Considerations regarding the implementation of the ESPL programming language

Horia Ciocarlie, Cosmin-Mihai Vacarescu

***Abstract*—**This paper's scope is to describe the back-end implementation of the ESPL compiler, precisely the platform for the cod's simulation – a virtual machine – and the virtual code generator for this platform. ESPL is an object oriented programming language that is used for embedded systems and consequently the virtual machine is implemented in a manner that allows the simulation and testing of specific features of embedded systems. The virtual code generation is strongly dependent on the configuration of the virtual machine. Its starting point is the intermediate code produced by the front-end of the compiler, and, as final point, a file with machine code that will be load and executed by the virtual machine. The implementation is based on the OOP principles and every module can be replaced and upgraded if the interface rules are followed. The interaction with the user is realized trough the ESPL GUI. New windows and options were added to the previous GUI to support the features offered by virtual machine and the virtual code-generation.

***Keywords*—**Compiler, embedded systems, real-time, virtual machine.

I. Introduction

THIS paper represents the third step in the project called „Programming environment for the development of real-time distributed applications, designed for embedded systems" [1] which proposes a comparative study between real-time features of many already existent programming languages, the definition of a language based on intensive study and its implementation, and finally the construction of a simulation and execution environment for the object code.

An embedded system represents a computing system used in the environment of a greater system with the purpose of ensuring the computing and command functions.

A real-time system term refers to a system which coordinates a certain number of activities more or less simultaneous, with time constraints, and that has to assure the continuity, the security and the performance of their execution [2].

The real-time systems often appear under the form of embedded systems, but not all the embedded systems have a real-time behavior also like not the all real-time systems are embedded systems.

Although, the two system types do not exclude each other and in the area they overlap appears a combination of systems known as real-time embedded systems.

II. The Programming Language ESPL

ESPL acronym comes from "Embedded System Programming Language".

The primary definition for this language was the standard C language grammar. In accordance with the criteria for defining a programming language for real-time systems, we kept some constructions for it with some modifications [3].

ESPL is an imperative language, object oriented after the Java model, which offers concepts like: encapsulation, inheritance, polymorphism. Also it has elements of concurrent languages through the possibility of defining threads and mutual exclusions.

Furthermore the language offers possibility to access the resources of the embedded system such as registers and ports.

The exceptions mechanism in Java has been remodeled in order to correspond to an embedded environment. Instead of using objects (exceptions) to signal and treat errors, it was decided to use the interruption mechanism present in all the embedded systems [1].

A. Present language constructs

Classes

The *class* is the fundamental building block in most object oriented languages. A class is a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the immediate superclass of the class [3].

The Java model is the basis of the object-oriented approach of the new programming language. Hence, the *private, protected* and *public* modifiers, used in a method or variable declaration, have the same meaning as in the Java programming language.

Fields and methods

Two types of elements in are specified when a class is defined: *fields* (sometimes called data members), and *methods* (sometimes called member functions).

A *field* is an object of any type that you can communicate with via its reference. It can also be one of the primitive types. *A field* is a data item associated with a particular class as a whole (not with particular instances of the class) and they are defined in class definitions [1].

A *method* represents a function that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class [1].

Inheritance

Objects are defined in terms of classes and one can know a lot about an object by knowing its class. Object-oriented systems take this a step further and allow classes to be defined in terms of other classes.

Each subclass *inherits* state (in the form of variable declarations) from the superclass. However, subclasses are not limited to the states and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Subclasses can also override inherited methods and provide specialized implementations for those methods.

Inheritance is one of the cornerstones of object-oriented programming. The concept of inheritance is also based on the Java model, so the *extends* keyword is used for inheritance. A class extends another class to add functionality, either by adding fields or methods or by overriding methods of that class [1]. We will also use the keyword *this* to represent an instance of the class in which it appears and the keyword *super* to access members of a class inherited by the class in which it appears.

Constructors

The concept of a *constructor*, introduced in C++, is a special method automatically called when an object is created. The language will also adopt constructors. A constructor is a pseudo-method that creates an object. Constructors are instance methods with the same name as their class and they are invoked using the `new` keyword [1], [3].

Allocation and deallocation of memory

The *new* keyword is used to create an instance of a class. C++ is using destructors to free memory. Using destructors is no longer needed because the system will automatically free unused memory.

Exception handling mechanism

C and other earlier languages often had multiple error-handling schemes, and these were generally established by convention and not as part of the programming language. Typically, you returned a special value or set a flag, and the recipient was supposed to look at the value or the flag and determine that something was amiss. However, as the years passed, it was discovered that programmers wouldn't check for the error conditions. If you were thorough enough to check for an error every time you called a method, your code could turn into an unreadable nightmare.

The solution is to take the casual nature out of error handling and to enforce formality. C++ exception handling was based on Ada, and Java's is based primarily on C++ .

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception* [1].

