Embedded System Modeling and Synthesis in OO Environments. A Smart-Sensor Case Study

Germán Fabregat & Germán León
Universitat Jaume I
Campus de Riu Sec
12071 Castellón, SPAIN
{fabregat,leon}@inf.uji.es

Bernard Pottier, Olivier Le Berre* & Loïc Lagadec
Université de Bretagne Occidentale
20 avenue Le Gorgeu
Brest, 29287, FRANCE
pottier, llagadec@univ-brest.fr
(*) now with Alcatel CIT/Network Management Unit
Route de Nozay, 91460 Marcoussis, France
olivier.le-berre@nmu.alcatel.fr

Abstract

Efficient application development for complex hardware systems is becoming more of a challenge everyday. Reduced time to market and specificities of hardware are putting severe problems into programmers. In this context, automatic generation of code based on object oriented models of the target systems reveals as an efficient solution both to obtain high quality applications and to reduce effort and thus cut development time. We present a case example for this idea, where two different architectures are targetted by a Smalltalk-80 development system. We present the basics of code generation supported by the system and the main aspects to be considered when designing such a development platform. We describe the target platforms and give examples of the results obtained.

1. Introduction

Integration technologies provide more and more resources and functionalities to new devices. Among them, we can mention:

- increase in speed and gate count,
- added flexibility: reconfigurable technologies,
- intelligent data processing: smart sensors.

As a result there is an emerging need to define and synthesize architectures for families of new embedded applications with demanding requirements, mixing different computation models: parallel processing, planar acquisition, intelligent sensors and actuators, networking…

On the other hand, software and system integration requirements are also quickly increasing. The simplest device will need support for communication and distributed control, high level application software and development tools, standardized data exchanges.

To add more pressure on the industrial designers, time to market of designs including these new features is becoming shorter and shorter.

In this context, there is an urgent need for methods that provide: short hardware and software development paths, easy composition of computation models, data interoperability.

This paper uses a workbench for a vision smart sensor to describe a methodology that allows the rapid development of applications for different system architectures. The whole workbench is developed using a single object environment. The functionalities of the sensor are addressed at the instruction set level. To rapidly fit different systems we have used object oriented modeling to produce metacompilers that start from a high level description of the target platform to create compilable code suited to it. The environment has been tested on two different platforms:

- a remote embedded system having its own microprocessor, the smart sensor and communication support,
- a laboratory development platform where the smart sensor is accessible on a parallel (ISA) port of a PC.

At the lowest programming level, the sensor appears as an object model (architecture) understanding messages close to the instruction set, while non specific code (messages) is addressed to the host processor. At an intermediate level, we build application level objects (data) suitable for remote control or debugging. These objects can be used to develop and test high level application programs. An objective is to avoid lost of semantics due to a layered software architecture. Another objective is to use object oriented programming and technology in a vertical way, down to hardware specification, to obtain fast and secure development methods.

The paper presents the target architecture and platforms. It provides a short discussion about Smalltalk-80 characteristics and their relation with typing. The programmer model for the remote camera is a composition of hardware and system components such as: a specific circuit, a sequential thread, communication activities, FPGA control. The paper describes the architecture of the general compiler and some technical points on the translation. Then we present the implementation on the development platform. The last section of the paper presents current research directions including logic synthesis, reconfigurable architectures modelling and support for interoperability. We conclude with practical issues and results.

2. The MAPP2200 smart sensor

The *Matrix Array Picture Processor* MAPP2200 smart sensor [1] combines on a single chip a sensing array of 256×256 photodiodes and several SIMD processing units to be suitable for control systems based on the near-sensor image processing paradigm [2]. Although it is able to decode and execute its own instructions, it does not include a sequencing nor a program flow control unit, thus needing the operation of external logic for these tasks.
The programming model offered by systems using the MAPP always includes a parallel SIMD execution core and a sequential control. This model is usually extended with further sequential or parallel processing. MAPP based systems are a perfect example to realize the difficulty of programming each different target architecture, and to show how an approach from high level specification plus low level architecture models is the best solution for the cost-effective production of target code.

