DISSERTATION

submitted

to the

Combined Faculty for the Natural Sciences and Mathematics

of the

Ruperto-Carola University of Heidelberg, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

Richard Donald Sylvère Leys
Born in: Croix, France

Raising the abstraction level of hardware software co-designs

Advisor: Prof. Dr. Ulrich Brüning

Date of Oral Examination: ……………………………………..

Abstract

As lithographic processes' size to manufacture transistors shrink, the number of available transistors on integrated circuits (IC) increases. Newly manufactured ICs require innovations to leverage improved performances or area occupation, and feature more and more components on the same chip, which work together and/or independently to provide an advanced set of functions.

The complexity of hardware design flows consequently increased: from circuit description to functional verification and in-system interface, every stage is now more and more driven by a cross-product function between a set of reusable functional units and constraints to target a specific technology (ASIC, FPGAs etc…) and configuration.

This diversity in the possible outputs for a set of components calls for the development of new methodologies to raise the abstraction level in the design flows. A better abstraction allows optimizing and automating more processes, from component specification to final implementation and interfacing.

New Abstraction levels have always emerged through industry standards like Verilog and VHDL for digital circuit description, SystemVerilog/UVM/e for functional verification, or by vendor specific toolchains. However, standards and software toolchains usually lack flexibility as they operate for a bounded range of functionalities.

This thesis presents some novel applications covering various stages of the design flow, ranging from digital design input (register file generator) and ASIC circuit implementation (Hierarchical Floorplaning), up to in-system IC integration (Part design language). They are backed by a generic software design methodology based on functional programming used to develop domain specific languages embedded in the TCL interpreter.

To complete the design flow path from circuit implementation to software integration, a hardware-software interfacing point linked with the Register File Generator design tool will be presented. It is based on a generic and innovative XML-Data binding technology which was developed during this work.

The iterative loop between application definition and flexible software components reuse presented along this work also provides a general guideline to develop future design flow components, and guarantee their integration in any target environment.

Zusammenfassung

Die menge von Transistoren, die auf einen Integriertes Schaltung (IS) zu Verfügung stehen steigt mit die Verkleinerung der lithographischem Prozesse. Neue erstellte IS benötigen Innovationen um Performanz und Fläche auszunutzen, und bitten immer mehr Komponente auf den gleichen Chip, die miteinander oder unabhängig von einander arbeiten müssen, um ein fortgeschrittenes Satz von Funktionalitäten anzubieten.

Als Konsequenz davon, steigt die Komplexität der Design Flows mit: von Schaltung Beschreibung bis Funktionale Verifikation und In-System Integration, jede Stufe ist immer mehr abhängig von ein Satz von wiederverwendbarer Funktionale Einheiten und so einfach wie möglich Einschränkungen, die zu einen bestimmte Technologie und Ausstattung gezielt sind.

Diese Vielfältigkeit der möglichen Ausgaben für einen gegebenen Satz von Komponente spricht für die Entwicklung von Methodik, die den Abstraktion Grad in Design Flows erhöht. Einen besseren solchen Abstraktion Grad erlaubt eine Optimierung und verbesserte Automatisierung der Design Prozesse, von Spezifikation bis Endgültigen Implementierung und Integrierung.

Neuen Abstraktion Grade stellen sich typischerweise heraus durch Industrie Standarte, wie Verilog oder VHDL für Digitale Schaltungen Beschreibung, SystemVerilog/UVM/e für Funktionale Verifikation, oder durch Vendor eigentümliche Lösungen. Standarte und Software greifen allerdings immer auf einen bestimmten Untersatz der Design Flows zu, und fehlen dadurch die benötigte Flexibilität um sich mit anderen Aktoren geschickt zu integrieren.

Diese Dissertation stellt eine ausgewählten Satz von neuen Anwendung vor, die verschiedenen Design Flow Stufen decken: Digital Schaltung Beschreibung (Register File Generator), ASIC Implementierung (Hierarchical Floorplaning) und PCB Design (Part Language). Diese werden von einen Generischen Software Programmierung Methodik unterstützt, die sich auf Funktional Programmierung Konzepte bezieht, um Domain Specific Languages in dem TCL Interpreter einzukapseln.

Um den Design Flow Pfad zu vervollständigen, einen Hardware-Software Schnittstelle, die sich mit dem Register File Generator integrieren lässt wird eingeführt. Sie greift auf einen generischen und bahnbrechende XML-Data Binding Technology, die währen diese Thesis entworfen wurde.

Die Entwicklung Schleife, zwischen Anwendung Spezifikation und anpassungsfähige Software Komponente Wiederverwendung hin und her, die durch dieser Arbeit vorgestellt wird, bittet sich als Leitfaden für zukünftige Entwicklungen von Design Flow Aktoren.

Table of Contents

1 Introduction

Following the famous Moore's law on the growth of transistor count in an integrated circuit, the complexity of hardware designs has grown over time, as more fields of application developed: microcontrollers, mobile processors, graphics processors, application specific co-processors etc…

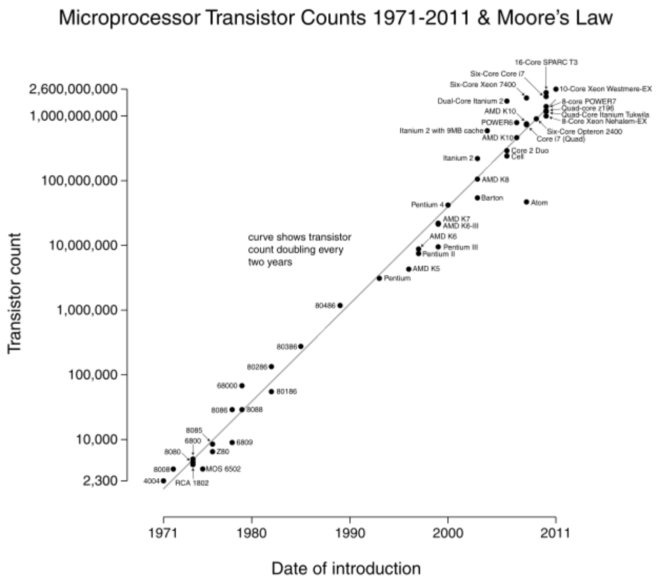


Figure 1-1 Moore's law applied to microprocessor transistors count [1] **(Logarithmic scales)**

Meanwhile, complexity in terms of number of transistors also means complexity in terms of number of features that are integrable in a system on a chip (SOC), and by extension, it impacts a project's design space in terms of:

- ✓ Architecture specification
- ✓ Integration of components
- ✓ Testing and verification
- ✓ Feasibility
- ✓ Human resources and time to market costs

This last criterion being quite prominent, especially in the context of a start-up, as business financing entities tend to look for a rapid high margin return on investment, underscoring the need to reach the break-even point [2] [3] as fast as possible.

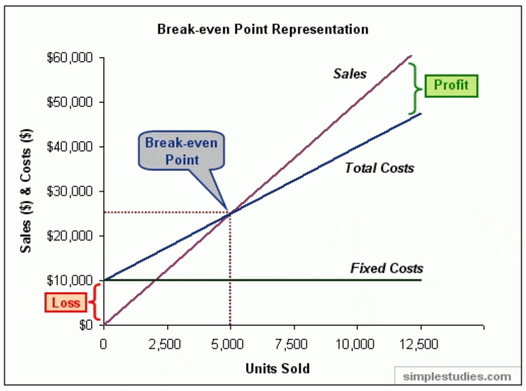


Figure 1-2 Break-even point representation [3]

This surge in complexity has forced research and industry over the past 25 to 30 years to develop new tools, languages and methodologies to help tackle issues at each step in the design flow, a few examples being:

- ✓ Digital Hardware design input
  - Hardware description language (HDL)
  - Finite state machine editors
  - Linting etc…
- ✓ Simulation and verification
  - Simple simulation
  - Advanced verification
- ✓ Technology mapping
  - Synthesis
  - Timing analysis
  - Signal integrity
  - Power analysis

On the other hand, with the help of these new design methodologies, sub-designs became more and more reusable, allowing the optimisation of the engineering costs by sharing them over multiple product lines, and/or buying some from third parties (they are then called IP Blocks), as illustrated in Figure 1-3.

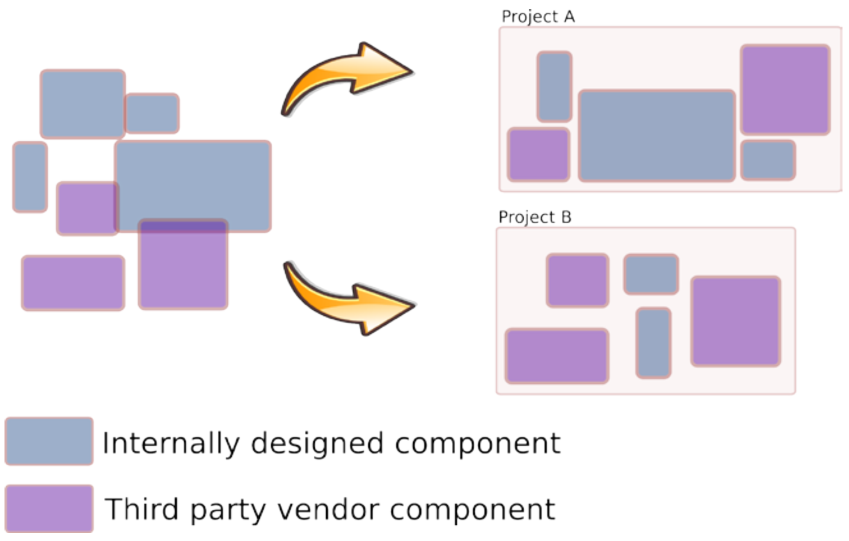


Figure 1-3 Component reusability in designs

Therefore, more than ever, engineering teams have to implement features that are likely to be integrated in different designs and mapped to various technologies, while keeping up with the constraints of all possible configurations.

In Figure 1-4, some of the main design flow steps to produce an integrated circuit are presented (each also embeds its own sub-flow). We can see that each stage is connected to its previous and next through an evaluation loop, building a transversal dependency across the processes. This leads the engineering teams to facing a great variety of software to handle each step from hardware design input to technology mapping, each of them with varying degrees of compatibility to its successor/predecessor.

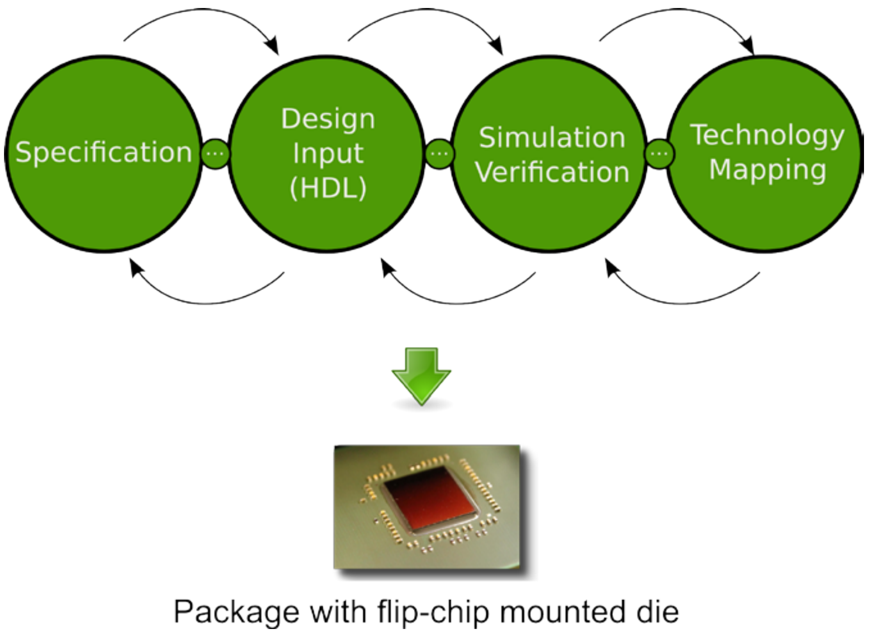


Figure 1-4 Design Flow example with major steps to produce an integrated circuit

Typically, designers try to leverage this issue by creating various sets of custom software (simple scripts for most), quite often written in a different language, or even develop some Domain Specific Languages (DSL), that only have a localised usage, and little interoperability.