After a method throws an exception, the runtime system attempts to find something to handle it. The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler* [3]. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception* [3]. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.

The synchronized keyword

The language will be distributed. Separate, concurrently running threads share data and must consider the state and activities of other threads. The *synchronized* keyword, when applied to a method or code block, guarantees that at most one thread at a time executes that code [1], [3].

B. Removed Language Constructs

Pointers

Although pointers represent an element which confers the programmer a great deal of flexibility, memory errors are the worst kind of errors in C and C++. They are hard to reproduce and thus hard to debug. Most developers agree that the misuse of pointers causes the majority of bugs in C/C++ programming. This is the main reason why the new language will not tolerate the use of any kind of pointers [1].

The new language will not have pointer variables, but it will have reference variables which are equivalent in concept, but use a simpler syntax. All variables that refer to objects (values of a class type) are reference variables. They refer to the object indirectly and are implemented with something like a machine address

Pointer and reference variables give the programmer great flexibility, but they can lead to difficult to find and correct errors in programs.

Global variables

Global variables are variables that are within the scope of all code. This usually means that references to these variables can be made in direct mode, and thus are faster than references to variables passed via parameter lists.

Global variables are dangerous because references to them can be made by unauthorized code, thus introducing subtle faults [3]. For this and other reasons, unwarranted use of global variables is to be avoided. Global parameter passing is only recommended when timing warrants, or if the use of parameter passing leads to convoluted code. In any case, the use of global variables must be clearly documented.

Data types

In C and C++, the *sizeof( )* operator satisfies a specific need: it returns the number of bytes allocated for data items. The most compelling need for *sizeof( )* in C and C++ is portability. Different data types might be different sizes on different machines, so the programmer must find out how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits. Programs could store larger values in integers on the first machine [3].

Portability is a huge headache for C and C++ programmers. The new language does not need a *sizeof( )* operator for this purpose, because all the data types will be the same size on all machines.

All data types will be signed so the keywords *signed* and *unsigned* from the C language are not necessary anymore.

The goal is to define a language with a minimal set of language constructors. Hence, the *short, long* and *double* data types can be eliminated. The functionality of the *struct* and *union* structures are provided by classes, so there is no need to include them in the language definition.

Type qualifiers

C recognizes three type qualifiers, *const,* *volatile*, and *restrict*. For the same reason (defining a grammar as simple as possible) all three type qualifiers were removed. The *const* type qualifier declares an object to be non-modifiable. The *volatile* type qualifier declares an item whose value can legitimately be changed by something beyond the control of the program in which it appears, such as a concurrently executing thread. Another reason for eliminating the *restrict* type qualifier is that it may only be applied to pointers (and the new language doesn't offer support for pointers) [1].

Type modifiers

As the local lifetime is the only possible way for local variables (there will be no global variables in the new language), the *auto* keyword is extremely rarely used. The *register* type modifier tells the compiler to store the variable being declared in a CPU register (if possible), to optimize access [1].

Iterative statements

The only control-flow statements for iterative execution will be constructed with the reserved keyword *for*. The other two iterative statements, *do* and *do while* were not included because *for* loop can be used to replace both of them.

Labeld statements

The *switch* statement is a construct that is used when many conditions are being tested for. Nested if/else statements arise when there are multiple alternative paths of execution based on some condition that is being tested for.

It is well structured, but can only be used in certain cases where only one variable is tested and all branches must depend on the value of that variable. The variable must be an integral type. Another case is when each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases. The *switch* statement can be implemented using nested *if/else* statements, when there are multiple alternative paths of execution based on some condition that is being tested for.

The goto statement

Most programming languages allow the programmer to trasfer program control unconditionally, usualy using the *goto* function. Modern software design practice avoids *goto* constructs as their use can result in badly-structured programs [1].

C. The Compilation Process

The compiler is a translator which reads a program written in a high level language (source language) and it translates it in equivalent language, another language, of lower level (object or destination language) [4].

The main functions of the compiler are: the analysis of the source program and the synthesis of the destination program (object code). As an important part of the translation process the compiler signals the user of to the presents of errors in the source program.

The compilation process breaks in more phases; each phase changes the source program from a representation to another [5].

The first phases of the compiler, until the intermediate code generation, were implemented in the former step of the project.

Also, the code optimization, an optional phase, can be implemented in a future step of the project.

The generation of virtual code represents the final state of the compilation process and is also implements modifications of the intermediary code instructions (perhaps optimized) in machine instructions (or assembly code) for the designated computer [4].

The generator of virtual code presents more main functions:

- Instruction selection
- Ordering instructions
- Registry allocation

In this project, because of the object oriented nature of the ESPL language, the generator of virtual code also implements functions for loading the classes into memory and generation of routines for object allocation, and their linking with the source program.

Therefore, this step of compilation is strongly dependent on the target system architecture, in this case, the virtual machine, that is going to be described in the following paragraph.