3. Smalltalk-80, object scripting and typing

Smalltalk-80 has been chosen as the optimal environment to develop the proposed methodology for System modeling and programming. The most important characteristic, unique to most popular OO languages (Java, C++...) is the absence of typing, which easily allows handling high level specifications isolated from particular target details. Other important aspects are the inclusion of C and Smalltalk (or yacc-style) parsers in the basic system, greatly helping the analysis of high level code and the generation of target software.

3.1 Language main characteristics

Smalltalk-80 has been designed to address the complexity of evolving environments for future workstations at Xerox PARC. To be short, the language has a simple syntax, with no control structure.

Sequential or concurrent control is described in classes whose instances can respond to particular messages: *Boolean* implements conditional execution in response to *ifTrue:* or *ifFalse:*, *Integer* implements *timesRepeat:*, *Collection* are able to sequence an execution specified by *do:*, *collect:* ...

The *block* code structuring mechanism does not imply an immediate execution at *blocks* definition. They are commonly passed as parameters in messages whose receiver decides if the execution will take place or not. *Blocks* are objects that enclose a sequence of instructions, plus a referential environment.

Messages are understood by instances from a family of classes. They implicitly refer to a functionality or an action having the same semantic on this family. As an example, every collection knows how to iterate on elements in response to the message *do:*, every number knows how to compute its square in response to the message *squared*, every object produce a text in response to the message *printString*. Programmers know the semantics behind messages. They can reuse message names for new classes that can share some behavior with existing ones. They can also retrieve existing functionalities using their knowledge of the system.

Another important point is the fine grain modularity in the system. Classes are grouped in categories that correspond to "kernels" having specific data organizations and functionalities. The message interfaces to these kernels allow a systematic and secure reuse of existing software. As an example the *Collections* form a generic support that is reused everywhere, that implements proved and efficient algorithms, that has a clear, complete and orthogonal interface. Smalltalk-80 data structures work securely for existing classes, and will work for future classes as well.

3.2 Types

Smalltalk-80 only relies on classes to decide how a message will be executed. At run time, the system searches the class of a receiver, looks up in the inheritance tree for the message selector, fetches and executes a method (code for a message implemented in a particular class). Dynamic binding allows to use an existing algorithm or data organization on newly developed classes. However it implies an overhead at run time for method searching. This is the cost for extensibility and message selector sharing.

Instead of using a language coercing in the wrong direction plus a too much vague methodology, the choice was to use a language that excels in modeling and to work on additional constraints allowing to produce code and later, hardware description. See [3, 4, 5]for related work.

4. Compiling to C and MAPP instruction set

Smalltalk code is implicitly sequential although some messages on collections have parallel semantics. Smalltalk methods are sequences of messages sent to receivers. The programmer focus his attention on these receivers and knows precisely their class or their behavior.

The system can be viewed as a set of objects having particular functionalities: sequential processing, synchronization, communication, specific processing… The major difference with normal Smalltalk computations is that additional constraints (typing) allow to translate into microprocessor code, operating system calls, instructions, hardware controllers and more…

System synthesis separates models from each other, enabling code generation for each component. It produces intermediate descriptions for the data exchanged among the low-level entities, as well as interfaces (object descriptions and exchange procedures) between the low level and the applicative or development tools.

The core of the translation process is a Smalltalk to C mapping.

4.1 Sequential model

4.1.1 Type mapping

The C language has been designed for system programming efficiency on the processors of the 70s, and has remained as the standard model for sequential program execution. Using C as an intermediate language provides basic utilities to manage memory and microprocessor resources in a portable way.

In order to produce C code out of Smalltalk methods it is necessary to provide a mapping between some classes and C simple types. There exists such a mapping in CORBA IDL specification [6], but we were mostly interested in building a top-down system generator with possible hardware support for which CORBA mechanisms are too complex.