Although modern software design tends to maximise function reusability, hardware designers tend to throw suspicion at software experts, who usually try to create a perfect tool that solves too many problems. Probably they are right to do so, as the typical cost and complexity of industrial tools does not give credit to the idea of constraining any design flow under a single solution, creating an illusion of freedom by enslavement.

1.1 Stakes

As seen so far, hardware designs are exposed to quite complex and fast evolving design flows, and have to meet multiple constraints while minimizing redesign needs, justifying the difficulty to offer efficient tools able to follow a project along all its implementation phases. But moreover, software concepts trying to offer “one tool to rule them all” are unrealistic, and contra-productive in regard to modern software designs. What we are looking for would be then to be able to:

- ✓ Bridge the gap between abstract design concepts (Top-Down view) and implementation (Bottom-Up construction)
- ✓ Easily specialise the design as the requirements are getting clearer
- ✓ Efficiently analyse the design at each flow step, to outline refinement iteration to be done based on specification changes, and specification feasibility issues.

We can try to sum up those issues in one question:

Can we marginalise the implementation of a specification, while consistently guaranteeing behaviour and feasibility?

In this thesis, we propose a set of open software design principles, inspired from functional programming paradigms, applied to hardware design flow challenges, in an attempt to raise the global design abstraction level, while not stealing control from the designer. Although focus will be given to the TCL programming language, which is present in most of the Electronic Design software in the industry, and the Scala programming language for hardware-software interfacing, the presented concepts are meant to be translatable to other technologies.

This work is structured around a presentation of the core concepts of functional programming and domain specific language development, which lead to defining a methodology for creation of Embedded Domain Specific Languages.

In a second time, some functional programming extensions to the TCL language are introduced to support Embedded Domain Specific Languages in TCL.

Finally a set of chosen applications covering digital hardware design input, physical floorplanning for integrated circuit, and high-level software interfacing are presented.

For readers not familiar with Functional Programming, it is advisable to focus on the presented applications, and come back to the lower level concepts iteratively.

1.2 Contributions

This work shows how learning from functional programming and domain specific language development allowed us to build very creative and elegant software solutions to create abstract programming interfaces inside the TCL interpreter.

The choice of the TCL dynamic language offered direct interoperability with existing industry software, showing how applications could be developed to nicely integrate inside existing design flows.

Beyond the applications, the proposed abstract methodologies for embedded domain specific language development can be used as basis to develop abstract programming interfaces, not only using the TCL language but in every possible context.

The architectures of the presented applications moreover prove that flexibility in software can be reached by reusing generic building blocks, and actually pushing the application-specific behaviour mostly to the binding logic layers.

2 Functional programing and domain specific languages

The guiding thread of this thesis is the development of software design paradigms that are flexible enough to be adapted to design flow specifications. In this perspective, the technological choices made when developing a software component have an impact on the flexibility degree that can be reached. Indeed, three main criterions will have a major impact:

1. Language features: Is the base programming language good enough to limit the human costs of development and maintenance?
2. Acceptance: In the case of programming interfaces, are the users going to be willing to use them, or be reluctant to learn new paradigms and syntaxes.
3. Integration: If multiple software pieces must work together, how well are they going to integrate with each other? Are we going to need extra data exchange formats and protocols to cover incompatibilities?

Starting from imperative programming knowledge (C/C++, Java etc…), we explored alternative ways to design programming interfaces that would closely match design issues encountered by hardware designers. Classical C/C++ or Java library development is a way to go, but they provide a low level programming view to solve a problem, and are not very well adapted to the flexibility required by top level views of designs. Moreover, non-software experts tend to be very reluctant to verbose languages and to the usage of standard software programming patterns.

Figure 2-1 shows the development path followed along this work. Through experimentations with Domain Specific Language (DSL) design, which have the chance to correctly answer issues 1 and 2, and learning about functional Programming, we developed a way to create programming interfaces called Embedded Domain Specific Language (EDSL), which address the three mentioned challenges. The main implementation focus will be set in chapter 3 on the TCL programming language, as it is the first-choice language for most concrete applications presented in chapter 4. The functional language Scala will be used as an existing technology to support and illustrate the design patterns ported into TCL.

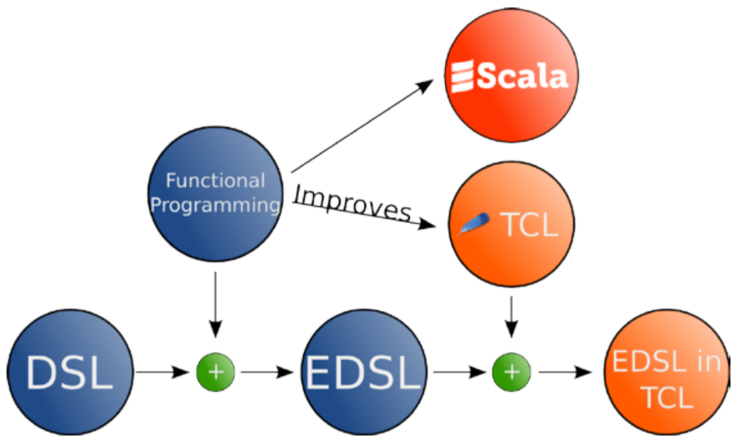


Figure 2-1 Functional Programming to EDSL development map

2.1 Imperative and Functional Programming styles

Historically, computers were designed to perform computations in an automated and faster way than humans could on their own. That is to say, basically solve mathematical problems. Programming languages emerged as a human-understandable way to describe some computations to be performed by computing units, just like scientists write down and solve equations. However, those computing units (commonly called processors) can't understand human languages, as they are only electric circuits which can process a simple instruction (in the form of binary digital signals [4]) and produce outputs for the next ones. This instruction format is called a machine code, and modern processors as well as the first one ever build still work by running such machine code.

An interesting analogy to this concept, presented in Figure 2-2, can be made with fairground or street mechanical organs. They use a "mechanical" representation of music notes, taking the form of a barrel or a punchcard music book, which triggers actuators producing the desired note. The first computers were also built the same way, at the time when computer programs would be translated to punchcards, then run by the machinery to produce computation results.

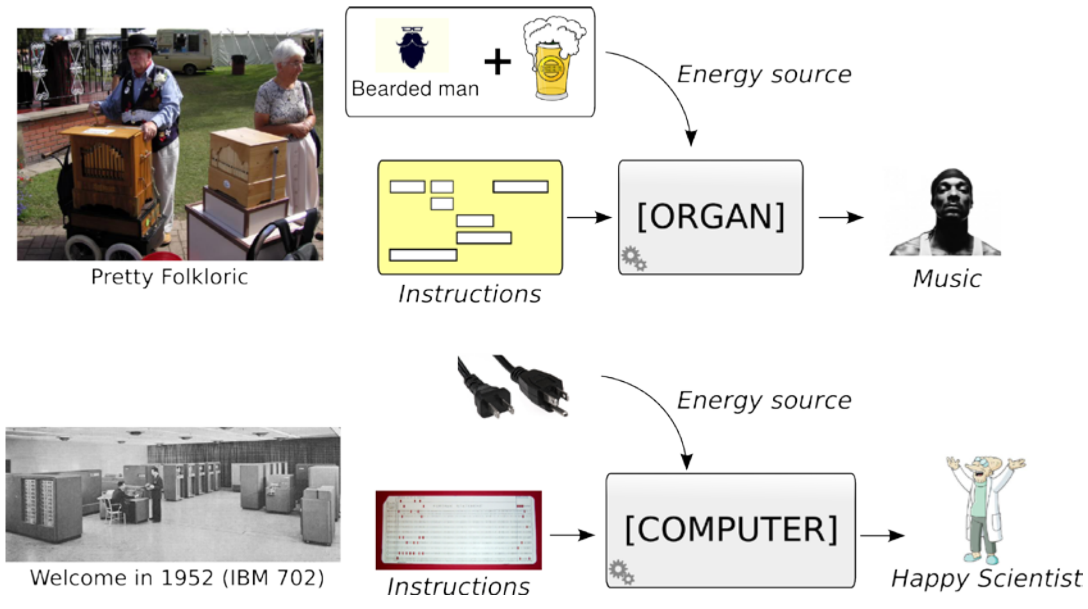


Figure 2-2 Computers and punchcards organs are not so different

Because the machines they are running on have special requirements related to their architecture, the instructions present at the machine code level are not tightly related to the initial problem description. Programming languages must therefore be compiled from a human understandable text representation to an executable machine code format, as presented in Figure 2-3.

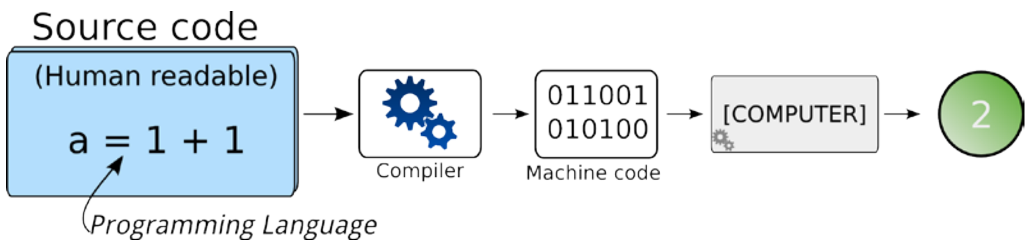


Figure 2-3 Simplified compile-execute flow for a program computing "1+1"

This is where the story of computer programming languages begins. The goal of a good programming language is to allow the user to clearly and efficiently write a computer program, while compiling to the most efficient possible instruction set for the target machine (we define efficiency by the amount of instructions required for a computation and the inducted power consumption)

Therefore, a trade-off at the language design has to be made, so that it will remain easy enough for the compiler to understand and produce optimal machine code. The first reference book on compiler design cover (Figure 2-4, from the second edition) [5], featured a dragon fighting with a knight, illustrating the language syntax and associated compiler design challenge

