let's note here that the hardware compilations were executed on a conventional Pentium-4 platform running the MS-Windows-XP-SP2 operating system, and which however exhibited short run times of the benchmark coprocessors.

The state reduction of the MPEG design after the initial schedule is processed by the PARCS optimizer is shown in Table II. Up to almost 40% improvement ("compression") of the initial schedule is observed using the PARCS optimizer.

The reduction in the number of FSM states before (initial schedule) and after (PARCS schedule) optimization by the backend compiler is shown graphically, for all the five tests, in 0It is obvious from this diagram that the optimization can achieve impressive results, especially for large designs such as the MPEG model.

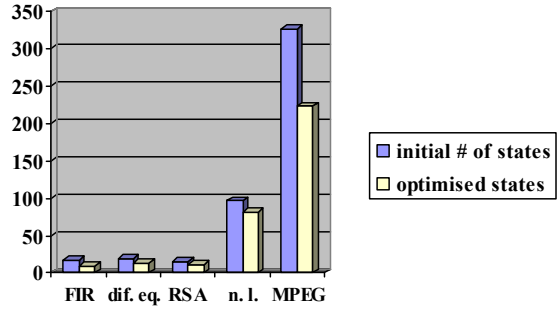

Figure 25: State reduction after PARCS for the five tests

The backend compiler is produces RTL (V)HDL for both the initial schedule (right before the PARCS optimizer is applied) and the optimized (parallelized) schedule.

TABLE III. XILINX VIRTEX-5 IMPLEMENTATION STATISTICS OF MPEG

Area and speed statistic for Xilinx	MPEG 4th routine (unoptimised)	MPEG 4th routine optimized (PARCS)
Number of slices	**3351**	**3087**
Number of flip-flops	**9503**	**9384**
Number of LUTs	**3626**	**3509**
Number of MACs (DSP48Es)	**111**	**111**
Number of adders/subtractors	**111**	**111**
Number of comparators	**9**	**9**
Minimum clock period	**9.911 ns (10 ns constraint)**	**9.930 ns (10 ns constraint)**
Number of FSM states	**462**	**344**
Number of bits of state encoding	**462**	**344**
State encoding	**Automatic one-hot**	**Automatic one-hot**
Synthesis & Place & Route run-time	**~31 mins**	**~25 mins**
temperature	**Range: 0.000 to 85.000 Celsius**	
Core voltage	**Range: 0.950 to 1.050 Volts**	

All of the produced RTL modules were synthesized with the Xilinx XST and Synopsys DC Ultra RTL synthesizers. Due to lack of space in this paper, only the fourth (top-level) routine of the MPEG benchmark Xilinx Synthesis and Place & Route, as well as Synopsys RTL synthesis run statistics are shown here, in III and IV. The XST and technology place and route ran via the Xilinx Design Suite 10.1 and the design was mapped on Xilinx Virtex5 XC5VLX330T, package FF1738, speed -2 device. The Synopsys flow ran via the DC-ULTRA version C-2009.06-SP3 and was mapped on TSMC 1.3 um technology libraries.

Both Xilinx and Synopsys statistics refer to design implementations that were compiled with the massively-parallel architecture option. Therefore, even more economic (in terms of area and used resources) figures could be attained with the conventional FSM+datapath

micro-architecture option.

It is remarkable that for both the initial and the optimized schedule the area and resource figures don't differ dramatically. This is due to the nature and connectivity features of the chosen RTL VHDL model coding style and architecture that are followed in the design flow of the presented design framework and hardware compiler toolset.

TABLE IV. SYNOPSYS DC-ULTRA WITH TSMC LIBRARIES RESULTS

Area and speed statistic for Synopsys DC	MPEG, 4th routine (unoptimised)	MPEG, 4th routine optimized (PARCS)
Number of cells	3152	3186
Combinational area	154345.422880	148026.032483
Noncombinational area	390809.611002	389015.961706
Total cell area	545155.033882	537041.994189
Eq. NAND2X1 gates	~54.515K gates	~53.704K gates
Run-time	30 mins	18 mins
Minimum clock period	9.19 ns (10 ns constraint with 0.81 ns slack)	9.26 ns (10 ns constraint with 0.74 ns slack)

Abstract, algorithmic coding in standard programming languages can be done in a fraction of the time required to model and debug the application directly in cycle-accurate, detailed (scheduled) RTL code, using any hardware description language. This method which *automatically transforms* whole ADA programs into VHDL RTL, along with the coprocessor's *interfaces*, is more efficient compared even to RTL coding by very experienced hardware designers, and particularly when the complexity of the hardware machine increases usually over a dozen states.