C simple types appear as translations of classes (or subclasses) Integer, LargeInteger, Boolean, Float, Double, String, CharacterArray…

C arrays correspond to classes having a single type for elements (IntegerArray, FloatArray …). Structures correspond to the aggregation of instance variables declared along an inheritance tree.

4.1.2 Syntax for type declaration

When generating C code from Smalltalk, private variables appear in 3 situations: instance variables, temporaries and method parameters. The current version of the compiler uses class comments and standard method annotations for type declarations. Class comments have no formal syntax. However there is a consensus on the declaration of the preferred class for each instance variable declared in the class.

For each referenced class, the translator does an upward traversal of the inheritance tree and accumulates data descriptions. Later these descriptions will be rewritten into C structure declarations.

4.1.3 Code analysis

Additional tools allow to structure the Smalltalk class descriptions to produce libraries, C declaration files, or executable programs. The core of these tools is the standard Smalltalk compiler extended to support multi-target code generation.

The scanner produces a program tree using subclasses of *ProgramNode* for methods, variables, statements, assignments… The standard compiler provides support to recursively process this tree by subclassing the class *ProgramNodeEnumerator*. A new tree is built that holds C nodes equivalent to Smalltalk ones for assignments, blocks, literals, parameters, variables, returns and sequences of statements. Methods are mapped to C function definitions. Messages are transformed into an intermediate *CMessageNode* node for further processing.

Message nodes are translated into a variety of C operations, iterations, tests, function calls, as well as specific model processing.

The example method below comes from the class `MappExample`. This is the method for a polling loop controlled by the `time` parameter:

waitLoop: time
<resource: #(#(Integer i time))>
time timesRepeat: [0 to: 7 do:[:i| ”do nothing”]]

This method is transformed into a function whose name starts with the class name. Parameters are always passed by addresses and the receiver is passed as the special field *self*:

```
void MappExample_waitLoop(self,time)
MappExample *self;
int (*time);
{  int i;
   int i3;
   for(i3=1;i3<=(*time);i3=i3+1)
      { for(i=0;i<=7;i=i+1) { } } }
```

4.2 Software interface for the MAPP

Variables that are instances of the MAPP class or subclasses are recognized by the compiler. When a message is addressed to these variables, a translator is built that holds the node environment. There are particular translator classes for each of the MAPP instruction set groups. A code generation functionality is built in these classes, that the compiler can use to get a textual C representation corresponding to the Smalltalk message.

The MAPP programming interface is a set of messages corresponding to the basic instructions with variants and parameters. The whole instruction set has been developed in a very short time due to the regularity existing between the software model and the actual instruction decoder.

C code generation is the choice for the remote camera. The program will access the MAPP using an FPGA that can lock the system bus to obtain a synchronization with its internal state. The representation of FPGA control operations is handled using C pointers at constant addresses, plus functions allowing to reset or load configurations.

The following example is part of a method from class MAPP that sends a move message to the sensor. The symbols #ad and #r denote the conversion result and the working registers files. The call execute 8 consecutive transfers indexed by the *y* variable:

sobelTransform
<resource: #(#(#Integer #y))>
0 to: 7 do: [:y | self
  store: #ad number: y in: #r number: y]. "etc."

The C translation is given below. `FPGA` denotes a pointer on an FPGA port that maps the MAPP instruction port into the microprocessor address space:

```
void MappExample_sobelTrans(self)
MappExample *self;
{
int y; /*...*/
for(y=0;y<=7;y=y+1) {
*FPGA = (0x800+((0x0+y)<<7)+(0x0+y))<<10;
} /* etc.*/
```

4.3 System resources

System programming can use existing software or hardware resources defined as C libraries or data. As the system synthesis comes exclusively from the object specification, it is necessary to integrate these resources as classes. The Smalltalk environment allows to parse external languages to represent external programs as instances of the accurate classes. The support for C is more complete since the DLLC tools enables the automatic production of interfaces for ANSI C dynamic libraries.