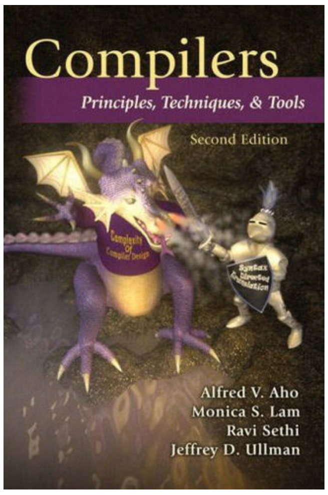


Figure 2-4 "Compilers" book [5] **cover**

2.1.1 The Imperative programming style

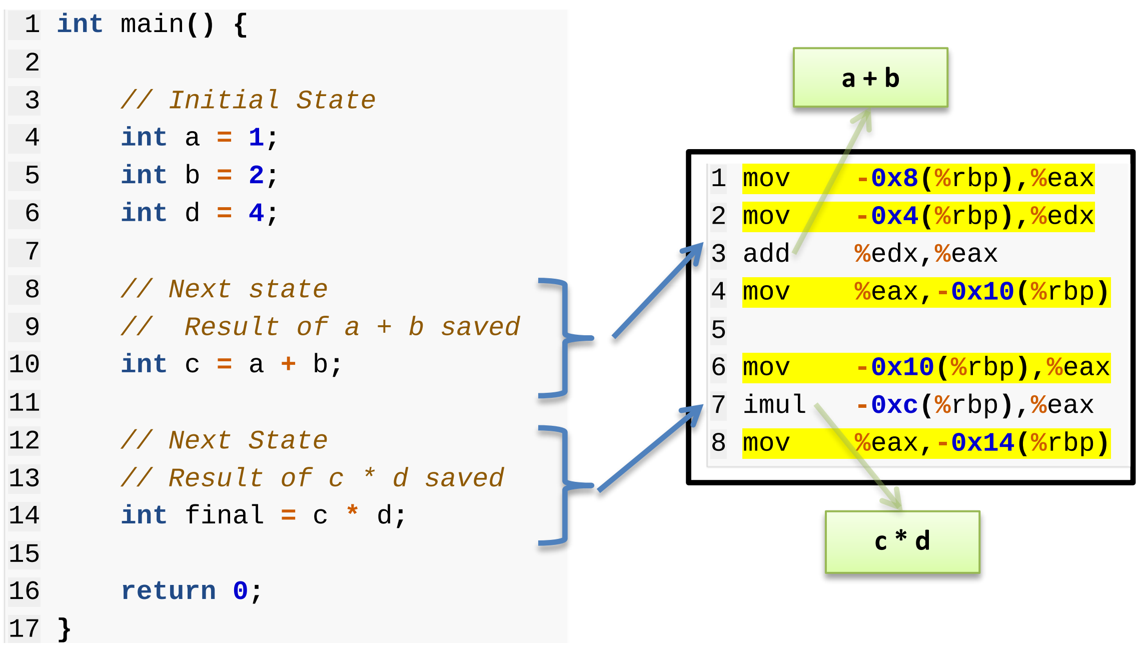
The first developed languages were designed to mirror the sequential control flow of a program, in each of its step. This model is called *imperative programming*, and mostly requires the programmer to describe the computation in its various steps that produce a desired result. This approach makes programming quite close to the underlying computing architecture and machine code instructions (which in turn are all the steps the machine has to go through to produce the desired output), and keeps compiler design complexity acceptable.

To describe all the steps of a computation, the user has to manage two aspects:

- ✓ The state, which is represented by data values hold in memory which allows keeping track of the computation flow.
- ✓ The instructions, which work on the state and update it.

Data and instructions are the base building block of the original Von Neumann computer architecture and its extension the Harvard architecture, in which instructions are executed on data as fast as possible. By expressing the data and instruction flow explicitly, the language limits the possible semantic abstraction, and thus is easily to optimise to run fast on the underlying processor.

To be more concrete, we are going to analyse a trivial program written in C [6], which performs two operations: $c = a + b$ followed by $final = c * d$. The first equation will need an initial state which is defined statically, but could be requested from a user interface. The source code is shown in Figure 2-5 on the left, while on the right side the actual *machine instructions* generated by the compiler (GCC [7] on an x86-64 [8] architecture) is presented. We can see that both source code and machine instructions follow the same flow. On the machine side, ***add*** $(+)$ and ***imul*** $(*)$ are the actual computations performed by the processor, while ***mov*** performs memory copies and relocations which manage the state in the computation machine (yellow highlights).



```c
int main() {

    // Initial State
    int a = 1;
    int b = 2;
    int d = 4;

    // Next state
    //  Result of a + b saved
    int c = a + b;

    // Next State
    // Result of c * d saved
    int final = c * d;

    return 0;
}
```

```
mov     -0x8(%rbp),%eax
mov     -0x4(%rbp),%edx
add     %edx,%eax
mov     %eax,-0x10(%rbp)

mov     -0x10(%rbp),%eax
imul    -0xc(%rbp),%eax
mov     %eax,-0x14(%rbp)
```

Figure 2-5 Imperative C program with associated machine instructions excerpt

This imperative programming style is the most widely used, and many programming languages are designed following this logic. The piece of code we just used as illustration is of course not useful for real applications, and most languages offer advanced features like object-oriented programming [9] and complex design patterns [10] to structure a program and scale it efficiently as it grows.

2.1.2 The Functional programming style

We just defined two aspects of an imperative program: State and Instructions. In other words, we can say that we had to specify:

1. What do we want to reach? This is the $final = (a + b) * d$ specification.
2. How do we reach the goals? This is the way we wrote the source code, i.e. the control flow.

The idea behind Functional Programming is that the user only focuses on writing the composition of functions that will lead to a result, just like solving equations. A function, as pure mathematical object, therefore only produces an output based on a set of immutable inputs. This way, the source code can more closely express the desired result and be safer to manipulate by avoiding state management, like temporary results saving, and forbidding side effects (a function cannot modify its input arguments or surrounding context).

Historically, this concept goes back to the 1930's and research on lambda (λ) calculus. First published by A. Church [11] and extended by A.M. Turing [12], λ-calculus defines a representation of computable functions as anonymous terms, which can be composed to form transformation expressions. Programming models and λ-calculus are extensively presented by Kluge [13], we will only give here the basic notation elements which are relevant to understanding most of λ-calculus' implications in functional programming.

If we consider an algebraic function, defined by an expression $expr$ applied to a set of input parameters $x_1 \dots x_n$, it can be noted :

$$f(x_1 \dots x_n) \rightarrow expr$$

The general form of such a function in λ-notation is:

$$f = \lambda x_1 \ldots x_n . expr \text{ short } \lambda x_1 \ldots x_n . expr$$

- ✓ $x_1 .. x_n$ are variables representing the input parameters of f
- ✓ $expr$ is the function body, which may contain references to the $x_1 \ldots x_n$ input parameters, called free occurrences of $x_1 \ldots x_n$
- ✓ $\lambda x_1 \dots x_n$ is called the binder for the free occurrences of $x_1 \ldots x_n$ present in $expr$. It defines the names of variable occurrences in $expr$ which can be bound to an input parameter.

Applying the function to some input parameters is written:

$$(\lambda x_1 .. x_n . expr \;\; arg_1 \dots arg_n)$$

This notation can be further refined in a n-fold nested form, or curried form, named after Curry and Schönfinkel who introduced it in [14], [15]. They stated that a function f of n arguments can be rewritten as a nested call to n functions of each 1 argument:

$$f = \lambda x_1 \ldots x_n . expr = \lambda x_1 \ldots \lambda x_r \ldots \lambda x_n . expr \text{ for } n \leq r$$

The latter has two important technical implications, both of which are presented using concrete examples in 2.2:

- ✓ Partial function definitions: It is possible to apply f to r arguments, with $1 \leq r < n$. The result is a partial function of f, to which the remaining $1 \leq r' \leq (n - r)$ arguments can be later applied. For example, a simple addition can be performed in two steps:
    - o Create the curried form of $\lambda abc. (a + b + c)$: $\lambda a \lambda b \lambda c. (a + b + c)$
    - o Apply only two input arguments to it: $temp = (\lambda b \lambda c. (a + b + c)\ 2\ 3)$
    - o Apply the last input argument to $temp$: $(\lambda a\ (a + 2 + 3)\ 4)$
    - o The result is 9
- ✓ Function arguments currying: Following the same rules, the formal notation of a function definition can be revisited to:

$$f(x_1 \ldots x_n) \rightarrow expr == f(x_1)(x_2)(\ldots)(x_n) \rightarrow \ expr$$

- ✓ High-Order functions: Each of the curried λ_r functions for $r < n$ is a binder for the x_r variable, and takes as body the next λ_{r+1} function to be applied. A function which takes another function as input argument is called a High-Order function.

The first clear statement that posed Functional Programming as a hierarchy of function composition, in opposition to imperative programming, was made in 1977 by John Backus in "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs" [16], although the very first language that was inspired from λ-calculus is LISP [17], first released in 1958 for the IBM 704 (one of the world's first language together with Fortran).

LISP influenced many programming languages, and is itself at the origin of various forked languages, which sometimes remerged over time. The most notable and still active ones are Common LISP [18] (CLISP, ANSI specification in 1994), Scheme [19] (1975) and Clojure [20] (2007). As we will see later, LISP base concept also inspired TCL [21], presented in 3.1. Other languages inspired from functional concepts are quite popular and still active although less visible to the masses, like Haskell [22] (~1990) or Erlang [23] (~1986), the most recent one being Scala [24](2004), presented in 2.2.

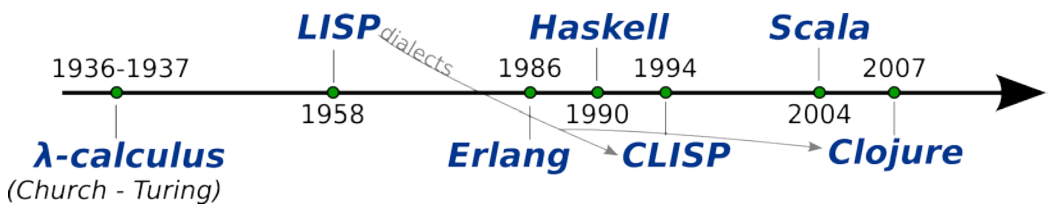


Figure 2-6 Chronology of a few functional programming languages

2.1.2.1 An example in CLISP

To give a glimpse on functional programming style, we will reproduce the example of Figure 2-5 using the Common LISP language. We thus need to adapt our source code to define the composition of functions that describes the computation:

- ✓ Formulate the computation function $(a + b) * d$ in code. LISP uses a *parenthesized polish prefix notation*, which means that operators precede operands. In our case, the operators are $+$ and $*$, with operands $a\ b\ c$, thus $(+a\ b)$ for the addition and $(*\ resultofaddition\ d)$ for the multiplication. Composition precedence is achieved by parenthesizing:

```
(* (+ 2 2) 4)
```

- ✓ In pure functional programming style, there is no state. However, if we want to use variables in our equation, we can do so:
  1. First replace constants by variables

  ```
  (* (+ a b) d)
  ```

  2. Compose the function with the let function, which binds variables to constant values when required

  ```
  (let
      ((a 2) (b 2) (d 4))

      (* (+ a b) d)
  )
  ```

Illustrating the concept of lambda function in CLISP is also very easy, as it is natively supported. We can rewrite our example in the following way:

1. Bind the computation formula to a function definition: $(x, y, z) \to (x + y) * z$

```
(lambda (x y z) (* (+ x y) z))
```

2. Apply the function to a set of input parameters called a, b and d

```
((lambda (x y z) (* (+ x y) z) ) a b d)
```

3. Compose the lambda with the let function, which binds variables to values to have a state to work on. Here *a* is set to 2, *b* to 4 and *d* to 8.

```lisp
(let
    ((a 2) (b 4) (d 8))

    ((lambda (x y z) (* (+ x y) z) ) a b d)

)
```

4. The result of the let call is thus the result of the lambda applied to 2, 4 and 8, so 48.

2.1.2.2 Recursive function call

Recursive function call is a classical concept for all kind of programming style, and is a base building block for functional programming. The factorial function, a typical illustration example, is easy to write both using functional recursive calls and imperative style:

