Revisiting Smalltalk-80 blocks:
A logic generator for FPGAs

Bernard Pottier
Université de Bretagne Occidentale,
6 avenue Le Gorgeu, Brest, France

José-Luis Llopis
Universidad Jaume I,
Penyeta Roja, Castellón, Spain

Abstract

A Smalltalk-80 block is an encapsulation of code the evaluation of which is delayed, either for casual or repetitive execution (message value), or for process creation (message fork). Execution is driven following the object oriented paradigm with late binding of messages to actual functions.

A logic generator is described, which is based on the compilation of blocks to logic specifications. The translation process applies the block to a collection of objects representing a definition set. Resulting and original object collections are then translated into binary logic depending on observed classes. Block reference to remote variables allows sequential circuit implementations.

The logic generator operates in an interactive environment supporting BLIF and XNF logic representations. Logic optimization and partitioning is achieved using the SIS package. The testbed is the ArMen parallel computer which includes an FPGA ring.

1 Introduction

Several years of work on an FPGA based parallel computer have led to the decision to design and build a new environment in an object-oriented framework. ArMen software have been based on Unix tools and custom developments on FPGAs. This is the case for a cellular automata generator, a support for the systolic language ReLaCS[1] or an implantation of the Unity formalism[2].

These tools have achieved their respective goals. However, a global overview of ArMen project reveals that it lacks support to mix and reuse individual components. The situation is similar to that of an operating system having efficient specific tools (awk, sort, lex...), but which fails to compose their actions and data. Other problems arise when dealing with ports of development tools to other FPGA platforms, and setting-up of irregular applications such as input/output controllers.

To solve these issues, it has been decided to use object-oriented technology to make multi-tools open development easier. The project first stage was the definition and development of a lower level layer including modelization of computation based on Smalltalk-80 blocks, modelization of logic synthesis and data structures based on SIS tools[3], and finally technology support for FPGAs.

Blocks provide support for high level specification and execution. Their most common use is the control structure in the Smalltalk-80 language. They can also be composed to form operators or computing networks. Blocks are executable objects, and so they are likely to be moved to hardware. Although this is not fully developed, they can also maintain their logic and technology formats to allow future pre-placement and floor-planning.

This paper focuses on logic generation using Smalltalk-80 blocks. This is based on a programming method that separates the code (a block) and the data (the definition set of the block). As a result the polymorphism characteristics of object oriented programming remain usable in the code specification as well as in the data flow. Our approach is behavioral, closer to CSP like systems[4], rather than structural as used by other authors[5].

The paper provides a short introduction to the Smalltalk-80 language[6], with a special emphasis on different aspects of the block construction. The following section describes compilation technique from block sources to logic models and circuits. Two cases are detailed. One concerns blocks without side effects and the other blocks working on so-called remote variables. The last section explains how elementary circuits can be bound together to form operators or networks.
2 Smalltalk-80 blocks

2.1 Language presentation

Smalltalk-80 has been developed at Xerox PARC as a pioneer language and environment for interactive programming. The language has promoted the concept of objects reacting to messages using interface protocols called methods. Objects are grouped into classes embedding data and protocol definitions. Classes are themselves organized into a hierarchy on the top of which is the Object class. They provide an inheritance mechanism for protocols and data. The behavior of objects that are instances of the lowest classes may be very complex due to the accumulation of properties in the inheritance tree.

A key feature of the language is dynamic binding of messages to protocols. This mechanism enables to use the same message selector for various classes, even if these classes are in an inheritance relation. A related property is the possibility to easily reuse existing code as the system is extended.

The following tree which shows the hierarchy of numeric classes stands as an example. The root has methods shared by all the objects in the system. Some classes have explicit instance variables (Fraction), others are implicitly mapped to computer resources (SmallInteger, Double). Finally, abstract classes are roots of a subtree (Number). They are used to define algorithms for concrete classes.