D. Simulation of the Virtual Code

In the microcontroller domain [6], because the real-time systems are in general only execution platforms, in the development phase, their emulation is a great help for testing, analysis and debugging the resulted code.

Based on these considerations, for simulating the execution of the virtual code, the solution of defining a virtual machine represents a good choice. In this thesis, the VM is designed to be very customizable and can emulate, if necessary, at a

generic level, the architecture and functionality of some types of real embedded systems.

In essence, a virtual machine is equipped with a virtual language (code) and also with an executive routine, which is the interpreter for this virtual language. The role of the executive is to pass through all the virtual code and execute each instruction in terms of machine code of the desired computer.

The virtual language and the structure of the virtual machine are settled after the following criteria [7]:

- Virtual commands should be sufficient to allow the correct translation of the instructions from the source language;
- Functionality of the virtual machine should be easy to emulate with the help of executives for each object language.

Generally, the virtual language is inspired from other assembly languages; however it's showing a greater abstract level [5].

III. The Scanner Generator

For the development of the compiler we decided to use scanner and parser generators that we developed as part of this project.

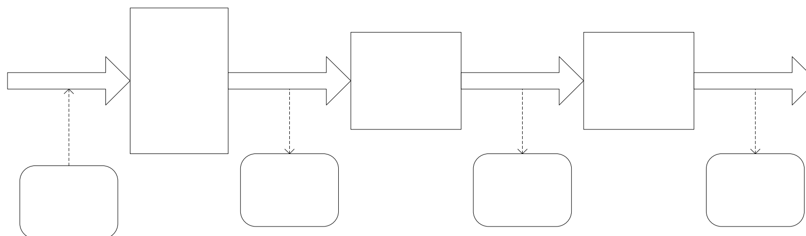

Fig. 1 The structure of the scanner generator

The input of the scanner generator is represented by a text file that contains a set of regular expressions which describes the source language atoms and generates the corresponding transitions table and the C++ code of the scanner.

The steps of generation follow below (fig.1):

- the generator creates first an explicit abstract syntax tree corresponding to the regular expressions set from the input file;
- then it creates the corresponding non-determinist finite state machine (NFSM) based on the abstract syntax tree from above;
- finally, the generator transforms the above NFSM into the equivalent determinist finite state machine (DFSM) represented by a transitions table and generates the C++ code of the scanner.

IV. The Parser Generator

The input of the parser generator is represented by a text file that contains the productions of the grammar and generates C++ code corresponding to the syntactic analysis as well as to the construction of the complete syntax tree.

We have chosen recursive descent parsing because it is best suited for an object oriented design. The syntax tree is complete because for each type of non-terminal in the grammar it is generated a new class of nodes.

Each of the functions that are produced by the generator return a certain type of node. The children of the node created inside a function are constructed through recursive calls to other functions of the parser.

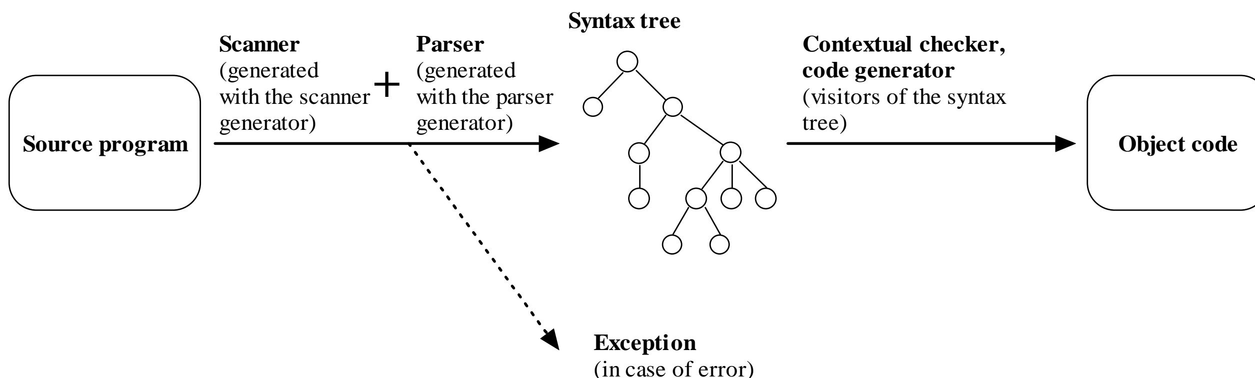

Fig. 2 The phases of a compiler developed using the generators

The parser generator produces C++ code that can be used together with the C++ code produced by the scanner generator.

The user of these tools will be working with the complete syntax tree constructed by the automatically generated parser to realize the other phases of a compiler. Fig. 2 shows the phases of a compiler developed using the two generators.

The generators were used at the developing of the Object Pascal compiler, but also at the developing of an assembler for the virtual object code.