- Recursive decomposition of factorial: $x! \rightarrow (x-1)! * x$
- Sequential decomposition of factorial: $x! \rightarrow 1 * \ldots * (x-n) * \ldots * x\ ;\ 1 < n < x$

Recursive versions in LISP and C are presented in Figure 2-7, as well as imperative style C implementation using a loop (sequential decomposition)

Functional style

```lisp
(defun fact (x)
    (if (<= x 1)
        1
        (* x (fact (- x 1)) )
    )
)
```

CLISP implementation

```c
int fact(int x) {
    if (x<=1) {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
```

C implementation

Imperative style

```c
int result = 1;
int i = 1;
int x = 12
for(i; i<=x;i++) {
    result = result * i;
}
```

C implementation

Figure 2-7 Functional and imperative style factorial

2.1.2.3 List/Elements Array processing

Collections processing is a classical example of how functional programming can be used to write code in a different way. It is indeed very common to work on collection of data in order to find elements matching specific criterions, extract parameters (in case of collections of structured data) etc…. To take a simple example, given a list of numbers, we would like to extract the subset of the even values, and represent them as strings. We define two high-order functions to perform these tasks:

1. <u>Filter</u>: returns the elements of a list, for which the λ function applied to them returned true:

$$f(\lambda x.expr\ \{y_1 \dots y_n\}) \xrightarrow{yields} \{y'_1 \dots y'_r\};\ \ r \leq n\ for\ each\ (\lambda x.expr\ y_v\ ) == true$$

2. <u>Map</u>: returns a list, whose each element is the result of applying a λ function to the input list:

$$f(\lambda x.expr, \{y_1 \dots y_v \dots y_n\}) \xrightarrow{yields} \{y'_1 \dots y'_v \dots y'_n\};\ \forall v\ \ y'_v = (\lambda x.expr\ y_v\ )$$

In imperative programming style, the control flow would have to be implemented per hand as in Figure 2-8, while in functional style, only the composition of functions matters (Figure 2-9):

$$\forall\ x = \{y_1 \dots y_v \dots y_n\}\ ; f(x) \xrightarrow{yields} map(intToString, filter(isEven, x))$$

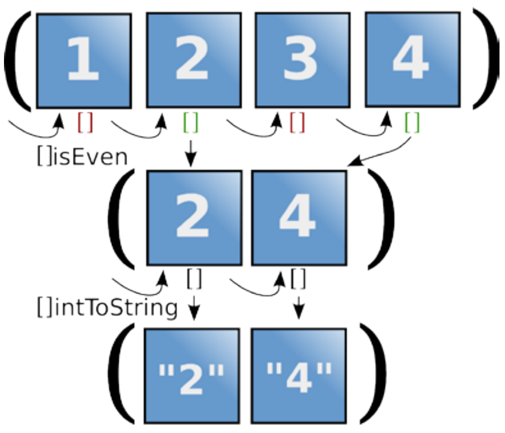


Figure 2-8 List filter and map imperative view

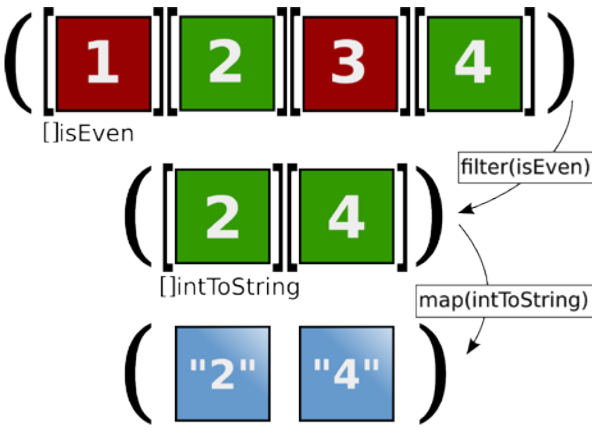


Figure 2-9 List filter and map functional view

A data-flow oriented representation of this composition can be achieved with an Object-Oriented programming interface. A concrete example is shown in 2.2, and refines the previous formula as:

$$\forall\ x = List\ \{y_1 \dots y_v \dots y_n); f(x) \xrightarrow{yields} x.filter(isEven).map(intToString)$$

2.1.3 Discussion

As we have just seen, functional programming abstraction level is very interesting, as it allows the user to write the solution to a particular problem the way it should be described, instead of having to go down specifying the implementation's control flow. However, wide adoption of pure functional programming has not already happened for various reasons:

- ✓ The control flow is hidden

  The natural way to create a software still consists in defining the various features it has to offer, and how they relate to each other. This orchestration requires a minimal state management. Sometimes computations are also very complex, and need an explicit state management to stay understandable by the programmer himself, but also by other human beings. Control flow and computations cannot completely be hidden from each other.

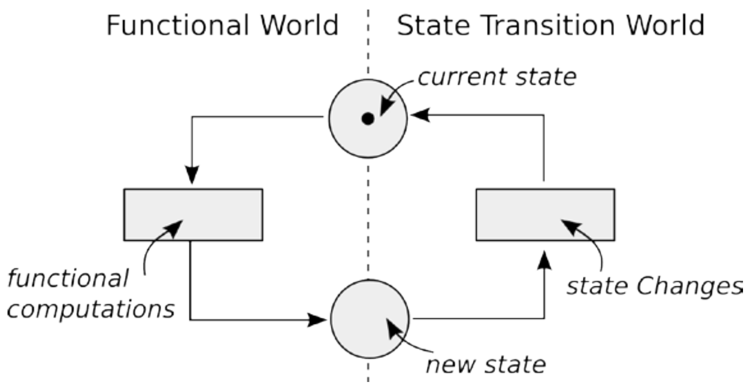


Figure 2-10 "Basic modus of operation of all computational models" [13] **p.75**

  For example, the language Erlang tried to solve this issue by proposing an implementation of an "Actors" model, where some Actor objects exchange messages to trigger reactions [25].

- ✓ Performance

  Functional Programming basically relies on stack execution, as it proceeds by reducing a function tree. Languages like LISP additionally featured paradigms like runtime type checking, which requires an overhead to check function calls arguments before the actual computation. In early computer science days the hardware was slow and expensive, and this lead to requiring the development of machines dedicated to functional languages, like the LISP machine [26].
  For this reason, most programmers desired, or had to keep control of the execution flow of their software to optimise performances, and stick to imperative languages like C. Moreover, the concurrent standard Von Neumann architectures became fast

and cheap, and allowed running LISP programs even faster than dedicated hardware.

- ✓ Syntax

  A strength of imperative style programming is still that reading the code provides understanding of the direct execution flow. In functional style, the result is provided by the composition of functions, which makes it more difficult to understand if the code is large, even if the formal definition is more powerful.

  Moreover, the syntax definition is often a challenge for most users. Dialects of LISP are very uncomfortable to read due to the prefix notation. So stated Alan Kay in his Ph.D. Thesis "programs written in them look like King Burniburiach's letter to the Sumerians done in Babylonian cuneiform" [27]

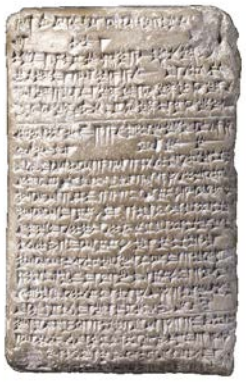


Figure 2-11 Letter from Burniburiach to Amenhotep IV [66]

To summarize, computation algorithms are better expressed in functional style, which in turn is less adapted to architecture definition and orchestration in larger designs. In facts, strict and efficient software design always tries to avoid state management and side effects at the lowest levels, to maximize code stability and reusability. Over time though, some imperative languages started improving their compilers to support functional style constructs. A lot of developers are indeed already applying functional concepts to their imperative code.

A trend can actually be seen in merging of imperative and functional styles, where functional features are used to implement elegant abstraction to problem statements, while imperative state management remains the glue binding the application world together (Figure 2-12).

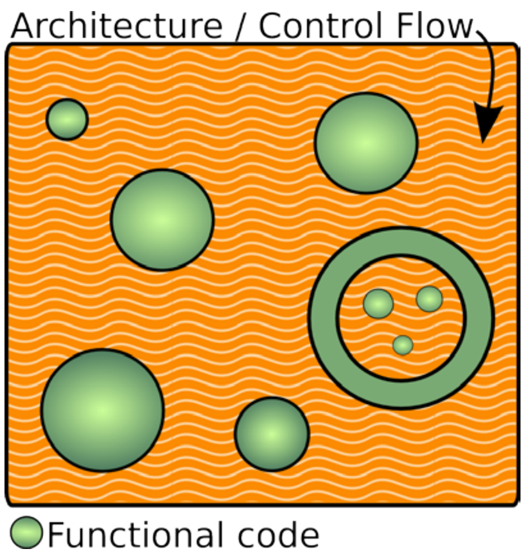


Figure 2-12 Functional islands in an imperative style ocean

Among the most popular languages following this path, we can mention:

- ✓ Python [28], which is very popular, although the functional features are limited and not very attractive as they were added on the existing language definition
- ✓ Groovy [29] has a good visibility, it runs on the Java virtual machine and was designed from ground up with functional features in mind.
- ✓ Even some newer languages, although not designed with functional programming in mind, naturally offer some features. An example of which is the experimental Rust language from the Mozilla foundation (it still may “eat your laundry” according to the official website) [30].

The most visible one over the past few years though is Scala, which we are going to use in section 2.2 to present some important functional programming paradigms applied to an imperative-looking language.

2.2 Merging styles: The Scala programming language example

Created in 2001 (first released in 2004) at the Programming Methods Group [31] of the École Polythechnique de Lausanne (Switzerland) by Martin Odersky's Team, Scala stands for "Scalable Language" [24]. It is a new programming language whose main design road is to unite imperative with functional programming paradigms introduced by earlier languages (LISP, Haskell, SmallTalk etc...).

The starting point of Scala's design is based on three assumptions:

- ✓ The programmer writes too many keywords for obvious statements (so-called boiler-plate code).
- ✓ Functional paradigms can be integrated in the language in an elegant way
- ✓ One can Import elegant concepts from existing languages and improve them, without reinventing something totally new each time.

Scala features its own compiler which mainly targets the Java Virtual Machine (JVM) for runtime, and allows taking advantage of the wide base of existing Java libraries and projects. It can be discussed if Scala could be the next Java, but the latter follows its own path, and the basic syntax of Scala can be repulsive for programmers used to state of the art C/C++ or Java-like languages. In the next sections, we are going to present a few features of Scala which proves the flexibility it brings to traditional imperative programming, without requiring the user to think in an unthinkable way.

2.2.1 Type definition and Type Inference

2.2.1.1 Type Inference

In Scala, the data type definition of a term follows it's name, unlike most usual imperative languages (C/Java etc...).

$$TERM\ NAME : TYPE = expr$$

The $TERM$ can be:

- ✓ A variable : $var\ NAME : TYPE = expr$
- ✓ A value (or constant) : $val\ NAME : TYPE = expr$
- ✓ A function definition: def $NAME\ (ARGS) : TYPE = expr$

A type checker typically checks the equivalence of type between the term specification, and the expression: $type(expr) === TYPE$. By adding a type inference mechanism, the $TYPE$ specification can be dropped and delegated to the type of the expression: $type(expr) \xrightarrow{yields} TYPE$. The common term specification becomes:

$$TERM\ DEF\ (:\ TYPE)? = expr$$

A few concrete examples for simple numerical data types are presented below (simply run in the Scala interpreter):

```
scala> var a : Int = 42
a: Int = 42

scala> var a = 42
a: Int = 42

scala> var b = 42L
b: Long = 42

scala> var c = 42.0
c: Double = 42.0
```

2.2.1.2 *Implicits*

The type inference mechanism in Scala is not static, as presented in Figure 2-13, and can be enriched by user provided conversion functions, called *implicits*, which simply take an input of type A to produce an output of type B (i.e. specifies an $A \rightarrow B$ conversion) . This feature is of great use to create flexible language interfaces, although it can sometimes lead to bad designs if too generic and possibly clashing implicit conversion functions are defined.