Object ()
  Magnitude ()
    ArithmeticValue ()
      Number ()
        FixedPoint ('numerator' 'denominator' 'scale')
        Fraction ('numerator' 'denominator')
      Integer ()
        LargeInteger ()
          LargeNegativeInteger ()
          LargePositiveInteger ()
        SmallInteger ()
      LimitedPrecisionReal ()
      Double ()
      Float ()

The basic execution paradigm consists in sending a message to a receiver object. Objects are recorded with a marker that allows to retrieve its class when necessary. Let us consider the execution of the sin function in various situations.

1) \( x := 3.2 \) sin.
2) \( x := 3 \) sin.
3) \( x := (\text{Fraction numerator: 32 denominator: 10}) \) sin.

In the first case the receiver is a floating point constant member of the class Float. Float has a method for sin and is thus able to directly execute this message.

In the second case, the receiver is a SmallInteger that cannot understand sin. The Smalltalk-80 interpreter will search classes upward in the inheritance tree, to look for the expected method. These inspections stop at Number where a sin method exists. This method converts the receiver into an instance of Float and sends the message sin to it.

In the third case the receiver is an object dynamically created from the class Fraction. The previous example will be followed to find a method for sin.

As a consequence of dynamic binding, it is possible to design algorithms that work for any instances of Number. Furthermore, it is possible to extend the system capabilities by writing new subclasses of Number that will reuse these existing algorithms. Switching from a data class to another is as easy as shown above for the sin messages. This is due to the applicative style of programming that clearly separates data and algorithms.

Before giving more information about the language control mechanisms it can be pointed out that Smalltalk-80 has managed to take into account software complexity and functional style programming. Smalltalk-80 available products are very powerful platforms for application prototyping with main commercial products targeted to database and windowing systems interfaces. Development tools available in sources with the system include various kinds of browsers, inspectors, symbolic debugger, compiler and decompiler.

Smalltalk-80 is basically interpreted, but methods can be translated into host binaries to speed-up execution. Garbage-collectors take care of unreferenced objects that are known to appear at a terrific rate.

2.2 Blocks

2.2.1 Blocks as program structures

The language control structures conform to the message paradigm. This means that method descriptions are sequences of expedient of messages to objects. Control structures make use of blocks, that are instances of the BlockClosure class, encapsulating a frag-
ment of code from their *home* method. Blocks and methods have similar forms. A block may have a list of parameters (before the first vertical bar), a list of local variables (between two vertical bars) and a succession of evaluable expressions, all enclosed in brackets.

```plaintext
[ :parameter |
  temp := 2*parameter.
  ...
]
```

A block represents a delayed evaluation. Its sequential execution can be enforced as often as necessary by sending the message *value*. Basic example is the alternative test, where an object from a subclass of *Boolean* receives the message *ifTrue: ifFalse:*. Boolean true reacts by sending value to the first block, whereas false evaluates the second block. Boolean are themselves produced by reactions to messages, such as `<` in the following example:

```plaintext
counter < 7
  ifTrue: [ counter := counter + 1 ]
  ifFalse: [ counter := 0 ].
```

Looping control is achieved by blocks when they receive the messages *whileTrue: or whileFalse:*. The `shiftRight:` method from class *BooleanArray* quickly demonstrates array indexing (`at:`, `at: put:`), iteration (minimum to: maximum do: aBlock), block arguments ([ :i | array at: i put: i]), as well as looping on a control block (*whileTrue:) and the use of local variables ([ src dst ]).

```plaintext
shiftRight: count
  [ src dst ]
  count >= self size
  ifTrue: ['whole array is shifted out'
    1 to: self size do: [:i | self at: i put: false]]
  ifFalse:
    [ dst := self size.
      src := dst - count.
      src > 1
        whileTrue:
          [ dst put: (self at: src).
            src := src - 1.
            dst := dst - 1].
      1 to: dst - 1 do: [:i | self at: i put: false]]
```

### 2.2.2 Blocks as objects

Blocks are full status objects. They are usually passed as arguments during method calls in order to parametrize algorithms. They can appear as instance variables in some objects. A well known example is the *SortedCollection* class which uses its instance variable *sortBlock* as a sort key.