V. THE DESIGN OF THE COMPILER

In fig. 3 we present the structure of the compiler.

The *abstract syntax tree* corresponding to a program includes two types of nodes: internal nodes, corresponding to the non-terminal symbols of the grammar; and leaf nodes, corresponding to the terminal symbols [8].

Both *contextual analysis* and *code generation* are realized through sequential traversals of the abstract syntax tree. The two complex operations are represented through the *visitor* design pattern. The two visitors are the contextual checker and the code generator. This pattern makes the abstract syntax tree structure independent of its operations.

The *symbols table* is the main structure of the compiler. It is implemented like a hash table. The table contains objects of different classes, each object corresponds to each kind of identifier (variables, constants, etc.). The symbols table is constructed during the contextual analysis [9]. Both contextual analysis and code generation use the information about identifiers contained by this table.

The *errors* reported by all phases of the compilation are treated through the exception mechanism from C++. The way of treating errors doesn't allow the recovery in case of errors, meaning that the compiler stops its execution when the first error is detected.

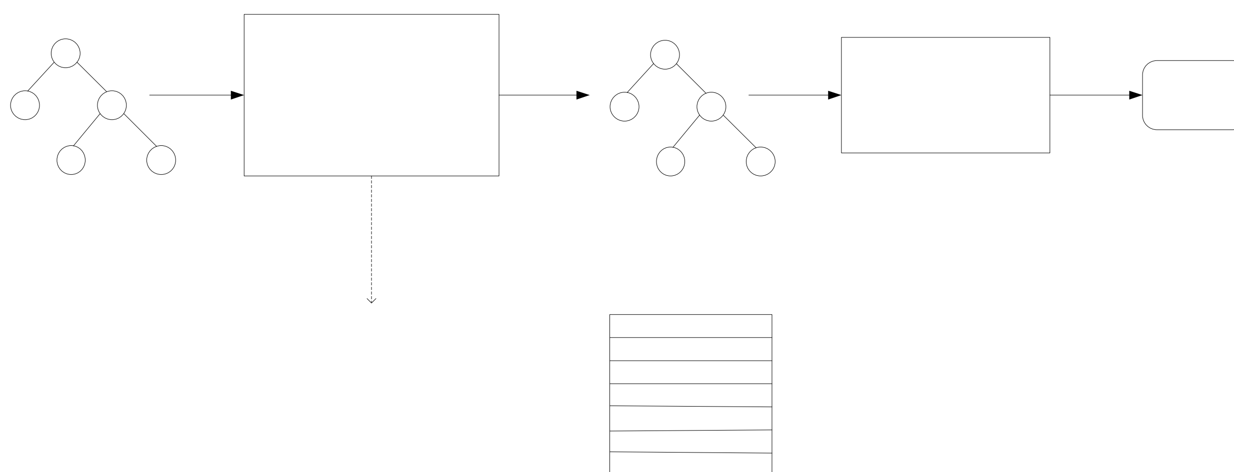

Fig. 3 The structure of the compiler

VI. THE VIRTUAL MACHINE - VMPC

The solution of a virtual machine [4] was taken based on initial purpose of a virtual machine – a hardware emulator for real-time systems.

Nowadays, virtual machines have acquired new roles and developed new uses and two big branches have emerged:

- System Virtual Machines, with the purpose of providing a complete execution platform within another one.
- Process Virtual Machines, with the purpose of introducing a new level of abstraction and the possibility of creating platform independent applications.

VMPC – Virtual Machine for Personal Computer – is designed as a hybrid virtual machine, but with the balance in favor for the System Virtual Machine class. It introduces the level of abstraction and the code can be run on any platform if the virtual machine is implemented. But, in the current implementation, the virtual machine is very complex, because it includes a series of customizations, and, strong debugging and testing facilities. This is why it represents more a System VM, than a Process VM. Still, with a simpler implementation of the default VM, other platforms than the PC could be targeted (even real-time platforms) and a good portability could be assured [9].

A. The Virtual Machine's Architecture

As stated before, the virtual machine VMPC is designed to offer strong customization capabilities. In order to accomplish this, the VM was build upon components that can be replaced or extended at any moment. In addition, each component has

its own customizable characteristics, as shown in the following paragraphs.

The main components are [7]:

- the instruction set
- the register set
- the data memory
- the code memory
- the port set
- the bus controller

These components, like a real system, can be interchanged – modified or extended – to create different configurations because they have a high level of independency.

To ensure the independence and substitution capabilities, the components are designed independently based on interfaces, in concordance with the OOP principles, encapsulating almost completely the actual implementation [7].

Still, because it emulates a real system and the components are working together, the components have to obey certain design conventions that are documented for every type of component. Good examples are the constants used in the instruction or register modules so that other modules can access a certain entity without knowing the internal structure or implementation.