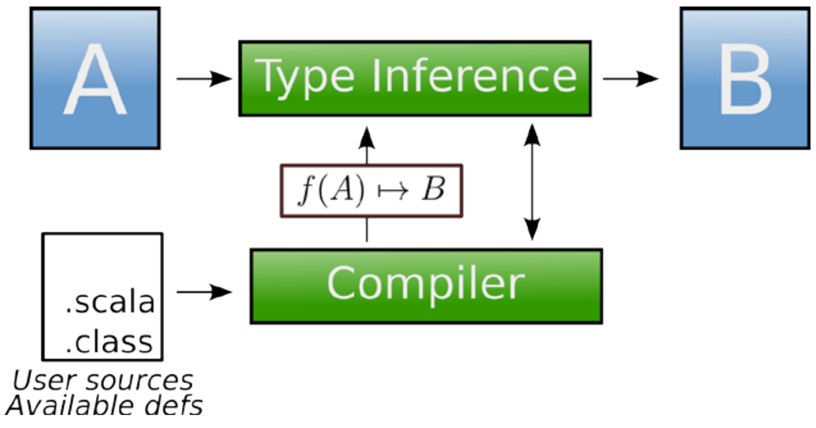
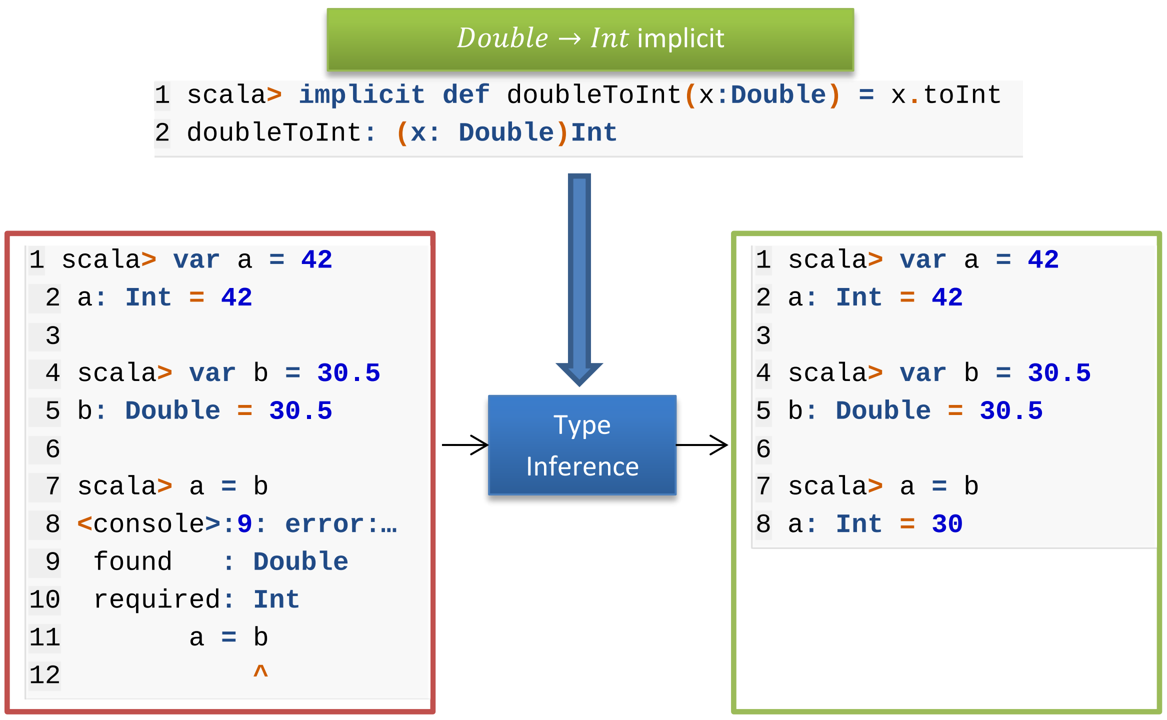


Figure 2-13 Type inference with dynamic type conversion

To illustrate this mechanism, let's consider the data types Integer and Float. The $Integer \rightarrow Float$ conversion is trivial, as an Integer can be represented as a floating point number with a mantis set to 0. The other way round is not possible, and requires an explicit rounding. To simplify an application where a rounding to the lowest integer would be the rule, we could define an implicit. Figure 2-14 provides the illustration, with an invalid Double to Integer assignment on the left-hand side made valid on the right-hand side by the implicit definition.



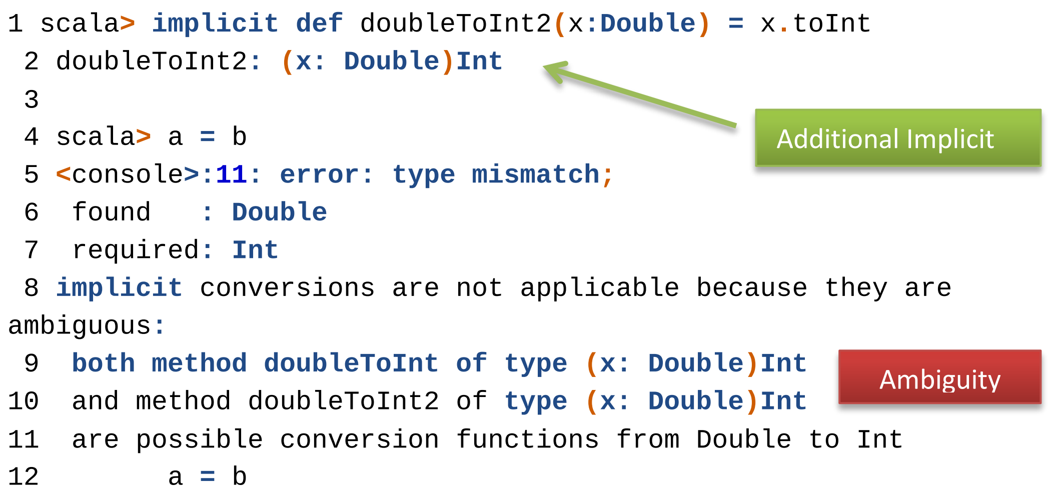
```
scala> implicit def doubleToInt(x:Double) = x.toInt
doubleToInt: (x: Double)Int
```

```
scala> var a = 42
 a: Int = 42

 scala> var b = 30.5
 b: Double = 30.5

 scala> a = b
 <console>:9: error:…
  found    : Double
  required: Int
        a = b
            ^
```

```
scala> var a = 42
a: Int = 42

scala> var b = 30.5
b: Double = 30.5

scala> a = b
a: Int = 30
```

Figure 2-14 Double-to-Int implicit type inferring example

But if another application part or a library had decided to provide its own inference function, we might run into ambiguity. In the following example, the type inference mechanism cannot decide if the *Int* to *Double* conversion should be performed by the *doubleToInt* method, or the newly added *doubleToInt*.



```
scala> implicit def doubleToInt2(x:Double) = x.toInt
 doubleToInt2: (x: Double)Int

 scala> a = b
 <console>:11: error: type mismatch;
  found    : Double
  required: Int
 implicit conversions are not applicable because they are
ambiguous:
  both method doubleToInt of type (x: Double)Int
  and method doubleToInt2 of type (x: Double)Int
  are possible conversion functions from Double to Int
        a = b
```

Figure 2-15 Double to Int ambiguous erroneous implicit type inferring

2.2.2 Closures and high-order functions

2.2.2.1 Anonymous functions

Functions are in functional programming “first class citizens”. They can be declared anonymously, as λ-expressions (i.e. inline without the ***def*** keyword), and be considered as simple values. The Scala syntax follows the function definition presented in 2.1.2 (→ is replaced by ⇒):

$$f = \{\,(args)\ \Rightarrow\ expr\}$$

The following example creates a λ expression which multiplies two integers:

```scala
var multiply = {
    (a:Int,by:Int) => a*by
}
multiply(2,2)
```

Formal λ specification

```
scala> var multiply = {
     |     (a:Int,by:Int) => a*by
     | }
multiply: (Int, Int) => Int = <function1>

scala> multiply(2,2)
res1: Int = 4
```

2.2.2.2 Closures

Closures are a key construct in functional programming. A good and understandable definition can be given based on [32] and [33]:

> **Definition 2.1** A closure is a λ-expression associated with an environment, which may contain occurrences of variables bound to the environment, not to the λ binder (i.e. the input arguments)

This next example defines a λ-expression which multiplies an integer by a coefficient defined in the environment surrounding the closure (*by* is not a *multiply* input parameter):

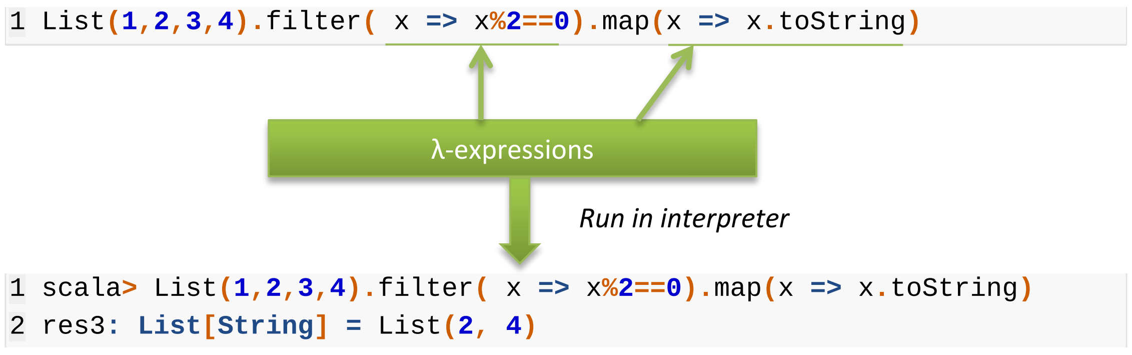
```scala
var by = 2
var multiply = {
    a:Int => a*by
}
multiply(2)
```

```
scala> var by = 2
by: Int = 2

scala> var multiply = {
     |     a:Int => a*by
     | }
multiply: Int => Int = <function1>

scala> multiply(2)
res2: Int = 4
```

2.2.2.3 *High-order functions*

Definition 2.2 A high order function is a function which takes a λ-expression (i.e. another function) as one of its input arguments, or returns one as result.