CONCLUSION

The main contribution of this paper is a formal, high-level hardware synthesis framework and a unified prototype tool-chain, which is based on compiler-compiler and logic-programming techniques, and UML as the starting system-level specification input to the tool chain.

The prototype tools transform a number of arbitrary input subprograms (at the moment coded in the ADA language) into an equivalent number of functionally-equivalent RTL VHDL hardware coprocessor descriptions. A very large number of input program applications were run through the hardware compiler, five of which were evaluated in this paper. In all cases, the functionality of the produced hardware accelerators matched that of the input subprograms. This was expected due to the formal definition/implementation of the various phases of the hardware compiler, including the intermediate IPF form and the logic rules of the backend phase.

Encouraging state-reduction rates of the PARCS scheduler-optimizer were observed for five benchmarks in this paper, which exceed 30% in some cases. Using its formal flow, the prototype hardware compiler can be used to develop complex systems in orders of magnitude shorter time and less engineering effort, than that which are usually required using conventional design approaches such as RTL coding or IP encapsulation and schematic entry using custom libraries.

Future extensions of this work include the backed HDL writer to produce RTL code in more hardware modeling languages such as SystemC and cycle-accurate C, which are currently under development. Another extension could be the inclusion of more than 2 operand operations as well as multi-cycle arithmetic unit modules.

Moreover, there is ongoing work to extend the IPF's semantics so that it can accommodate embedding of IP blocks (such as floating-point units) into the compilation flow, and enhance further the schedule optimizer algorithm for even more reduced schedules. Compiler phase validation techniques based on formal semantic such as RDF and XML flows are investigated. Furthermore, the UML-based system specification methods are being further studied and enhanced and updated versions of our CubedC framework are under development and expect to be published.

ACKNOWLEDGMENT

The student team: Giovanopoulou Georgia, Keskou Vasiliki, and Ntafou Evaggelia, contributed with initial development work, mainly on the WinTranslator UML diagrams and generation of ADA code. They worked under the guidance of the author of this paper.

REFERENCES

[1] M. Fowler,*UML Distilled. A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman, 2000.

[2] G. Booch, RA. Maksimchuk, MW. Engle, BJ. Young, J. Conallen and KA. Houston, *Object-oriented analysis and design with applications*. USA: Addison-Wesley, 3rd ed., 2007.

[3] RC. Martin, *Agile software development: principles, patterns, and practices*. USA: Pearson Education, Inc, 2012.

[4] G. Martin, and W. Muller, (editors), *UML for SOC Design*. (research review reference book) AA Dordrecht, The Netherlands: Springer, 2005.

[5] *Intelligent Knowledge-Based Systems, Volume I*: Knowledge-Based Systems, Edited by Cornelius T. Leondes, Kluwer Academic Publishers, USA, 2005.

[6] M. Dossis, "Intermediate Predicate Format for Design Automation Tools", *Journal of Next Generation Information Technology (JNIT)*, vol. 1, no. 1, pp. 100-117, May 2010.

[7] R. Camposano, and W. Rosenstiel, "Synthesizing circuits from behavioural descriptions", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, Number 2, pp. 171-180, Feb. 1989.

[8] R. Pellizzoni, and M. Caccamo, "Real-Time Management of Hardware and Software Tasks for FPGA-based Embedded Systems", *IEEE Trans. Computers*, vol. 56, issue 12, pp. 1666-1680, Dec. 2007.

[9] A.E. Casavant, M.A. d'Abreu, M. Dragomirecky, D.A. Duff, J.R. Jasica, M.J. Hartman, K.S. Hwang, and W.D. Smith, "A synthesis environment for designing DSP systems", *IEEE Des. Test Comput.*, vol. 6, Number 2, pp. 35–44, Apr. 1989.

[10] I. Auge, F. Petrot, F. Donnet, and P. Gomez, "Platform-based design from parallel C specifications", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, Number 12, pp. 1811–1826, Dec. 2005.

[11] M.C. Molina, R. Ruiz-Sautua, J.M. Mendias, and R. Hermida, "Bitwise Scheduling to Balance the Computational Cost of Behavioral Specifications", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, Number 1, pp. 31-46, Jan. 2006.

[12] R.A. Walker and S. Chaudhuri, "Introduction to the scheduling problem", *IEEE Des. Test Comput.*, vol. 12, Number 2, pp. 60–69, Summer 1995.

[13] A.A.. Kountouris, and C. Wolinski, "Efficient Scheduling of Conditional Behaviors for High-Level Synthesis", *ACM Trans. on Design Automation of Electronic Systems*, vol. 7, Number 3, pp. 380–412, Jul. 2002.