Also, some functions from a component might depend on a specific implementation from another component. For example: the usage of the retrieval command for the current value of the timer register (GTM), without having the specific register defined in the associated set of registers. Although the current implementation has some protection measures against inadequate uses (by treating the exceptions that may appear), the behavior would still be incomplete and should be avoided. In the case of the above example, VMPC generates an interrupt with the default action of ignoring the instruction.

To minimize the cases of incorrect configurations, the MVPC instantiation is realized trough an abstract factory which registers the valid configurations for the available VM modules.

The valid and the user defined (preferred) configurations are stored in external files, so they can be saved and loaded directly from the UI.

To ensure a simple substitution between different virtual machine modules, all the virtual machine implementations available to ESPL are stored in the package *emulators*.

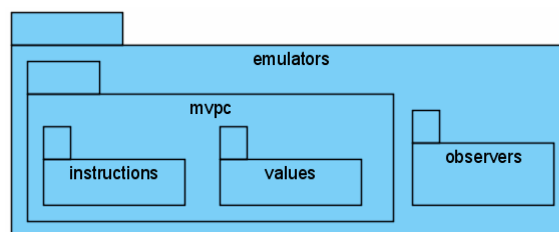

Fig .4 The virtual machines package structure

As it can be deduced from fig. 4, the default VMPC is in *mvpc* package. The other packages that appear in the figure correspond to some of the VMPC's components and will be described along with the parent component (fig. 5).

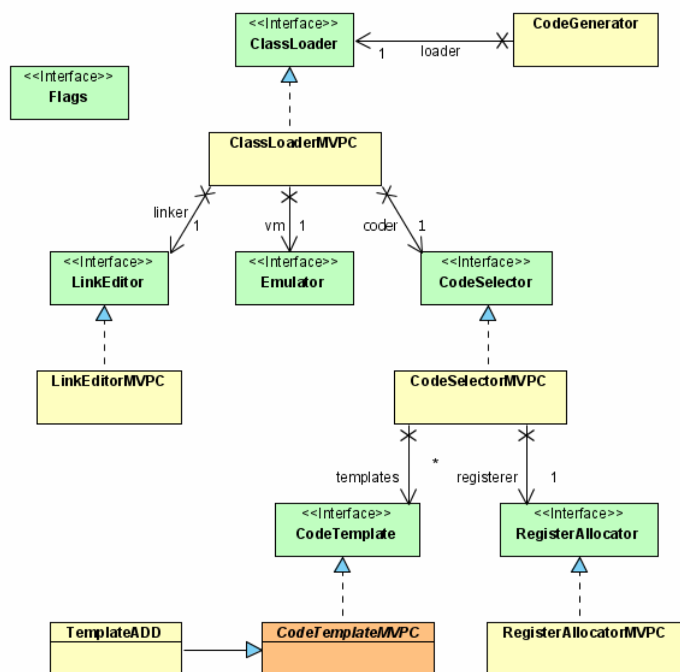

Fig. 5 MVPC diagram classes

B. The data representation

To bring a new layer of abstraction bound with the representation of data, an interface “Value” is used. Value classes include management and conversion functions for the r epresentation of data in the virtual machine’s components.

For the default implementation of VMPC have been defined 3 sets of data, for 8, 16 and 32 bit data. All the available datasets for VMPC are stored in the package *values* from package *mvpc*.

For every dataset, three fundamental types of data are supported in the current implementation: integer, real, address (unsigned integer). The dataset’s length gives also the type’s length.

The real type is based on the floating point standard IEEE 754 and is only available for 32 bit data. Implementing real values for less than 32-bit is possible, but it will introduce a number resolution, that makes them almost unusable. So, for 8 and 16 bit data, float data are truncated and stored as integers.

As support for these types, the value classes include functions for conversion (between data type) and comparison (between values of the same dataset).

Regarding data storage, the virtual machine is equipped with both registers and stack. This way the global variables are statically allocated and local variables are allocated in stack frames.

There is no direct support for the heap in the default implementation, so heap variables and dynamic arrays require manager modules, written in virtual code, that are going to be linked with the application’s virtual code (in virtual code generation phase, described in the following chapter).

C. The Instruction Set

The instruction set is the one that stores and manages all the instructions known by the virtual machine. The default implementation contains a number of 88 instructions.

Each type of instruction contains its own characteristics, coding (used for virtual / assembler code generation) and decoding methods, and its execution method.

The instructions are registered in the set of instructions and in this way subsets of the implicit set can be defined, or the set can be extended with new instructions, therefore leading to the construction of simpler or more complex virtual machines.

Besides the instructions, the instruction set contains also the executive. This represents the execution engine of the virtual machine. It simulates clock cycles and the processing unit of a real system. The routine polls the machine status and takes corresponding action if required, manages the internal timer, and decodes and executes instructions.