A very recurrent example is encountered when processing collection of elements. In Figure 2-16 we implemented the list processing example from 2.1.2.3.



```
List(1,2,3,4).filter( x => x%2==0).map(x => x.toString)
```

```
scala> List(1,2,3,4).filter( x => x%2==0).map(x => x.toString)
res3: List[String] = List(2, 4)
```

Figure 2-16 List filter and map in Scala

2.2.3 Currying and Partial Functions

Currying is implemented in Scala and allows splitting the input arguments of a function definition to take the form of nested call, as presented in 2.1.2.

$$\text{def } NAME(\text{arg}_1 \quad \ldots \text{arg}_n \quad) \rightarrow \text{def } NAME(\text{arg}_1 \quad) \ldots (\text{arg}_n \quad)$$

```
def multiply(a:Int)(b:Int) = a*b
```

Curried Call
Multiple arguments call forbidden

```
scala> multiply(2)(2)
res5: Int = 4
```

```
scala> multiply(2,2)
<console>:11: error: too many
arguments for method multiply: (a:
Int)(b: Int)Int
              multiply(2,2)
                       ^
```

Figure 2-17 A multiply function written in curried form

Figure 2-17 presents the previous multiply function definition example rewritten in a curried form. The other consequence of currying that was mentioned earlier is that it allows using partial function calls:

```
scala> var p = multiply(2)_
p: Int => Int = <function1>

scala> p(4)
res6: Int = 8
```

P is now a partial λ of multiply: $(x) \rightarrow 2 * x \text{ or } \lambda.x.2 * x$

Apply p to 4 $(\lambda.x.2 * x \quad 4)$

However a return on experience showed that partial calls are rarely used, but can be very convenient to raise the abstraction level by making a curried function call with default values for the first arguments, while hiding them from the final call. One could write a multiply by two function, which would be a partial call to multiply with $a = 2$, waiting for the final call to have b bound. As can be seen in figure, the multiply by 2 function is not a new definition, but a partial from the generic multiply definition.

```
scala> def multiplyBy2 = multiply(2)_
mby2: Int => Int

scala> multiplyBy2(4)
res13: Int = 8
```

Figure 2-18 Partial function usage to introduce an abstraction level

2.2.4 Discussion

Scala being a young language, some critics have been raised against it. One of the most prominent, which can be sensed after reading this section, is that the compiler gets assigned a lot of tasks, by trying to support many features. Type checking and inference slows down compilation and can lead to confusions, real generic runtime language reflection is a pain and not clearly specified (users have to use the standard Java introspection instead) and so on…

On the other hand, the language design choices are clearly targeted at giving the developer the tools he needs to use and not defining constructs which reduces errors possibilities by constraint. More pressure is put on the designer to think about clear and correct way to write code, rather than solving problems using a syntax not adapted and not adaptable. The science in Scala resides in finding the sweet spot between imperative flow, functional constructs and code clarity (which impacts robustness).

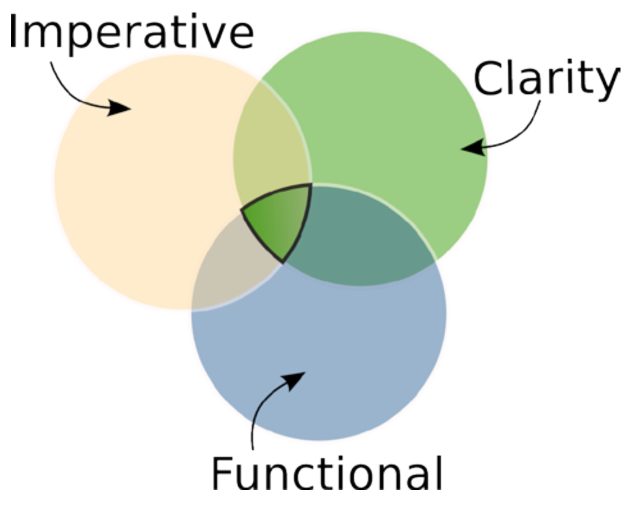


Figure 2-19 Finding the code quality sweet-spot in Scala

A Typical example for this is the type inference mechanism. Not specifying explicit types in the source code makes debugging more difficult, but Scala doesn't force anyone to not specify the type. That kind of decision is given back to the programmer, who needs to think for example:

- ➔ Is the type obvious?
    - o Yes: Let type inference work
    - o No: Specify the type
- ➔ Should I make the type obvious?
    - o Yes: Specify the type, and provide type inference specification for the end-user
    - o No: Specify the type

This kind of options is not thinkable in Java for example, where the type must be explicit and is non flexible.

During this thesis, some projects have been developed using Scala, on of them being the XML binding library presented in 4.4. Many lines of code have been written and we found true that syntax lightening, architecture design features (traits, object etc…) and functional paradigms (closure, type inference) brings an important speed-up in implementation phases, giving more time for unit testing, consequently improving applications stability.

To conclude, we can state that functional programming presents tremendous improvements for designer choices, if correctly flavoured in a traditional language. However, the way functions are composed with each other, especially in the case of closures for data flow programming, requires a change in the way we think algorithms. Developers need an adaptation period, but experience also shows that the benefits gained from using Scala strictly as an imperative language are quite limited. It only brings the language closer to Dynamic Languages because of its light syntax, while lacking their flexibility.

In the next two following sections, we are going to explore two techniques for abstract design language: Domain Specific Language and Embedded Domain Specific Language. Scala will be used as support for implementation examples in both cases.

2.3 Domain Specific Language design: LL and LR-based parsing

In our quest to optimizing software development methodologies for domain specific applications, we need to present some parsing algorithms which are often used to create languages. Programing languages usually try to follow a limited set of constructs, which are semantically different enough from each other to make parsing by compilers feasible and possibly with a complexity approaching $O(n)$. These languages thus usually are context-free languages in the Chomsky-Hierarchy [34].

Context-free languages are defined by grammars, which describe the allowed words for a language. A grammar is written as a set of rules, which a parser uses to determine if a character input is part of the defined language or not. Besides accepting or rejecting an input, parsers are used to produce an output, which can be for example an abstract representation of the input, or a transformation. The output creation is driven by actions run when encountering specific language constructs.

Analysing a very simple example is the best way to understand the way parsers work. We present in Figure 2-20 a C language *if* construct, which is parsed and transformed to generate an in-memory tree representation of the input, called an Abstract Syntax Tree (AST). This AST representation of the text input can then be processed by other software components like compilers, optimisers etc…

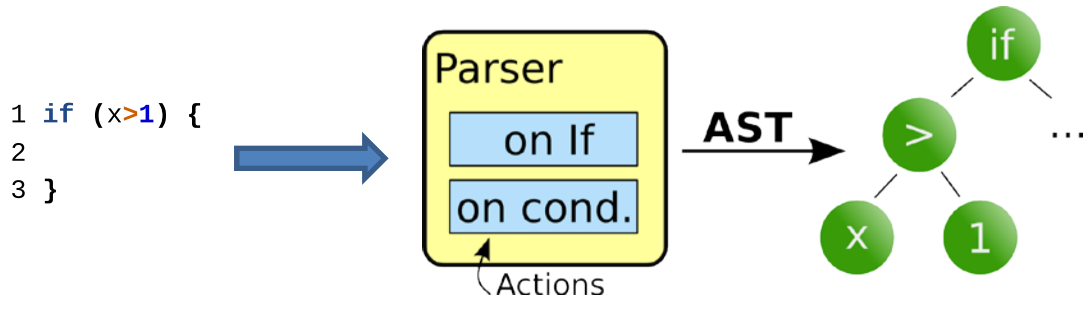


Figure 2-20 A simple C "if" parsing rewritten in an AST form

How can we parse such an input?

We are going to briefly focus here on *LL(k)* and *LR(k)* parsers [35], which are two kind of algorithms designed for context-free languages. They work by reading an input from the left to the right, with a *k* number of look-ahead characters in their buffer. A language is said to be deterministic *LL(k)* /*LR(k)* defined if it exists an *LL(k)* /*LR(k)* parser that can recognises it without backtracking, that is to say, without having to rewind the stream to try another parsing path. *LL(k)* /*LR(k)* both implement two different approaches:

✓ An *LR* parser produces a right-most derivation, meaning that the terminal grammar rules are first matched then reduced to find the top most rule.

- ✓ An *LL* parser produces a left-most derivation, meaning that the top level grammar rule to follow is predicted based on the input look-ahead, which is then compared to the expected following rules.

LR(k) parsers performances are linear in time, and generically support more languages than *LL(k)* ones (as there is no prediction issue), and are more prone to error detection because they always wait for a grammar pattern to be fully recognised before "pushing the result up". However, they are more difficult to write than *LL(k)* parsers, and their complexity rises in front of languages which would need to rely on look-ahead and prediction to decide which rule path must be followed.

LL and LR applied to Domain specific languages

Fortunately, *LL(k)* and *LR(k)* parsers can be generated from a textual representation of a grammar in a Backus Normal Form (BNF, first introduced in [36]). Creating a new language can thus be as simple as writing a definition grammar and generate the parsing code. Two of the most famous parser generators tools are *YACC+Lex* for *LR* parsers and *ANTLR* for *LL* parsers, the later of which we are going to present. The Scala language also features an interesting simple *LL(k)* API which allows to represent a grammar using a composition of functions and objects, without *BNF* notation and generation of parser code.

To better understand how parser generators work, it is interesting to note that parsers work on a single character input basis, while languages usually present lexical elements, or tokens, which are composed of multiple characters (like *if* in our example). It is thus efficient to pre-parse a character input in order to represent those multiple characters tokens as single "virtual" characters.

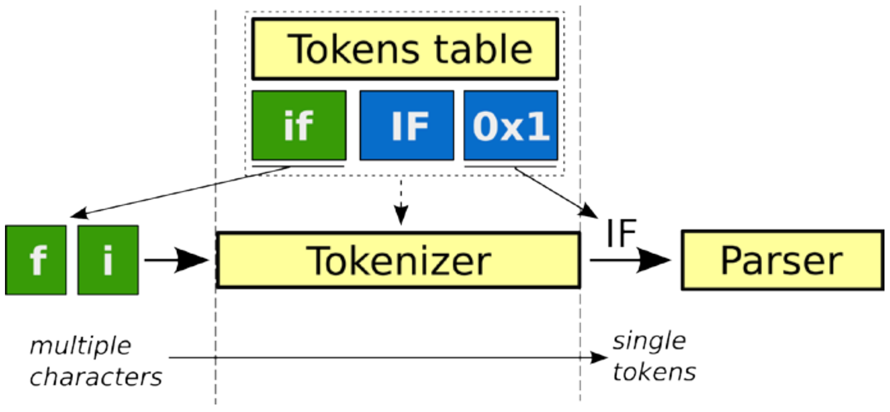


Figure 2-21 Tokenizer output to parser for "if" characters

As presented in Figure 2-21, a parsing chain thus normally first converts the textual representation of an input stream to a "tokenized" stream, using a lexical analyser and a token table. This stream is then fed to the parser which only sees single "characters".

2.3.1 LL Parsing in Java: ANTLR

ANTLR is a very popular *LL(k)* lexical analyser and parser generator [37][38] targeted at the Java Programming Language (some backend exist for other languages like C/C++ but they are not supported from ground up). It introduces the concept of *LL(*)* parsers, for which the number of look-ahead characters *k* is not fixed. Additionally, to help tackle the weaknesses of *LL* parsers against left recursive languages and context-sensitive constructs, it introduces syntax and semantic predicates, which allow the user to define rules whose matching results drive the choice of the main grammar rule path to be followed.

