A Data Oriented Approach to the Design of Reconfigurable Stream Decoders

Giovanni Agosta, Francesco Bruschi, Marco Santambrogio, Donatella Sciuto
Politecnico di Milano, Italy

ABSTRACT
The implementation technologies for electronic devices are experiencing a shift towards the use of reconfigurable devices. These devices are widely appreciated, especially in the design of embedded devices, since they allow to significantly lower the non-recurrent engineering costs associated with the hardware implementation of functionalities. In addition to these advantages, reconfigurable devices offer the possibility of implementing extremely flexible systems that can adapt or augment their hardware resources at run-time. To enable the exploitation of this important feature a key point is to make the designers able to model and validate reconfigurable systems, and to assist them in the evaluation of different target reconfigurable architectures. In this paper we address the problems of modeling reconfigurability for streaming data decoding systems at a high level of abstraction, and of the subsequent synthesis on a reconfigurable device. In particular, we define an abstraction of a generic reconfigurable architecture based on elementary modular cells (Reconfigurable Architecture Description Layer, or RADL). To model and validate reconfigurable behaviors we defined a formalism that is based on the object-oriented features of Java and on its possibility to vary the class pool at run-time. We then formally address the problem of synthesizing such an high-level model upon the reconfigurable architecture abstraction defined. In order to explore the effectiveness of the model, we implemented a single RADL cell prototype and evaluated area cost and reconfiguration times.

1. INTRODUCTION
Reconfigurable components such as FPGA are constantly gaining diffusion for the implementation of hardware functionalities in many embedded systems. These technologies are usually adopted for their flexibility and ability to reduce non recurrent engineering costs. One further advantage, at the moment only potential for the most part, is that reconfigurable components could allow the implementation of systems able to change their computational resources and circuitual behavior at run time. Among the main problems in exploiting such a possibility there is the lack of a formalism that lets the designer model the reconfigurable behavior in a high-level system representation; such a formalism would provide the designer with the possibility of modeling interesting dynamic behaviors in the early phases of the project. The executable models of the system thus obtained would allow the desired behavior to be simulated, enabling an early validation of the reconfigurable functionalities.

This work aims at investigating the potentialities of some object oriented languages features in the development of digital systems with dynamically reconfigurable components, with particular attention to the systems class of the stream decoders, that are of crucial importance in many multimedia applications (with the term stream decoder we mean a system that take as input a sequence of frames of different nature that must be decoded according to the type of the information they contain, and produces a set of decoded streams as output).

The interest in the object orientation is motivated by the growing attention that this paradigm is gaining also in the hardware/software design world, due to its ability to ease team design and to enable efficient and safe reuse practices. Some object oriented languages, such as Java, embed a native form of reconfigurability, that is the dynamic class loading. This mechanism allows an application to change the class pool it relies on at run-time in a safe way. This possibility can be exploited to give an application the peculiar flexibility of being able to change the data types it can deal with while the system is running. By exploiting this feature, we aim at showing how the behavior of interesting components of classical hardware and hardware/software stream decoding systems can be described in a data oriented fashion. In these descriptions, reconfigurability can be naturally introduced by the native class loading mechanisms of the language adopted for modeling. This reconfigurability implies the possibility to change, and in particular to widen, the set of frame types of the stream that has to be decoded.

As a second contribution, we propose a general model, based on elementary cells, that abstracts the main features of a reconfigurable differential data stream decoder. To obtain the DORM model we first define classes that represent the data packets forming the stream, representing all the information on the different packets as a class hierarchy, which, thanks to the dynamic loading, can be expanded at any time by simply adding new classes to the current pool.

As a second contribution, we propose a general model, based on elementary cells, that abstracts the main features of a reconfigurable hardware architecture. Such a model allows us to face the synthesis issue focusing on the peculiar features of the reconfigurability problem, avoiding concerns bound to the low level logic implementation. Moreover, the use of such layer makes the synthesis considerations independent of the individual reconfigurable architectures. We called this model Reconfigurable Architecture Description Layer (RADL).

After introducing the DORM and RADL models, we explore the problem of synthesizing a DORM design on a given RADL architecture, proposing an ad-hoc methodology based on a formalization of both levels of abstraction.