The default execution engine is equipped with a simple pipeline, with the following functional units:

- Instruction Fetch Unit
- Arithmetic Logic Unit (ALU)
- Floating Point Unit (FPU)
- Memory Access Unit
- Port Access Unit

Each of the functional units can have a customizable latency.

The instructions are broken into 2 phases, the fetch and the execution.

The fetch (and decode) is only made by the Instruction Fetch Unit (considered without latency), so in case of a balanced instruction load and/or small latencies for the execution units, a throughput of 1 instruction / clock cycle is obtain at best.

The execution is done by the other 4 units based on the instruction type. [6]

No “hardware” workload balancing is done in the current implementation, so this task has to be done by the compiler, most probably at the optimization phase (currently not available). In these conditions, the balanced load is not guaranteed by the virtual machine.

D. The Register Set

The register set contains all the available registers of the virtual machine.

Registers are of three types: administrative registers, general use registers, specific registers.

The administrative registers (like IC – Instruction Count) are hidden from the user, by being used and managed internally by the instructions to assure the base functionality of the machine.

The general use registers have associated a specific code and can be referenced by the user as parameters of instructions. There are two types of general use registers: for integers and for floating point operations. The numbers of these registers are among the selectable characteristics of the register set, the two values can differ.

The specific registers are associated with embedded special features, like timers, convectors etc. These registers are optional and their presence is also among the selectable features of the register set.

The registers, as in the case of instruction sets, are recorded in the register sets, allowing the extension or reduction of the set in use.

The dimension of the data in the registers is set by giving a prototype value (from the available datasets) at the initialization of the set.

The size of the integer registers in the default implementation determines the maximum addressable memory, because they are used for address resolution.

The floating point registers, in the current implementation, are locked to 32-bit, because of the data constraint presented in paragraph B.

E. The Memory

The memory is implemented using an array of values, with the data size specified by a prototype value, just like in case of the register set.

Besides this, the memory has the following characteristics: memory size (in number of cells), latency and its representation policy.

The latency represents the number of clock cycles needed for data access. A latency of 0 is also accepted and it means that any kind of memory access from a certain instruction is done within' a clock cycle (even if multiple accesses are necessary).

The representation policy can be big-endian or small-endian and it's used by the memory controller to make a correct transfer between data from memory and other components with different data size or bus.

In VMPC there are two types of memories used, a data memory and a code memory, but they can be set to be the same, in case the system doesn't have separate code and data memories. In this case the base segments have to be set correctly so that the two areas won't overlap.

The maximum size of the memory that can be used in the current implementation is 4GB, but in practice only about 16 MB are accepted because of the memory issues from the host system. To overcome this problem, a swapping system has to be designed for testing a system with more than 16MB of memory. At this point this module wasn't created.

F. The Port Set

The port set contains and manipulates all the ports that the virtual machine has. The ports, like the other components, have several properties that can be set when the set is instantiated.

First of all it's the type. There can be input ports, output ports and bidirectional ports. Incorrect actions on a inappropriate type of port (a read from an output port) will lead to an interrupt for I/O errors. To prevent I/O errors like this, pooling functions are included that tells the system the type of a certain port.

Like in case of the memory, the port set has the possibility to set the data size and the latency.

A special feature is that they are equipped with data buffers to prevent stalls due to poor synchronization, communication failures or other connection problems. The size of this buffer is also a selectable property. But even with buffers, the port could become busy and so, pooling functions are implemented in this case too, so that the system knows when the port will be accessible.

G. The Bus Controller

This is a support component that takes care of the communication between the other components.

It has conversion functions for the data types used by the virtual machine and special functions for component -> component communication [10]. So, it is responsible to correctly transfer (copy) data of different type and size among the virtual machine's components.

It is recommended that all the new components use the bus controller for communicating with other components, so that the conversion and representation standards will be followed.

H. The Interrupt System

The interrupt system is responsible for managing the interrupts and errors of the VM.

The interrupt system status is polled at the beginning of each virtual clock cycle and if an active request is waiting, the corresponding handling routine is called.

Interrupts can be originated from the system – system interrupts – or from the user (from the running application) – user interrupts.

The interrupt system of the default implementation of VMPC treats the system interrupts and the user interrupts the same. The system interrupts are defined through constants and depend on the implementation (they have to be respected by the application developer).

The interrupts are prioritized. The priority of a certain interrupt as well as the interrupt handler routine can be changed by the user. A running interrupt handler can be interrupted only by an interrupt with higher priority.

The number of interrupts of VMPC corresponds with the size of the integer registers (16 interrupts for a 16-bit integer register set).

I. The Machine Observers

Observers, like stated in the corresponding design patter, are objects that are attached to the virtual machine, have access to the machine's state and internal operations and are notified every time the machine's status changes.

This way they can react in various ways and can be used to implement other functions, like polling functions, debugging modules etc. without the need of rebuilding the VM.