Context sensitivity and left recursion issues can become quite cumbersome to solve, but *ANTLR* is backed by a powerful set of tools to analyse and debug a language. To illustrate the usage of *ANTLR*, we wrote a small grammar supporting the *if* construct presented earlier. It has been improved to support recursive condition expressions. As showed in Figure 2-22, the condition for *if* can be represented as $OPERAND\ OPERATOR\ OPERAND$. The $OPERATOR$ element is defined by the language, and will be for example an arithmetic operator like +, &, < or >. The $OPERAND$ can in turn itself be an $OPERAND\ OPERATOR\ OPERAND$ sub-expression, until it is only a sole identifier, like a variable name or a constant.

```
if ( (x > 1) & (x < 10) ) {

}
```

Figure 2-22 Simple If definition with recursive condition

The parsing tree produced against this input by the grammar definition, using the ANTLRWorks 2 debugging environment is show in Figure 2-23, while Figure 2-24 presents:

- ✓ The matching grammar in *BNF* form, containing parser rules, lexical analyser definitions, and a Java code action run at the end of the IF matching rule
- ✓ Diagram views of the *if* and *args* rules, created in ANTLRWorks.

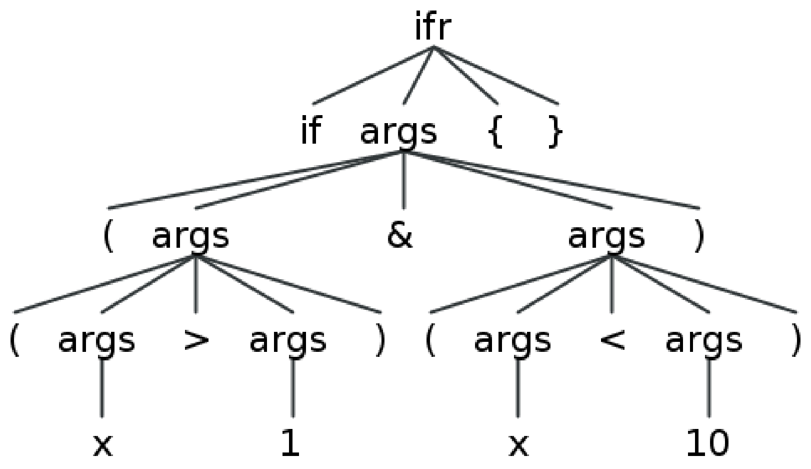
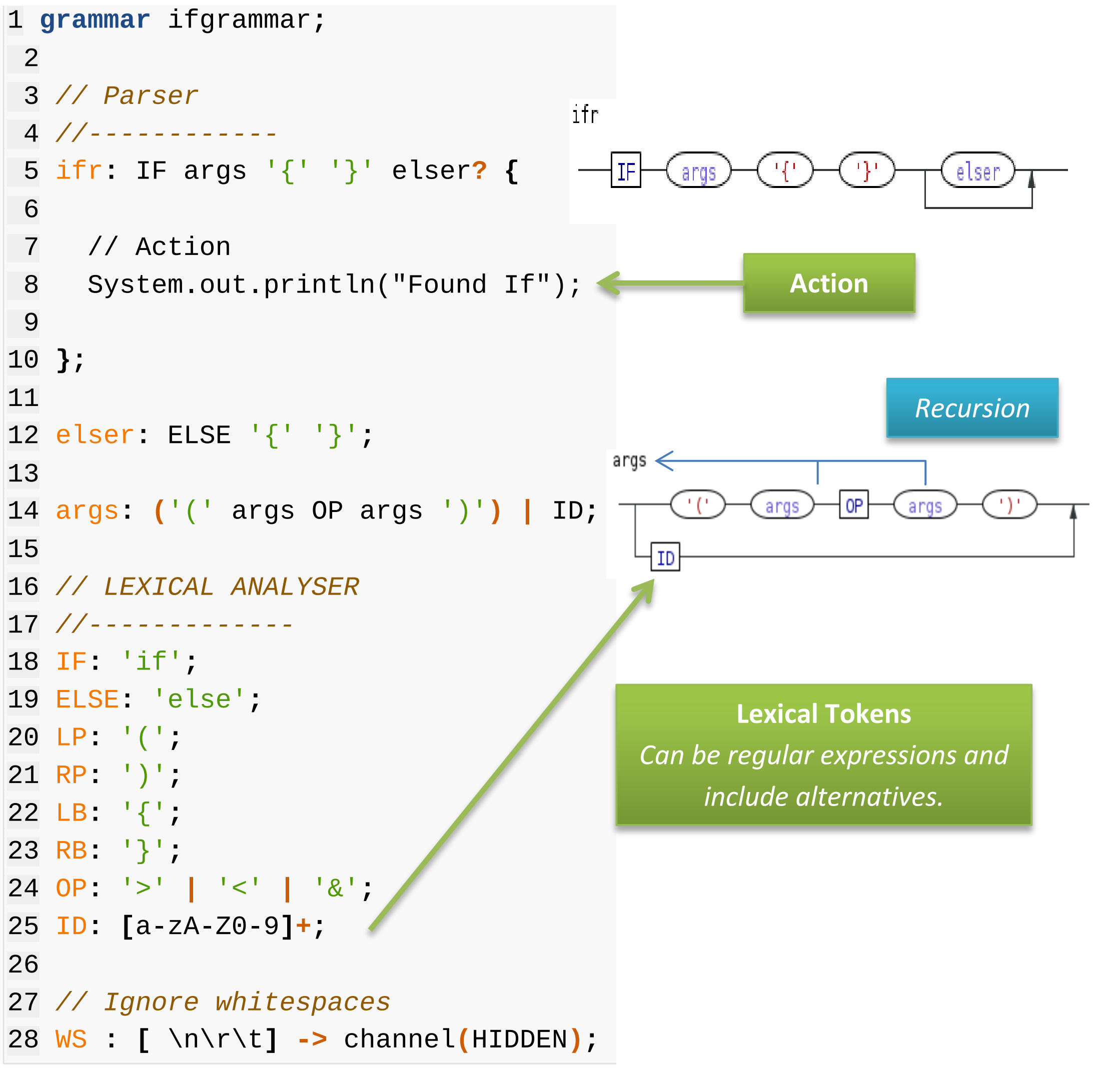


Figure 2-23 If grammar parse tree for a given input



```
grammar ifgrammar;

// Parser
//------------
ifr: IF args '{' '}' elser? {

  // Action
  System.out.println("Found If");

};

elser: ELSE '{' '}';

args: ('(' args OP args ')') | ID;

// LEXICAL ANALYSER
//-------------
IF: 'if';
ELSE: 'else';
LP: '(';
RP: ')';
LB: '{';
RB: '}';
OP: '>' | '<' | '&';
ID: [a-zA-Z0-9]+;

// Ignore whitespaces
WS : [ \n\r\t] -> channel(HIDDEN);
```

Figure 2-24 ANTLR Grammar with syntax diagrams

The lexical analyser rules, like *IF*, convert sub-parts of the text input to single-values as presented in Figure 2-21. The *args* definition is the recursive a parser rule, while the *ifr* rule contains a Java code action between curly braces, which is executed after a completed match.

ANTLR proves to be easy to use and its good tooling support makes it a good choice to develop languages in a short time. However, it mostly limited to the Java programming language environment, although some generators exist for other languages, their support and quality is not guaranteed.

2.3.2 Parsing in Scala

Both previously presented *ANTLR* and YACC+Lex generate a parser source code for a grammar described in *BNF* form. Taking advantage of its advanced type inferring features (see 2.2.1), the Scala language provides a parser library in its standard distribution[39]. It is not a generator-based parser, which means that the user instantiates a set of generic objects which basically integrate and repeat the logic a parser would generate for each a new input parsing process.

This strategy allows embedding parser code right where it belongs in the sources, with no additional tool-chain, but lacks the separate language design stage, and no specific debugging environment is available at the moment. Additionally, the Scala parsers are not like *ANTLR* or YACC *LR(k)* /*LL(*)* designs, but simple recursive top-down parsers which employ backtracking (stream rewind) for alternative decisions. For example, if a rule is written: $S \rightarrow P|\ Q$, matching Q will first require to fail matching P, then rewind and match Q.

To show how this performs, the following code example implements the *args* rule from previous ANTLR example using a Scala parser:

```scala
var argsr : Parser[_] =

// (      arg          OP          arg      )    |        ID
  "(" ~ argsr ~ ( ">" | "&" ) ~ argsr ~ ")"  | """[a-zA-Z0-9]+""".r

parseAll(argsr, "(x > 1) & (x < 10) ")
```

We can see in this case, that the Lexical tokens (> and & for example) and the rules (*argsr*) are chained directly using Scala function operators (~ or /) . For example, some implicit type inferring rules are defined in the Parser library to convert the String data types (ex: *"("*) to Parser objects which recognise the converted string. Using this mechanism, the developer can ignore the actual parsing class hierarchy, and focus on the rules content.

The Scala parsing API is well adapted for applications requiring simple parsing with flexible integration, and with little performance requirements. Those criterions should match a lot of use cases, but as we will see in 2.4, parsers are quite often not really required for language development in Scala, because the language itself is flexible enough to avoid requiring DSL parsers.

2.3.3 Discussion

Domain Specific Languages basically follow the principle of standard programming languages design, which involves defining a grammar and an associated parser to produce an output. Instead of producing runnable applications, DSLs typically are parsed within an existing one, where the embedded actions are used for control and configuration purpose, or to produce data structures to be used later by the runtime.

However, a few criterions need to be analysed when determining is a DSL should be considered for a precise application:

- ✓ Language design: language grammars are not so trivial to design, because when using a DSL to raise the semantic abstraction level of the language, the user rapidly falls in context-sensitivity and recursion issues which can lead to long parsing times, high development costs, and moreover, the testing complexity raises rapidly with the number of allowed language constructs.

- ✓ Supported Runtime: In the end, as we have seen with *YACC+Lex* (for C), *ANTLR* (for Java) and *Scala*, the parser generators tend to be adapted to one underlying language, making the choice of the parsing technology dependent on the chosen application language.

- ✓ Performance: Depending on the size and complexity of the typical input to be parsed, YACC+Lex/ANLTR or Scala behave differently. When Parsing small sized inputs, the performances are comparable, but as the input data grows, a linear *LR* parser performs better, assuming it can parse the language. A less optimised Scala parser will behave badly for large inputs, but similarly to generated parsers for small inputs. The performance criterion must also be compared to the context of usage. A DSL driving very long running processes in the application runtime won't require to be very performant for example, and the first-choice will be driven by the underlying runtime requirement (Native application, Java-based etc…) and the maintenance cost function.

In the next section, an alternative way to create domain specific languages, called Embedded Domain Specific Language will be presented. It differentiates from traditional parsing-based language design by using concepts from Functional Programming languages, to reach a syntax quality close to DSLs without parsers.

2.4 Embedded Domain Specific Language (EDSL)

Exploration of Functional Programming and language parsing technologies gives us so far knowledge about different ways to structure traditional code, and ways to define new language semantics. Still, most of the flexibility we want to reach using Domain Specific Language and Functional Programming revolves around hierarchy of problems (to complete control flow based or imperative programming, which focuses on sequence), and usually an existing application exists and constraints the technology choice for DSL development.

Embedded Domain Specific Languages, as presented by Paul Hudak in [40], serve the same purpose as traditional DSL design, but instead of parsing a language input on which reactions will trigger a behaviour in the host application, the process is reversed by defining the behaviour of the created language using functions, and try to allow valid function calls in the host language which will look like a new language. This approach presents one main advantage and one main drawback:

- ✓ Be embedded in the host language: The new definitions only need to focus on the core aspects of the language. Traditional control structures, for example, are already present and the user can mix the language elements with other libraries.
- ✓ Be embedded in the host language: Some limitations in the allowed syntax will be required to fit the host language, and the language usage validity mostly has to be proven by the implementation, as no parser can be configure to forbid specific semantic combinations.