Moreover, we provide some insight on the feasibility of the implementation of a RADL reconfigurable cell on actual reconfig-
urable hardware, by means of a prototype implementation on a 
XC2VP7 Xilinx FPGA. Area and reconfiguration latency issues are 
analyzed by means of the prototype implemented.

2. PREVIOUS WORKS

2.1 High-level reconfigurability models

There are few previous attempts at modeling reconfigurable hard-
ware or hardware/software systems. Luk et al. use static networks 
to model several types of reconfigurable mechanisms [1]. How-
ever, their model is informal, and the executable model is provided 
by means of the Ruby language [2], a logic programming–style 
which is then compiled to an FPGA. This does not allow the use of 
programming language features familiar to most designers, such as 
object-oriented and imperative programming.

Another approach, proposed in [3], is to model reconfiguration 
as a specialization of a function for a given subset of its para-
eters. This approach, while straightforward in its adoption of a well-
known compilation technique, seems to work better to identify po-
tential reconfiguration points than to allow the designer to specify 
his own reconfigurable system. The work in [4] has the same goal–
automated identification of potential reconfiguration elements, but 
adopts a graph–based technique.

A work closer to the present one is [5], where a hardware de-
scription language based on Java, JHDL, is proposed. Its recon-
figurability mechanisms are based on a socket structure, and it op-
erates at a considerably lower level than our methodology. Other 
approaches to synthesize higher level models onto reconfigurable ar-
chitectures can be found in [6, 7].

2.2 Java for HW/SW system design

A number of attempts to use the Java programming language 
for the design and specification of embedded systems can be found 
over the past decade. At first, the main focus was on the adop-
tion of Java as a way to add constructs to represent concurrency 
to the object–oriented features common to other languages such as 
C++ [8] [9], while keeping the models executable. The main issue 
faced by these early attempts was the difficulty in statically analyz-
ing Java.

Later, other authors chose to retain different features of Java, like 
its relative simplicity, and create new specification languages based 
on those principles [10]. A crippling flaw of this approach is that 
is does not take advantage from the standardization of Java, thus 
increasing its application costs.

Another relevant approach is presented for different levels of de-
scription, including RTL, in [11] and [12]. The same authors even 
propose synthesis of Java–based specifications [13].

All of these works focus on object–orientation and thread execution 
as the most useful language features for hardware/software speci-
fication. On the other hand, we prove that dynamic class loading 
can be successfully exploited in the design.

2.3 Synthesis of object-oriented specifications

Examining the existing literature, we see that hardware synthe-
sis of object-oriented descriptions has been treated in [14], also for 
models that make use of polymorphism. Polymorphism is the basic 
mechanism that allows the pool of classes to dynamically vary dur-
ing the system working. In [14,15] a methodology for the synthesis 
of descriptions with polymorphic features, where all the data types 
are known at synthesis time, is presented. The reconfigurable cell 
basic pattern was already outlined, in a preliminary form, in [16], 
where the link with upper level models was not established yet.

3. DATA ORIENTED RECONFIGURABLE MODEL

In this section, we define the Data Oriented Reconfigurable Model 
(DORM) by means of a set of guidelines that a designer must fol-
low in order to obtain a stream decoder model that can successfully 
employ dynamic class loading as a tool for describing reconfigura-
tion. In order to obtain a DORM model of a system using Java, the 
following rules must be observed:

Data-oriented Stream Frames Modeling Data manipulation of stream 
frames functionalities must be embedded into the data defi-
nition itself by means of classes methods; this way, the opera-
tions to be performed on a frame are directly coded within 
the data class definition; this data oriented perspective gives 
the possibility of modifying the structure of the description, 
de–centralizing the data manipulation modeling from the sys-
tem to the data processed and elaborated;

Inheritance Different frame types and different operations per-
formed on them must be structured into a class hierarchy;
there must be a root class defining the most general frame 
type and set of operations to be performed; all the infor-
mation that has to be shared in the computation of different 
frame instances must be represented by means of static data 
members of the root class;

Polymorphism Different operations on the frames must be rep-
resented by redefinition of children classes; different opera-
tions are then described by means of different redefinitions 
of the virtual methods of the parent class;

When the frame to be decoded is described with the DORM 
paradigm, dynamic class loading naturally models the reconfig-
urable capabilities of the system, in two peculiar fashions:

Adding a class The pool of classes that represent the frames to be 
decoded can be enriched by new elements; this has the mean-
ing, in the domain of the application, to enable the decoder 
to handle a new type of data frame. In Java models execu-
tion, adding a class is accomplished by adding the new class 
description file in the class hierarchy.