[14] P.G. Paulin, and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, Number 6, pp. 661-679, Jun. 1989.

[15] A.V. Aho, and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 3rd Edition by M.A. Harisson, 1979.

[16] J.P. Tremplay, and P.G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw Hill, 1985.
[17] W.M. Waite, and G. Goos, *Compiler Construction*, Springer-Verlag Inc., New York, 1984.
[18] R. Hunter, *Compilers: Their Design and Construction Using Pascal*, John Wiley & Sons Ltd., 1985.
[19] B. Lin, and S. Vercauteren, "Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation", *Proc. of ACM ICCAD 1994*, pp. 101-108, San Jose, CA, Nov 1994.
[20] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis", *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 9, Issue 4, pp 441-470, Oct. 2004.
[21] D. Gomez-Prado, Q. Ren, M. Ciesielski, J. Guillot, and E. Boutillon, "Optimizing Data Flow Graphs to Minimize Hardware Implementation", *IEEE Design, Automation and Test in Europe Conference*, DATE'09, pp. 117-122, April 2009.
[22] I. Shin, S. Paik, and Y. Shin "Register Allocation for High-Level Synthesis Using Dual Supply Voltages", *Proceedings of the DAC'09, San Francisco, California*, USA, pp. 937-942, July 26-31, 2009.
[23] C. Haubelt, M. Meredith, T. Schlichter, and J. Keinert "SystemCoDesigner: Automatic Design Space Exploration and Rapid Prototyping from Behavioral Models", *Proceedings of the DAC 2008*, Anaheim, California, USA, pp. 580-585, June 8–13, 2008.
[24] *WinA&D products*, Excel Software, Available: http://www.excelsoftware.com/wina&dproducts.html
[25] M. F. Dossis, "Intelligent Custom Block Generation", Universal Journal of Electrical and Electronic Engineering, *Horizon Research Publishing Corporation (HRPUB)*, vol. 2, no. 2, pp. 59–69, February 2014.
[26] U. Nilsson, and J. Maluszynski, *Logic, Programming and Prolog*, John Wiley & Sons Ltd., 2nd Edition, 1995.
[27] D. August, et al., "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development", *IEEE Computer Architecture Letters*, vol. 6, issue 2, pp. 45-48, Feb. 2007.
[28] B. Bruegge, and A. H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Upper Saddle River, NJ, Prentice Hall, 2000
[29] M. Fowler, *UML Distilled*, Reading, Massachusetts, Addison Wesley, 2000
[30] D. Harel, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming: pp. 231-274, 1987
[31] I.M. Jacobson, and Christerson, et al. *Object-Oriented Software Engineering: A Use Case Driven Approach*, Wokingham, England, Addison-Wesley, 1992
[32] B. Oestereich, *Developing Software with UML: Object-Oriented Analysis and Design in Practice*, London, Person Education, 2001
[33] D. Rosenberg, and K. Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*, Reading, Massachusetts, Addison-Wesley, 1999
[34] J. Rumbaugh, and I. Jacobson, et al., *The Unified Modeling Language Reference Manual*, Boston, Addison Wesley.1999
[35] K. Scott, *UML Explained*, Boston, Massachusetts, Addison-Wesley, 2001
[36] M.F. Dossis, and D.E. Amanatidis, "Synthesizing Neural Nets into Image Processing Hardware", *Journal of Pattern Recognition and Intelligent Systems (PRIS)*, vol. 1, iss (vol.) 1, pp. 10-17, May 2013.

Michael F. Dossis has an Advanced Engineering Diploma from NTUA, Athens, Greece and a Ph.D. in Electrical and Electronic Engineering from the University of Bradford, UK. He is currently an Associate Professor of Informatics Engineering with the Higher TEI of Western Macedonia, Greece. He has been post-doctoral research and teaching staff with the Universities of Bradford and Oxford. (both in UK). He has also long industrial experience in design and development of multimillion-gate ASICs, FPGAs and processor core designs at LSI Logic, ARM, Virata (now Connexant) and Intracom Telecom (Greece).
During his 20-year career in industry and academia he has developed some millions of lines of high-level program and computer description/simulation code. His research interests include design automation, methodologies and tools for the design of digital electronic systems, architectures of application-specific integrated circuits, computer architecture, computer languages and their compilers, high-level synthesis and hardware-software codesign, formal design methods, applications of artificial intelligence, embedded systems, custom processor and computer architectures, and reconfigurable computing. He holds a number of international patents and publications in Formal Methods for hardware and system design - automation and he has developed high-level synthesis and ESL tools.