It is also possible to return a block as the result of a message evaluation. This is shown in the method below:

```plaintext
counterPlusOne
  [ counter |
    counter := 0.
    [ counter := counter + 1 \ 8]
    ( counter + 1 ) modulo 8
  ]
```

Then, each call to the method `counterPlusOne` returns a block that can be used anywhere to get successive values of a small counter:

```plaintext
+ [ plusOne aBuffer |
    plusOne := self counterPlusOne.
    aBuffer := Array new: 8.
    [ true ] whileTrue:
      [ aBuffer at: plusOne value + 1 put: self anything ].
```

The `plusOne` block is like a procedure context having its own local variables and arguments. The previous example shows that this block maintains a reference to the counter variable, located inside the *home context* where the block has been created. Home context variables referenced by blocks are called *remote variables*. They provide a remanent memory store for client blocks. Successive calls such as `plusOne` value are achieved in a new copy of the original block. However, all these evaluations will share the counter variable.

It can be noticed that different calls to `counterPlusOne` generate different sets of remote variables. This is of interest for the modelization of parallelism involving arrays of similar computations.

Figure 1 represents a block evaluation that makes use of a remote variable.

**Smalltalk-80** blocks also knows how to create processes in answer to the fork message. Scheduling and synchronizing processes is described in the class *ProcessorScheduler*. Synchronization mechanisms are provided by the instances of classes such as *Delay*, or
Semaphore. In the following example, a process is created and scheduled by sending fork to a block. The process repetitively creates a Delay, and blocks itself as it sends a wait message to this delay.

```
[10 timesRepeat: [ | delay |
    delay := Delay forMilliseconds: 500.
    delay wait.
    something doSomeTask. ] ] fork
```

To summarize, **blocks** are the objects used to control sequential as well as concurrent execution of code. They encapsulate source code with possible references to local variables from their home method context.

3 Moving blocks to FPGAs

In the logic generator, blocks are used to describe the lowest level of computation. This means that they need to move from their status of computation context to a status of hardware function. In this new position, the activation messages will consist in writing arguments to an interface and reading back results.

In foreign environments blocks operate as combinational or sequential data flow circuits depending on the way they are programmed.

This section describes a logic generator based on SMALLTALK-80 blocks behavioral analysis. Given a block and an input set of objects, the generator first builds a similar collection of output objects. Then, it searches for adequate binary representations and generates a logic model for the block. This method does not require any change in the original SMALLTALK-80 block source code.

To support circuit generation, an object environment supporting BLIF\(^1\) formats as well as technology support has been developed.

3.1 Combinational logic

The sequential evaluation of a block can be seen as the binding of an algorithm with objects passed as arguments. As it is not feasible to execute dynamic binding on specific circuits, arguments need to be restricted to a finite collection of objects. Such collections can be instances of `Interval`, initialized instances of `Collection` subclasses, `String`, `Text`, `Boolean` arrays..., or application specific classes. Combinational synthesis method is performed in 7 steps:

1. Blocks and argument collections are compiled out of textual specifications. Then, two collections of structured objects are produced, for the definition set and the result set.

2. these collections are analyzed and translated into boolean textual representations.

3. a SIS process receives this text and achieves an external minimization as well as a mapping on technology resources (Xilinx CLB logic gates, for example).

4. this mapping is parsed back to build a structured internal modelization using BLIF compatible classes.

5. optional registers are generated, and signals are renamed.

6. the BLIF boolean network is internally translated into an XNF\(^2\) object modelization.

7. input-output application signals are bound to actual chip resources and a final XNF file is generated for the chip.

3.1.1 Look-up table production

Following example **Bl1** shows a block that operates on two arguments \(i\) and \(j\). \(j\) is intended to be an index in a circular array of arbitrary numeric constants. \(i\) is an operand to be multiplied with one of the constants indexed by \(j\).

```
[ i j | "Bl1 example"
```

\(^1\)Berkeley Logic Interchange Format
\(^2\)Xilinx Netlist Format
constantArray nextJ x 1
constantArray := #: (2 3.2 4.3 5.4),
nextJ := j := constantArray size
  ifTrue: [1]
  ifFalse: [j + 1].
x := (i * (constantArray at: j)) asInteger.
Array with: x with: nextJ with: i \ 2 = 1

A possible set of objects on which the block can be applied is defined by two instances of Interval:

iSet := Interval from: 15 to: 23.
jSet := Interval from: 1 to: 4.