Redefining a class An existing class could be updated in order to 
upgrade its functionalities; in Java, classes description files 
can be replaced with different versions;

These two reconfiguration methods allow to extensively change 
the system behavior; the reliability of the reconfiguration phase is guar-
anteed by the object–oriented interfaces mechanism implicit in the 
use of DORM.

4. DORM MODELING PARADIGM

As the most general modeling paradigm, consider a decoder that 
receives as input data frames, performs some kind of elaboration 
on each of them, and encodes the result back as a bit string. 
This general pattern captures many multimedia stream decoders; for in-
stance, in an mpeg stream, the tokens are the key frames and the 
differential frames, while the result is a bitmap encoded image. The 
system is modeled as a set of Java classes, according to the DORM 
style. The way the instances of these classes are connected is shown 
in figure 2. The type recognizer is a class that receives an encoded 
data packet and produces a type code and a bit string that represents 
encoded data. Thus, each different data token is associated with a 
class by means of the type code.
The elaboration to be performed on the different types of data tokens is encoded as a method of the class associated with the data type; on the other hand, the information that must be persistent between the different packet instances is represented as a data member of the root class, as shown in the class diagram of figure 1. When the executor receives the type code, it looks into the class pool for an item whose name matches the type code. After that, the right class is loaded, and an instance representing the data token is generated from the encoded data. Then, the decoding method is invoked on this newly created object, and the result is passed back to the output encoder. In this example, the result type is fixed, and the output produced is always a bitmap encoded image. In a more complex scenario, many output channels could simultaneously be fed with the result of the decoding of different frame types, such as, for instance, audio and video frames of an mpeg stream.

5. RECONFIGURABLE ARCHITECTURE DESCRIPTION LAYER

In order to consider the synthesis problems peculiar to the reconfigurability apart from the low level circuitual implementation issues, we define a description layer that highlights the specific features of a general reconfigurable hardware architecture. The general problem of the synthesis of a high level model down to a reconfigurable hardware implementation can be split into two decoupled phases:

1. the synthesis of the high level system description on the RADL representation;
2. the mapping of the RADL model on the specific target reconfigurable architecture;

The basic element of a RADL model is the reconfigurable cell. The basic structure of a reconfigurable cell is shown in figure 3. The interface of the reconfigurable cell requires a couple \((data_{in}, op_{id})\), and provides a \(data_{out}\), where \(data_{in}\) represents the encoding of the data to be elaborated, \(op_{id}\) is a code that represents the operation to be performed on it, and \(data_{out}\) is the encoding of the result produced. The cell also contains the following elements:

- an array of functional units; a functional unit is the basic computation element. It represents an operation that maps the domain of \(data_{in}\) on the codomain of \(data_{out}\). To each functional unit a certain operation is assigned at a given time point; the reconfigurability of the model lies in the possibility of changing this assignment over time;
- a type table; this element contains the assignment between the functional units and the operations;
- a reconfigurator; this element receives as input \(Type\), and checks whether the corresponding data token type is assigned to any of the functional units of the array; to do this, it exploits the information contained in the type table; if the type is unavailable, the reconfigurator searches for the required functionality into a configuration pool that is external to the cell, and modifies the assignment to the units in order to make the required operation available; the configuration pool represents the reservoir of all the possible functionalities available;
- a functional unit selector; this element receives as input \(data_{in}\) and \(Type\), and dispatches the data to the functional unit that implements the operation corresponding to \(Type\). To do this, it uses the type table information;
- a memory; this represents the state of the cell as a memory element that can be accessed by the functional unit both for reading and writing.

A RADL model is defined as a description that uses an arbitrary number of such cells to describe its reconfigurability component. The structure above depicted was chosen since it allows to focus on the reconfigurability properties of the system, representing it by means of the functional units reassignment, abstracting them from issues such as the logical synthesis of the operations.