These observers can be dynamically registered and removed from the VM at runtime.

VII. Virtual Code Generation

A. The Code Generator's Architecture

The code generator [4] is implemented using the data structure generated by the compiler's front end modules, with some small improvements. The changes were necessary but were omitted in the code generation phase because they weren't of interest at that time. For example the access modifiers volatile and static were omitted because the intermediate code doesn't use registers. But these changes can prove useful for that stage too, making the intermediate code more powerful, consolidating its role as generic assembly language.

The code generation is highly dependent on the target VM and that's why this module uses the virtual machine configuration and components in order to provide specific code for that machine type.

The result of the virtual code generation is an object file written in machine code. The code generator has the possibility to write user friendly code using the description of the instructions, and thus, being able to generate assembly language code. This feature is also useful for debugging, verification and validation and, possibly for profiling.

In this implementation, the code generator also takes care of the class loading and the generation of the heap and dynamic arrays manager's codes.

Code generation modules are located in the *codgen* package, each with its own package – the default code generator for VMPC has *mvpc* package.

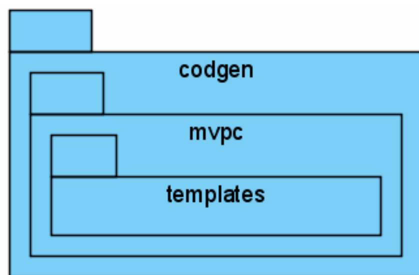

Fig. 6 The virtual code generator's packages

The module components are:

- The code selector
- The register allocator
- The class loader
- The link editor

The address resolution is done by all the components, so a library of functions is created for this purpose. But, because it's only for support, it is not considered a separate component.

B. The Data Storage

In the current implementation, because in ESPL there are no global variables, statically allocated will be only the data resulted from class loading and the variables needed by the specific managers.

The local variables will be allocated in the stack frames, so addressed relative to the SB register.

The dynamic tables and objects will be allocated in the heap by the corresponding manager routines.

The heap and the stack will start at opposites ends for better memory utilization. The heap will start immediately after the statically allocated data and grow ascendant, while the stack will start at the end of the program's memory and grow descendant. This way the system will be out of memory only when the two areas collide.

Dynamic arrays are used instead of static arrays because ESPL doesn't provide a mechanism for setting the array lengths, so they have to be allocated dynamically. This way they will have a manager that along with the heap manager will manage them similar to objects.

C . The Code Selector

The code selector is in charge of translating the instructions from the intermediate form, provided by the front-end, in corresponding virtual code. This module works closely with the address resolver, the class loader and the register allocator to generate de instruction's parameters.

The translation is done upon code templates that are applied to an intermediate form instruction or a set of intermediate form instructions, so the generated object code is as good as possible [4].

In the current implementation, the translation is done almost directly, because the two instruction sets are pretty similar. There are only loading and data transfer functions added to the virtual instruction set, that don't have a corresponding instruction in the intermediate instruction set.

This module also takes care of the address calculation of the instruction destination used in jumps and calls.

D. The Register Allocator

This module manages the allocation of registers needed for storing variables and intermediate results in expression calculation.

It has to select the best fit data to be stored in specific registers and which to be spilled in the stack. It also has to take in account the variable's access modifiers, volatile and static.

The selection is based on the constraints and a usage frequency indicator (LFU – least frequently used – replacement technique) in the current implementation. [4]

E. The Class Loader

In the default implementation, inspired by Java's class loader, the code generator's class loader is the main component. It starts and manages the code generation process.

Functionally, the class loader crosses the intermediate type structure and allocates the memory necessary for class objects and object prototypes (that will be used by the heap manager to instantiate objects of a specific class).

It's roles are:

- parsing the input data and identifying classes and methods;
- calculating the real addresses for every class object (descriptor);
- calculating relative addresses for static and instance attributes; and methods;
- calling the code selector and link-editor components when needed;
- building the output data;
- writing the output files.

F. The Link-Editor

This component is responsible for loading and adding the code for the heap and dynamic arrays managers. It will also collaborate with the class loader to allocate the necessary data and variables used by the managers.

The corresponding routines are written in object code and they are read from a file and then link-edited with the current program.

Because the object code for the manager required data for address resolution, in the external object code file for the managers, standardized constants are used, that will be replaced by the link-editor with the required value. This way the managers can be "upgraded" in a very simple manner.

This module is also responsible for adding the default interrupt handling routines. In the current implementation, these routines will ignore external interrupt requests and halt the system in case of internal error interrupts.

G. The object file

The object file is composed out of 3 main areas:

- the memory data area, that is composed of the data generated by the class loader;
- the program code generated by the code selector component;
- a general information area, that is generated to identify the virtual machine configuration that the file is destined to.