The modified compiler detects these particular classes and generates the appropriate C code to call these functions. This is used for FPGA reconfiguration, communication library, etc... The programmer view for these classes is a set of well identified functionalities that need a creation message, plus a set of messages to obtain services. The following text shows the process of attaching to the camera kernel for message passing service, then FPGA reconfiguration and reset operations:

...
fpga := Fpga new. "FPGA control class"
mesgPassing := CommunicationService new.
image := GraphicImage new.
mesgPassing kinit. "attachment to the kernel"
fpga ConfigureFpga: 'someFile.conf'.
fpga ResetFpga. "... Image production ..."
mesgPassing sendImage: 256 with: 256 with: image bits

To ease the transport from the embedded system to the external control or development tools, each class has its own serialization functions that describe the object on a stream, or reconstruct an object from a stream. The similar functionality exists on the high level client platform.

5. Recompiling for the MAPP2200 PC camera

The generality of the proposed approach has been proved by targeting a second, much simpler platform. The same high level Smalltalk code is now compiled into a different class to automatically produce working target code. The new class includes only coded instructions sent to the MAPP by the Smalltalk-80 virtual machine, defining portability at the functionality level (Sobel filter).

5.1 System description

This second system is the MAPP2200 PC System [1], including a MAPP camera that communicates with a PC using a very simple ISA board. This board offers three PC ports to communicate with the MAPP: one for sending instructions and reading status, other for reading and writing data to the serial port, and the third for control tasks.
The system has been tested under Windows95, using the VisualWorks Smalltalk environment, from ObjectShare. Communication with the ISA board has been performed through the TVicHW32 library, that offers a C interface to a DLL for accessing PC physical resources. It is worth noting that the interface between the Smalltalk environment and the C language is so well designed that no C code has been developed, and Smalltalk programs accessed directly the DLL interface.

5.2 Generating code

Code has been automatically generated by compiling Smalltalk methods in the MappPC class. Now the compilation tree is generated and analyzed, but only leaf methods accessing the MAPP hardware (MAPP instructions) have been translated into DLL calls to access the required ISA ports. The same example coming from the sobelTransform is presented below. Now Smalltalk code remains unchanged, and only the message *store:number:in:number:* has been changed for the message *writeMappInstruction:* that access the DLL and writes to the adequate port. As instruction coding depends on the registers used, in the Smalltalk system the translation is done on the fly by the *code* message:

```
sobelTransform
0 to: 7 do: [:y | self
  writeMappInstruction:
    ((MappStoreTranslator new)
      src: #ad → y; dest: #register → y) code]
...
```

6. Status and perspectives

Ongoing work related with the described methodology includes logic synthesis for objects, modelling and tools for reconfigurable architectures, methods and tools for automated data handling and further, program translation.

These works can be seen as research on the *application-systems path* sharing objectives such as: flexibility, speed for development, inter-operability based on object technology. The general idea behind is to avoid “blind interfaces” in the development tools and to build models providing control all over the path.

6.1 Logic synthesis

Logic synthesis has proven to work efficiently for Smalltalk-80 *blocks* having a small number of input/output signals. The idea is to use *blocks* as the lower level execution grain as an alternative to technology libraries. This idea presupposes the existence of other tools for architecture synthesis and floorplanning.

As for C synthesis, a *type* specification for the block variables and parameters is required. Types for logic synthesis are *collections* of *values* (Smalltalk-80 *Interval, Set*... including objects). Each object appearing as a value must be able to represent itself as a bit string. Therefore, there is no need to restrict binary representations or to use a fixe set of simple data types.

The collections of block input values are exhaustively sweept to produce a look-up table that still contains objects. Then this table can be folded into a PLA using binary translators, and the SIS package[7] is used to produce a minimized equivalent mapped for a technology (see [8] for more details). Combinational elements of circuits come from *blocks* without side effects, while sequential circuits are produced by retrieving the different states of variables writen by the *blocks*.