In this way, the abstract problem of mapping an object oriented specification in which reconfigurability is expressed by means of polymorphism, such as DORM, onto a reconfigurable device, is faced in its generality, independently of any given specific architecture. Due to the level of generality adopted for this analysis, timing details are purposefully omitted. Nevertheless, this omission at this point of the study does not preclude the possibility of adding lower level information in the RADL model for finer grained analysis or performance evaluation.
6. ISSUES IN RECONFIGURABLE SYSTEMS SYNTHESIS

Let us now consider the problem of mapping a DORM model onto a RADL architecture. For the sake of clarity, we will consider a DORM model based on a single root class. Moreover we will assume that the root class contains only one public method called execute. These assumptions do not imply any conceptual restriction to the analysis. Let \( R \) be the model root class, defined as a tuple \( R = \langle \delta_0, \delta_o \rangle \), where \( \delta_0 \) is the data type of the static data shared by all instances of the derived classes, and \( \delta_o \) is the type of the output returned by method execute.

Let \( C \) be the set of classes that inherit from \( R \); then, \( \forall c \in C, c = \langle \delta, m > \) where \( \delta \) is the set of properties added to class \( c \) with respect to root class \( R \), and \( m \) is the execute method implemented within class \( c \). We represent the elements of class \( c \) as \( \delta_c \) and \( m_c \) for the sake of brevity.

The execute method \( m_c \) is a function defined as \( m_c : \delta_c \times \delta_o \to \delta_0 \times \delta_c \). This function receives as input the class data token and the shared static data, and its output domain is defined by the root class output data type. Method execution can also have side effects, in that it can modify the shared static data component \( \delta_0 \).

Let also \( \Delta = \{ \delta_c \mid c \in C \} \) be the set of all data token types and \( M = \{ m_c \mid c \in C \} \) the set of all methods. This model can represent a generic class hierarchy. To properly address dynamic class loading, however, \( C \) must be considered as function of time \( C(t) \). Therefore, \( \Delta \) and \( M \) are generalized as \( \Delta(t) \) and \( M(t) \).

Based on this representation, the translation of a dynamic class hierarchy into a RADL reconfigurable cell can be described formally. Let \( t_s : \delta_0 \to \text{bits}\text{sequence} \) be a transform that maps the domain of the data type \( \delta_0 \) to a bits\text{sequence}. The set \( T \) of all the transformations \( t_s \) is defined as \( T(t) = \{ t_s \mid \delta_0 \in \Delta(t) \} \).

The transformations \( t_s \) formalize the representation adopted to encode the data. An example of a transformation that implements \( T \) is the Java object serialization mechanism.

The representation, in the RADL architecture, of a complete DORM data packet must encode the information of the packet type. An encoding function that maps \( C \) to a set of bit sequences is then needed: \( \tau : C \to \text{bits}\text{sequence} \), where \( \tau \) is an isomorphism.

The RADL functionality pool \( P(t) \) can then be seen as the set of functionalities

\[
P = \{ f_c = f(t_0^{-1}, m_c) \mid c \in C(t) \}\]

where each functionality is obtained by combining the inverse transform of \( t_c \), with the execute method of class \( c \). \( t_0^{-1} \) represents the decoding function needed to rebuild data from their representation.

The actions needed to synthesize a DORM single class hierarchy onto a RADL architecture can then be summarized with the following steps:

1. Define the encoding transformations \( T(0) \);
2. Define the isomorphism \( \tau \);
3. Implement the functionalities \( P(0) \) onto the target architecture.

To add a new data packet type the appropriate \( t_{new} \) and \( f \) must be defined.

Figure 5 shows how the DORM to RADL translation can be formalized by means of a BNF grammar with semantic actions.

7. RADL CELL SYNTHESIS EXPERIMENTS

One of the main issues we need to address in order to prove the effectiveness of our modeling approach is the feasibility of the implementation of a RADL cell on an existing physical architecture. To address this concern we implemented and tested a prototype of a single cell reconfigurable system implementing the RADL model.