The whole definition set for the block is the product of iSet by jSet. Applying the block to each couple of this set creates the collection of result objects. Figure 2 shows a view of such a table for the previous block BL1. Its result is an array of three fields from classes Integer, Integer and Boolean.

![Look-Up Table](image1)

Figure 2: Arrayed result from combinational block of example BL1

External variables must not be referenced inside the block, since the results will depend on unknown values. The side effect question is treated in a further section.

### 3.1.2 Translation to booleans

The second step is the translation of the high level look-up table into an equivalent table of binary strings. It has been chosen to implement a separate hierarchy of specialized classes under a Binary/Translator root class. These classes are responsible for computing an effective range for binary representation of operands, and then to translate these objects into bit strings. Starting from a table similar to that of figure 2, the generator sweeps the left and right columns of the table calling a binary translator to compute the range.

In a second pass, each object is effectively translated into a bit string matching its computed column width. The object structures are flattened during the translation. They are pushed into the input stream of a SIS process as a PLA format.

### 3.1.3 Logic minimization and mapping

The external SIS process receives the previous data and a command script. The logic specification is minimized. Then it is mapped into small tables that will fit into XILINX CLBs. The new logic specification is passed back to the generator that parses it to produce a structured BLif object (SIS also produces information to merge CLBs).

The resulting BLIF model is traversed to rename signals in conformance with the original block arguments (see figure 3). Latches are optionally added on inputs and/or network outputs. Figure 3 shows a visual interface for a logic gate from example BL1, section 3.1.1. Browsers based on tables and graphs allow an interactive navigation inside the logic network.

![BLIF Format](image2)

Figure 3: View of the list of BLIF gates generated from example BL1 and a zoom on one of them.

The following step of synthesis consists in the transformation from BLIF to XNF models. Classes for XNF modelization allows to translate BLif representation into a technological format. This is a simple translation from BLIF logic gates into XNF symbols. A target computer dependent class is in charge
of binding the application to an actual interface module and chip resources. User interfaces operate on the top of this class. The final XNF file is produced by an external process which is transparently driven from the SMALLTALK-80 environment.

![Diagram of logic generation](image)

Figure 4: Logic generation diagram. Blocks are executable in 3 forms which are (1) sequential code, (2) lookup dictionary, (3) logic network

Last steps leading to a block execution in an FPGA are currently not supported by the generator. It remains necessary to place and route the circuit using XILINX specific tools. Then, very few interventions are required to boot the target system, to write some C code to feed the FPGA, and finally to observe the circuit behavior.

### 3.2 Sequential logic

It has been stated that references to external variables forbid the generation of an exhaustive precomputed table. This is the case especially if remote variables are referenced and modified inside a block. Sequential logic synthesis method is proposed through a practical discussion.

#### 3.2.1 Look-up table generation

The following method is a simplified version of the *home* for a block to be translated. It has a variable index which is first initialized to zero. Each expedition of values to the block will increment index, or recycle its value to zero. The result is the multiplication of the x argument by an element of the constants table.

```
makeBlock "BL2 example"
[index xBlock | "index is a remote variable for xBlock"
index := 0. ... 
xSet := Interval from: 0 to: 63. ... 
xBlock :=
[x ]
| constants result |
constants := #(1 2 2.4 4.4 6.6 4.3 3.1 0.8 0.1).
result := (x * (constants at: index + 1)) asInteger.
index := index + 1.
index = 8 ifTrue: [index := 0].
result].
↑ xBlock
```

Applied in this case, the section 3.1 look-up algorithm would produce a collection of meaningless results because each call to the block will make the index to progress. However, this problem can be solved by:

1. saving the value of each remote variable before evaluating xBlock,
2. observing all the changes operated on the remote variables,
3. restoring the saved state after the return of the block.

To execute these operations, two additional blocks are produced into the *home method* which save and restore remote values.