The areas are entered in a TLV format (Tag-Length-Value), where the Tag is 1 byte long, identifying the area, the Length is 4 bytes long and gives the total length of the area and the Value contains the area data with the specified length.

The information area can be missing, but running a file on a machine even with slight configuration differences, rather the one destined to, could bring a defective execution.

VIII. CONCLUSIONS

The added modules are substitutable, because of the combinations of the concepts of inheritance and composition, and because of the use of several creational design patterns like Prototype and Abstract Factory.

An important role is played by the Observer pattern that offers many external actions like analyses, measurements, debugging etc.

The components are not only substitutable, but also very customizable because of the many attributes that can be set, and so, even the default implementation of the virtual machine, offers many usage approaches, by decently simulating specific classes of embedded systems.

An important aspect is that, through its complex structure, the virtual machine is not suitable to be used as a virtual machine on a specific platform, although the Java environment offers great portability. This is because it was designed especially for simulation and testing on development platforms.

Also with the possibility of generating assembly code, with the correct configuration of the instruction set and the usage of a cross-compiler on the resulting code, real machine code can be generated.

As a testing strategy, most black-box testing was used. Unit tests were created for the most important components and scenario testing for the whole system.

A minus in the current implementation is the error reporting in the virtual machine. Improving this module would be a good direction for future developing.

REFERENCES

[1] H. Ciocarlie, “Programming Language for the Development of Distributed Real-Time Applications Dedicated”, IMECS 2007, *International MultiConference of Engineers and Computer Scientist*, Hong Kong, 21-23 March, 2007.

[2] Ph. A. Laplante, “Real-Time Systems Design and Analysis” (Third Edition), *Wiley-IEEE Press* 2004.

[3] H. Ciocarlie, L. Simon**,** Definition of a High Level Language for Real-Time Distributed Systems Programming, *The IEEE Region 8 EUROCON 2007 Conference*, Warsaw University of Technology, Poland, September 9-12, 2007.

[4] D. A. Watt, D. F. Brown, *„Programming Language Processors in Java. Compilers and Interpreters”,* Preatice Hall, 2000.

[5] D. Grune, Henry E. Bal, Cariel J.H. Jacobs, Koen G. Langendoen, *Modern Compiler Design,* John Wiley & Sons LTD, 2000 (rev. 2003).

[6] M. Barr, *Programming Embedded Systems in C and C++* (First Edition), 1999.

[7] T. Lindholm, F. Yellin, *The Java™ Virtual Machine Specification,* Addison-Weslez, 1997.

[8] M. Bouhdadi, E. M. Chabbar, H. Belhaj, “A Meta-model Semantics for Structural Constraints in ODP Computational Language”, *WSEAS Transactions on Computer*, Issue 2, vol. 6, February 2007, pp. 341

[9] H. Ibrahim, B. Zaqaibeh, A. Mamat, Md. N. Sulaiman, “Enforcing and Maintaining Constraints Base During the Compile-Time”, *WSEAS Transactions on Computer*, Issue 2, vol. 6, February 2007, pp. 373

[10] C. Tianzhou, H. Jiangwei, Qianjie, Liangxio, “Using Dynamic and Static Approach with Compiler-Assisted in Power Management for DVS-Enabled Embedded System”, *WSEAS Transactions on Computer*, Issue 6, vol. 6, June 2007, pp. 967

Horia Ciocarlie was born in Timisoara, Romania, on March 9, 1953. He graduated in 1976 the Politechnic Institute of Timisoara, the Faculty of Electrical Engineering, with a degree in electronic computers. He received his Ph.D. in 1991 in field of Computer Science.

He is working at “Politehnica” University of Timisoara since 1977. In 1995 he became a UNIVERSITY PROFESSOR at the Faculty of Automation and Computers, Computer and Software Engineering Department of the above mentioned university. Among his published books there are: *The programming Language Universe*, Orizonturi Universitare Publishing House, 2006 and *Fundamental Concepts of Programming Languages*, Vest Publishing House, 2007 and among his scientific papers there is: *Definition of a High Level Language for Real-Time Distributed Systems Programming,* The IEEE Region 8 EUROCON 2007 Conference, Warsaw University of Technology, Poland, September 9-12, 2007. His research interests include: programming languages (definition, implementation, programming techniques) and distributed system programming.

Prof. Ciocarlie is a member of ACM, IEEE and Computer and International Society for Engineering Education.

Cosmin-Mihai M. Vacarescu was born in Caransebes, Romania on September 22, 1984. Cosmin obtained his bachelor degree in Computer Science in July 2008 from the Faculty of Computer Science, the “Politehnica” University of Timisoara, Romania.

He works at ACI World Wide EED Timisoara, Romania since June 2007 as an Associate Engineer (Lever 3 Support for ACI’s BASE24-EPS Payments Engine Solution). He has conducted research for Embedded Systems Compilers for his diploma thesis.

Not part in other professional societies and no awards received.