The target architecture chosen for the prototype implementation was a Xilinx FPGA XC2VP7. The features of this device are summarized in 1. The frame is the smallest reconfigurable unit of the
data_token_class :
    CLASS class_id ':' IMPLEMENTS data_token_interface '{
    dt_class_definition '}

$$ = new radl_token();
$$:token_type = hash($2);
$$:token_data = $7:datad;
global_behavior_list.append($2,$7:behavior)

};
dt_class_definition :
    dt_attribute ':' dt_class_definition
{$$ = $3:datad.append($1); }
    dt_method ':' dt_class_definition
{$$ = $3:behavior.append($1); }
    /* empty */
{$$ = new radl_data_info(); }

dt_attribute :
    type_id attribute_id
{$$ = new radl_data_field();
$$:type = $1;
$$:id = $2;
};
dt_method :
    type_id method_id '(' parameter_list ')' '{
    dt_method_implementation '
{$$ = new radl:behavior();
$$:RADL:implementation = java2RADL($1, $2, $4, $7);
};

Figure 5: DORM to RADL translation formalized by means of a BNF grammar with semantic actions

device: a single frame is 424 bytes (or 3,392 bits) long, and a total of 1320 frames is available.

| Table 1: Features of the Xilinx FPGA XC2VP7 |
|--------------------------|--------------------------|
| CLB                      | 4 Slices                 |
| Slice                    | 2 Logic Cells            |
| Logic Cell               | 1 4-input LUT + 1 FF + Carry Logic |
| Reconfiguration          | JTAG / Boundary Scan Programming Mode |
| Number of Frames         | 1,320                     |
| Frame length             | 3,392 bits               |
| Configuration bits       | 4,477,440                |
| Configuration bits + header | 4,485,472                |

In our prototype cell, provided with four reconfigurable functional units, the dynamic reconfiguration relies on an external reprogramming schema: the reconfigurator is implemented on an external processor that receives the information on the current packet type and, if reconfiguration is needed, reconfigures one of the functional units of the array.

The functionalities considered for this experiment range from very simple logic functions to slightly more complex arithmetic elaborations. These functionalities are not intended to be meaningful with respect to any particular decoding application; rather, their role is to verify the practicability of our reconfiguration pattern, and to obtain some initial information on the relationship between functionality complexity and reconfiguration time overhead.

Figure 6: Experimental data: reconfiguration times for the RADL implementation

The experimental data collected in Figure 6 show a linear dependence of the reconfiguration time on the bitstream functionality size. Even though the external reconfiguration times are in the order of magnitude of the seconds, ongoing experiments suggest that a completely internal implementation of the reconfigurator would allow reconfiguration times of microseconds.

Our experiments also show that a trade-off is to be considered between the size of the reconfiguration data and the overhead given by the reconfiguration block headers. For reconfigurations involving only a few frames, this overhead can be larger than the actual data, so each reconfiguration should involve at least ten frames, which is approximately the size of an adder or similar functionality.

8. CONCLUDING REMARKS

In this paper, we offered four distinct contributions:

1. We presented a model (DORM) for reconfigurability in stream decoding systems at a high level of abstraction, using one of the main, but more underrated in the field, features of Java for this purpose, dynamic class loading;

2. We proposed a general reconfigurable architecture description layer (RADL), useful for capturing the features of a reconfigurable system and isolating them from the implementation details;

3. Based on the formalization of a simplified DORM model, we analyzed the synthesis problem, highlighting its main issues.

4. We verified the feasibility of the implementation of a RADL cell on existing reconfigurable architectures by means of a working prototype.

The exploration possibilities offered by the DORM formalism could enable the design of reconfigurable systems in fields of application even different from that of multimedia stream decoders. Moreover, using a language that is widespread in the software world and that features object orientation allows DORM to be matched
with other approaches that model hardware/software systems with Java.

The work will continue in different directions:

- Application of the design methodology to other potentially interesting scenarios, with special regard to mobile code applications for embedded systems;
- Implementation of a tool for the mapping of the dynamic class loading upon reconfigurable devices;
- Synthesis of the components of a RADL reconfigurable cell on an FPGA platform with internal reconfiguration;
- Analysis of the application of other advanced programming language features, among which aspect oriented programming, to the refinement of the high level models we defined.

9. REFERENCES