As the variable index is known to influence the block behavior, it must be considered as an input for the block. Because its value can be modified by a call to the block, it is also eligible as an output. Figure 5 shows a view of a look-up table for xBlock, with from left to right: x variable, old index value, new index value and finally the block results.

After one pass over the argument collection, an observation of the resulting index values can reveal additional values of remote variables. This implies an extension of the table that will contain these new states in the input remote column. Through dynamic building of the set of values on which the index variable is defined, it becomes possible to enumerate the whole set of entries and results for the block. An obvious minimal requirement is that the block algorithm gives a finite set of values to its remote variables. Back to the example, the highest order bit from the index variable has been extracted from the BLIF model and is presented on figure 3.

A noticeable difference with combinational blocks is that the definition set for remote variables is not
specified by the programmer. These variables only require an initialization similar to the reset state of an automaton. Blocks with side effects behave like finite state machines with the remote variables representing an internal state. Classes and values associated to these states are retrieved from the previous execution results.

3.2.2 Logic generation

Logic generation is achieved in the same way as for combinational blocks. It implies inspection of the collections representing arguments, remotes, and block outputs.

RANGES and binary translators are computed for each collection, then the object table is transformed into a logic table passed to SIS. Minimization and mapping take place, and a BLIF model is built. At this stage registers are generated for the remote variables. These registers are inserted in a loop between the outputs and the inputs of the logic network. Figures 6 shows a view of the BLIF model resulting from the previous example. The block input range for argument x is 6 bits; the range for the remote variable index is 3 bits, and the block output takes 9 bits. On the right side there is a list of components with more than 400 logic gates, 6 latches that have been required on block input x, and 3 latches to register the index variable.

Figure 9 shows a view of the program browser that can be used to avoid textual programming of the generator.

3.2.3 Interactions with the logic generator: an example from regular expressions compilation

SMALLTALK-80 uses an incremental approach to software development. Applications are written in terms of classes that add new features to the system. Every class can potentially use every other class in the system. In particular, the logic generator co-exists with the rest of the system and can therefore take advantage of it.

An example is the automatic generation of SMALLTALK-80 blocks from a grammar description. The objective is to generate automata behaving as scanners and parsers on an input data flow.

Grammar specifications are expressed using T-gen[7], a general-purpose object oriented tool which lives in the SMALLTALK-80 programming environment. T-gen provides almost the same features as the known UNIX tools LEX and YACC. It produces classes which can be used to scan and parse data executing methods on the fly.

Logic generation for such task follows the same steps as before, with remote variables holding internal states. At the moment, automata have been generated directly from a regular expression describing an expected token.

The scanner input is defined as a regular expression, e.g.:

\[ cg(g[tc]+gg)*tt(gc)+ \]

The block simply call the scanner to manage its internal state. The example above produces a BLIF file having 27 CLBs and 4 internal registers. Logic
4 Binding blocks

Blocks arguments, results and remote variables can be bound to physical FPGA or other block ports. The example below is provided to illustrate nesting of a block into an FPGA. The context is the ArMen distributed memory parallel computer. In this machine the node system bus are connected to an FPGA. These circuits are connected together into a ring. Figure 7 shows the interconnection system of an FPGA.

4.1 Linking a block into a system

The block searches a string pattern in its input stream. It normally copies the input char to its output. However, on two occasions, copying is replaced by the output of the string 'Hi!'. The first occurrence concerns completion of a string match in the input and the second one, a special condition signaled by a neighbor. Following is the code for remote initializations, argument set specification and block definition.

"Remote initializations" "Bl3 example"
state := 0, found := false, ptr := 0.
"Argument specifications"
argumentArray := OrderedCollection new: 2.
argumentArray add:
(Array withAll: 'allons a messine pecher la sardine').
argumentArray add: (BooleanArray new: 1).
"Beginning of the block"
[aChar otherNode |
values |
values := 'amen' asArray."To be found"
result := 'Hi!' asArray."To be printed"
(otherNode at: 1) "found variable, elsewhere"
ifTrue:
[found ifFalse: [ptr := 1].
found := true].
found
ifTrue: "output result using ptr"
[[ char |
char := result at: ptr. ptr := ptr + 1.
ptr > result size
ifTrue: [found:=false. ptr:=1. state:=0].
char]
ifFalse:"try to find the pattern values"
[state = values size
ifTrue:
[state:=0. found:=true. ptr:=1]

ifFalse: [aChar = (values at: state + 1)
ifTrue: [state := state + 1]
ifFalse: [state:=0. found:=false]].
aChar]]