In the case of the MAPP, logic synthesis can be called from the compiler to produce a hardware automaton. Although much work remains to be done to obtain mature tools, the loop of **sobelTransform** has been easily translated into an automaton producing the instructions for the MAPP. The principle is to allocate a variable that will retain the different values in the $0 \ldots 7$ interval while the MAPP code translator is called to produce an output.

```
step:= 0.
↑ [| y | y := step.
  step := step +1 \\ 8.
  (( MappStoreTranslator new )
    src: #ad→ y; dest: #register→y) code ]
```

More complex sequencing is feasible by construction of a hidden program counter or controller composition [9]. An interesting point is the efficient allocation of FPGA resources in a full program, and the study of system support for swapping or composition of controllers.

6.2 Abstract execution platform

To deal with synthesis for FPGAs, an *abstract reconfigurable platform* (ARP) is under design. The aim of this platform is to cover a significant part of the functionalities and tools of reconfigurable circuits, to allow portability of such behavioral programs. The ARP allows to define concrete reconfigurable architectures (CRA) from patterns of cells, routing facilities, specific circuits. It also allows to produce routers, editors, and various control operations for CRAs. ARP is a hierarchical object model embedding components with common interfaces. The set of components is extensible. The model topology and the components can be characterized using a complete set of tools.

In this stage of the work, there is a CRA implemented for the Xc6200 architecture. Available functionalities are editing, geometric operations (mirrors, cut, paste, replications), point to point routing, partial configuration and readback. Global routing is still under investigation with a basic pathfinder algorithm implemented in the ARP.

The basic idea behind the ARP design is to provide a complete framework for system synthesis and performance evaluation for families of applications and families of architectures. The ARP isolates the CRA from the high level synthesis tools, but due to the object environment it is very easy to obtain interactions between high-level descriptions and hardware. Figure 1 shows a view of the editor with a small controller for the MAPP. This controller can work on an Hot2 PCI board with input/output control from the Smalltalk-80 environment (VisualWorks on Linux).

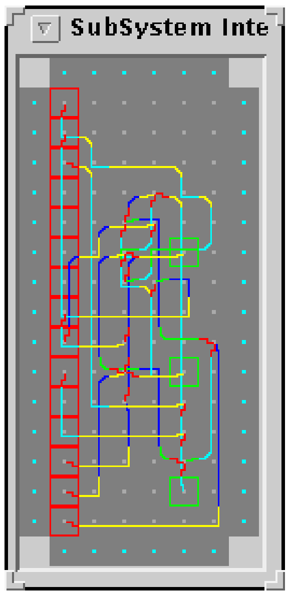

Figure 1. Xc6200 editor with a Mapp controller

6.3 Translation tools

This work is based on STEP technology citeSTEP. STEP recommands the use of the Express language for data and constraints modeling. A method and an environment has been developped to allow formal description of translators and semi-automatic production of associated tools (see [10]). Although this approach has only been applied to practical problems in software engineering, it is expected that it will assist the production of various data interfaces for inter-operability and further, program transformations for various targets. The new version of this environment is designed in Smalltalk-80 and will easily mix with the system synthesis tools.

6.4 Conclusion

Two smart sensor platforms have been addressed from small variations of the same environment. One platform is a remote intelligent camera interacting with a network, having a microcontroller executing the application software. The synthesis workbench and an equivalent of the commercial software support for the MAPP has been developped in 4 men-month. This includes code generation, basic image filtering, parallel arithmetic, transport to and from the camera, application level interactions with the camera. Porting to the second platform where an interpreter remains in charge of control was achieved in less than three weeks. The MAPP code generator was done in three days.

A programmer in charge of basic development has a clear model of the sensor that is directly usable in a powerful object software environment. Both presented platforms have been extensively tested by unexperienced programmers, that have been able to develop complete applications.

To complete the development environment, two main goals remain. First, efficient instruction scheduling to exploit low level task overlapping can be implemented using the C/MAPP program tree. This is critical for performance since computation and I/O can take place at the same time while photodiodes are being charged.