Block argument aChar is bound to the processor interface (see figure 7). Block argument otherNode is bound to the FPGA left port. Block output is bound to the FPGA internal long lines that transfer back the values to the processor. Remote variables are also brought to the processor port, except for the found boolean that also goes to the FPGA right port. Block inputs are registered and clock comes from the processor transaction.

Passing the remote variable found to the right FPGA results in a wave of Hi! propagating over the machine nodes as soon as the searched string is detected somewhere.

Figure 7: Binding a block (Bl3) in ArMen FPGA ring.

This block produces a circuit with 9 input bits, 8 output bits and 6 internal registers which represents a 1700 lines logic table of 15 inputs and 14 outputs. Together with the char output, the internal state vector can be fetched out of the FPGA during processor accesses. SIS produces a description for one hundred logic gates within a 5 minutes delay (SUN SS5, 85Mhz). APR delay is 1 hour with a success rate of 50%. The full design linked with processor interfaces represents half of a 3090 FPGA.

4.2 Composing blocks together

Larger circuits can be described by composing blocks together in a hierarchical way. Several classes 5Automatic Place and Route
are proposed for block networking, which currently provide:
— block aggregation into cells,
— two dimensional arbitrary grid definition,
— two dimensional cell network creation,
— network connectivity specification and link creation.
— simulation and BLIF generation.

Simulation is supported by views that allows interactive session on the network. A BLIF model can be built out of preliminary minimized PEs blocks. A simple example is the structural description of an adder using full adder cells with a block for the carry, and an other block for the sum (figure 8).

Figure 8: A network of two adder cells

5 Conclusion

First, we would like to stress that the capacity of the generator is the capacity of underlying tools consisting of a logic synthesizer and, mainly, place and route programs. This works describes a logic generation method from blocks of a high level system. It must not be confused with a general purpose hardware language or compiler.

Several significant configurations have been brought out of SMALLTALK-80 blocks to an execution on a target computer. This shows that blocks can be migrated from their sequential form to look-up tables, then to an FPGA implementation. The second level also allows execution, due to an immediate conversion in Dictionary instances. Therefore, there is an interesting way to compare different execution technique and performances for a given problem: microprocessor, memory, specific circuit. A related point of interest is the application of Amdahl law in the case of this work. It is noticeable that the operations and the algorithms that are used in the block code can be very loosely related with the circuit complexity and performance. The most significant characteristics are the dimension of encoded data in the input and the output of the block. To make the most of such FPGAs, the challenge is to have the blocks work in parallel and harmony with an host processor, compose blocks together, preferably from SMALLTALK-80 native constructions, and finally pipeline the logic networks.

From the programming point of view, it has been shown that standard SMALLTALK-80 code allows to generate hardware functions. This means that algorithms can be written and tested in-situ for given object classes, and then, they can evolve easily to other classes. Internal states of sequential circuits can be made observable by bringing them to a processor interface. Such an approach could also help to implement high performance circuits by working at the data level rather than at the programming level (no bitst). Specific numeric classes can be programmed with variable or tunable precisions to support this approach.

The characterization of result sets provides a way to compose block together, and it remains possible, with some care, to pass different kind of values on the same data path.

Finally, it will soon be possible and easy, to program, reconfigure and to use FPGAs from a workstation SMALLTALK-80 environment.

6 Acknowledgements

Thanks are due to french "Programme de recherche coordonné en Architectures Nouvelles de Machine" (PRC/GRD ANM), as well as the spanish research foundation "Fundació Caixa Castellò".

References

Figure 9: Block programming browser (Example B13). The left part is dedicated to arguments or remotes specification or initialization. At the bottom of the view, there is the type inferred for results. The right top pane is for block programming. The right bottom pane shows a method which is produced by the interface.