Second, additional hardware generation is also important since the control processor spends useless time looping to wait some conditions in the sensor. Producing such control inside an FPGA is also very attractive to increase performance. The same FPGA programming facilities can be exploited to implement filters or other regular transformations at the shift register output.

Further development should also focus on the relationship between the system components: synchronized transfers, buffered communication, asynchronous communications and other hardware details should be manageable by the compilers. The general context is the efficient automation of system synthesis, using a reconfigurable FPGA platform with compatible models and tools. Ongoing efforts are addressing hardware synthesis from Smalltalk behavioral specifications.

Different strength of types are required as an input for this kind of compilers: efficient logic synthesis requires more information on functional unit inputs than coding for microprocessors. Applying type constraints to Smalltalk methods is somewhat like a partial evaluation where it becomes possible to simplify the program expression in relation to low level synthesis rules. Progress on these points imply new mechanisms to manage types at the language level.

References

[1] IVP, *MAPP2200 PC System User Manual*. Linkoping, Sweden: IVP, Integrated Vision Products AB, 1994.

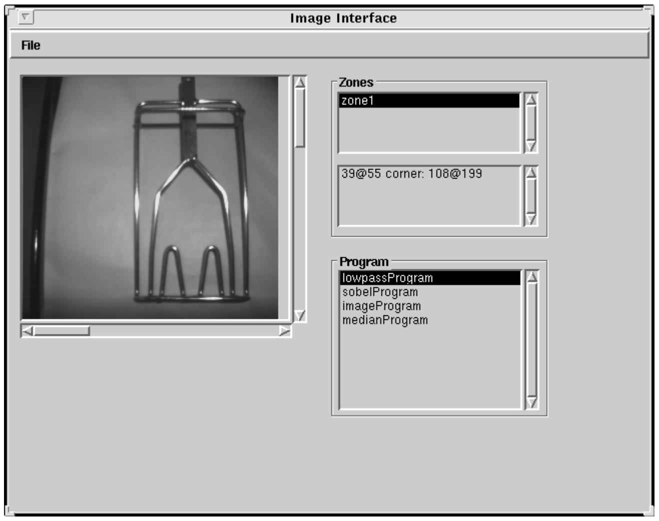

Figure 2. The remote MAPP system showing a captured image

[2] J.-E. Eklund, C. Svensson, and A. Astrom, “VLSI implementation of a focal plane image processor - a realization of the near-sensor image processing concept,” *IEEE Transactions on VLSI Systems*, vol. 4, pp. 322–335, September 1996.

[3] M. B. Ballard and A. Wirfs-Brock, “QuickTalk: a Smalltalk-80 dialect for defining primitive methods,” *OOPSLA’86 proc.*, vol. 21, Nov 1986.

[4] T. D., “Ubiquitous applications: embedded systems to mainframe,” *Communications of the ACM*, vol. 38, Oct. 1995.

[5] I. D., K. T., M. J., W. S., and K. A., “Back to the future. The story of SQUEAK, a practical Smalltalk written in itself,” in *OOPSLA’97, ACM.*, 1997. (see also http://squeak.cs.uiuc.edu/).

[6] J. Siegel, *CORBA, Fundamentals and Programming*. Wiley and sons, 1996.

[7] E. Sentovich and al., “SIS: a System for Sequential Circuit Synthesis,” Tech. Rep. UCB/ERL M92/41, U. C., Berkeley , May 1992.

[8] J.-L. Llopis and B. Pottier, “Revisiting Smalltalk-80 blocks, a logic generator for FPGAs,” in *FCCM’96*, IEEE, 1996.

[9] D. Micheli, *High level synthesis of digital circuits*. Nov. 1992.

[10] A. Plantec and V. Ribaud, “The STEP Standard as an Approach for Design and Prototyping,” in *The 9th Int. Workshop on Rapid System Prototyping, RSP’98*, IEEE, 1998.