In our sense, the main criterions we will retain for EDSL design should lead to reaching a Software development cost as presented in Figure 2-25 (base graphic from [40]):

- ✓ Ease of development: $(C2 - C1) \rightarrow 0$, better maintainability in time than DSL methodology
- ✓ Clear Syntax
- ✓ Hierarchy mirroring

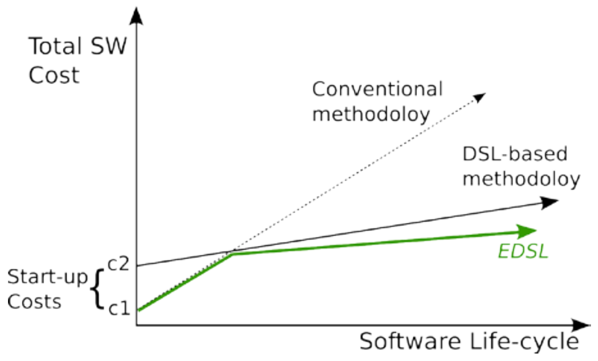


Figure 2-25 Domain specific language cost gain [40]**, with EDSL projection**

To illustrate our purpose, we are going to analyse a simple concrete example of how to build a DSL to create a Graphical User Interface (GUI), and then analyse the creation of an EDSL extension to the Scala language.

GUIs are typically component graphs with actions (like button clicks), in other words: A component hierarchy with function composition. Figure 2-26 shows a window with two buttons which we want to be able to design efficiently. This first step is to define in clear text what we see:

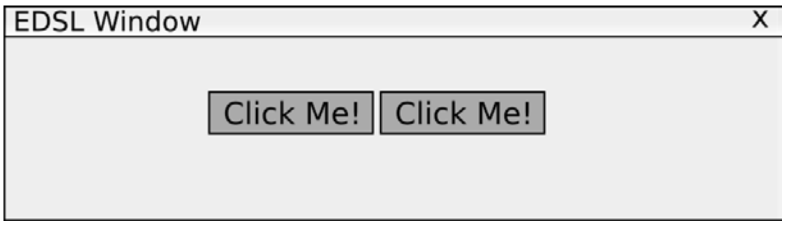


Figure 2-26 Window output of Designed DSL

- ✓ Alternative 1 : The buttons placement is dynamic
  - o A window
    - ▪ A Content Panel
      - A button
      - Another button
      - The panel is laid out using an Horizontal Box algorithm
- ✓ Alternative 2: The placement of the buttons is static
  - o A window
    - ▪ An horizontal box
      - A button
      - Another Button

From this perspective, we can try to write the two alternatives down using a new language:

```
window("EDSL Window") {

   panel {

      button("Click Me!") {
         println("Hello World!")
      }
      button("Click Me!")

      layout = hbox

     }

}
```

Alternative 1

```
window("EDSL Window") {

   hbox {

      button("Click Me!") {
       println("Hello World!")
      }
      button("Click Me!")
   }
}
```

Alternative 2

Using tools presented in 2.3, we could now write a grammar definition to parse the input. However, some interesting issues can already be seen:

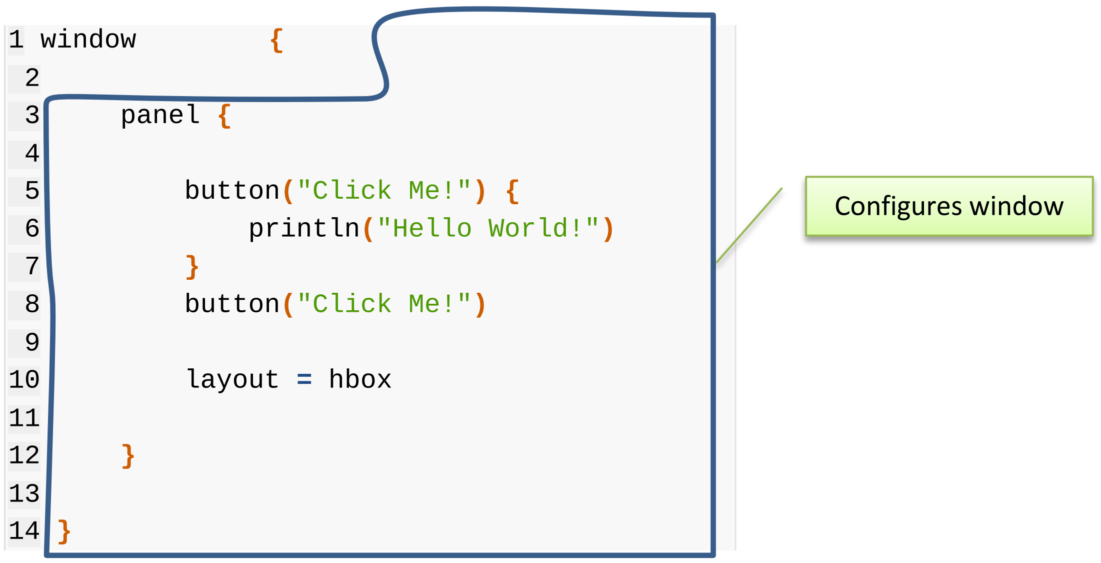
- ✓ The parser actions must build the hierarchy using a stack. The software complexity is just hidden and moved to the parser maintainer.
- ✓ Some code is embedded for the buttons actions. This case is very problematic, as it forces to define a programming syntax, or handle action nodes as simple text, and be able to evaluate it dynamically. This is feasible, but poses the issue of context binding, i.e. with which environment can the action code (button click) interact. Concretely, before running the action code, the application would have to explicitly bind some variables with which it can interact to modify the global application environment.
- ✓ Do the curly-braces-surrounded section after a button means "onClick", or can it be used to configure the button (colour, other kind of listeners etc…) ?

Those few issues illustrate the difficulty to design languages where the output has to be flexible and context sensitive interpretation is required. The efforts to maintain a classical DSL would be in this case for such a simple problem overwhelming. Can we from there on find another path?

2.4.1 Functional Programming for EDSL

Basically, under the light of [40] or [41] (a design applied to Scala) , we can review the hierarchy of the Alternative 1 example as:

- ✓ A window
    - o A function to configure the window
        - The window title is “EDSL Window”
        - A panel
            - A function to configure the panel
                - o A button
                - o Another button
                - o The panel is laid-out using an Horizontal Box algorithm



```
window        {

    panel {

        button("Click Me!") {
            println("Hello World!")
        }
        button("Click Me!")

        layout = hbox

    }

}
```

Figure 2-27 Function + Configuration closure design pattern

In Figure 2-27 we can first recognise a typical λ -calculus pattern: Currying (see 2.2.3). The ***window*** function creates a Window and accepts another function as input, which will be applied to the new window for configuration.

In a second time, the presented example delegates the actual hierarchy creation to the underlying runtime. Indeed, no explicit keyword or function call explicitly places the panel in the window, and the buttons in the panel. This has to be done by maintaining the hierarchy stack while creating the components, following this simple flow:

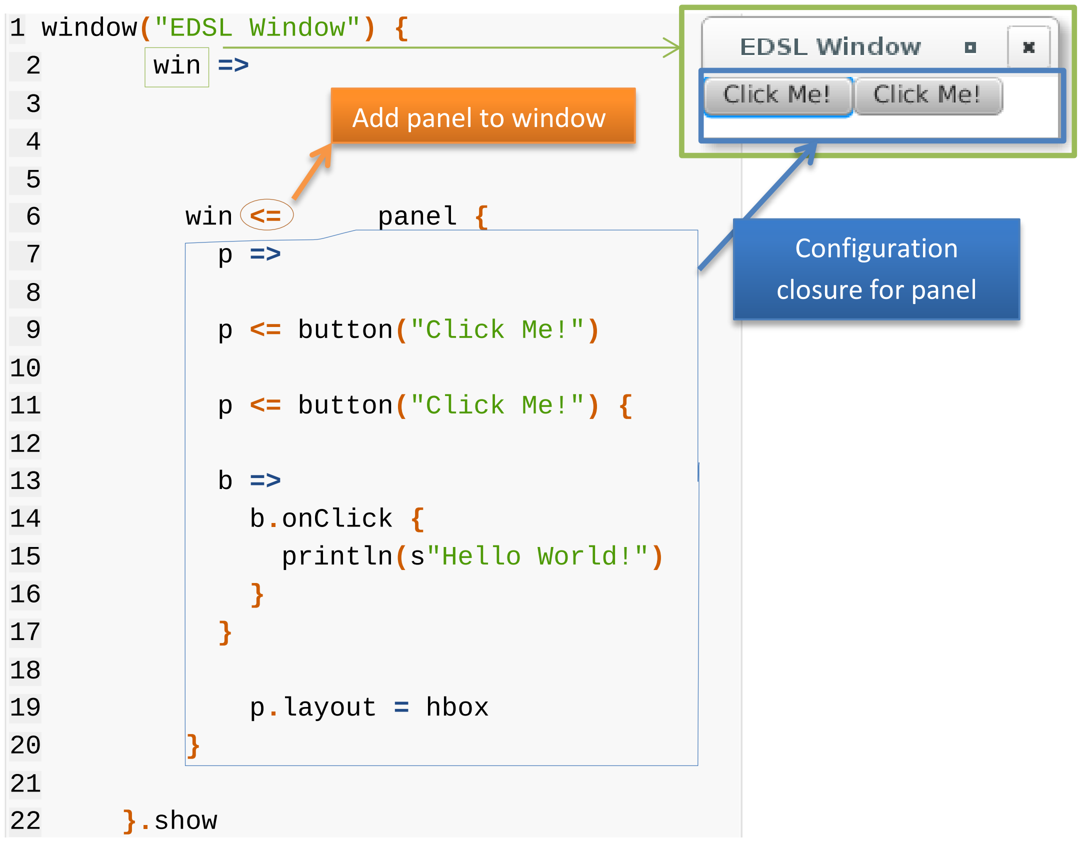
1. Create new component (the panel for example)
2. Stack
3. Execute the e configuration closure
4. De-stack

Is this always true?

Various options exist depending on the type of language that is used for implementation. Basically, in the context of a compiled language, the code must be fully consistent for the compiler to validate the function calls have a clear and existing reference. This leads to two design cases:

- The hierarchy building is delegated to the runtime. This option we just described. It has the main advantage of reducing the required explicit coding, but introduces potential runtime errors.
- The hierarchy is created explicitly by appropriate function calls. This makes code writing more difficult, but triggers clearer results, and better compile-time validation.

This second option is illustrated by a concrete implementation example in Figure 2-28, where the hierarchy is constructed by calls to functions named “<=”. It is based on a Scala library created to virtualise user interface creation in an appropriate programming interface, enable faster development time, while delegating the real component creation to the underlying GUI library (Java Swing or JavaFX for example). We can see that it is a bit more expressive than the proposed versions so far, but answers all the presented issues.



```
window("EDSL Window") {
    win =>



     win <=      panel {
      p =>

      p <= button("Click Me!")

      p <= button("Click Me!") {

      b =>
       b.onClick {
        println(s"Hello World!")
       }
      }

       p.layout = hbox
     }

   }.show
```

Figure 2-28 Embedded DSL for a GUI applied to Scala (JavaFX runtime)

On the contrary, when using a dynamic (or scripted) language, it is sometimes possible to create a hybrid solution. The configuration closures would be interpreted when encountered, and thus do not need to be compiled before being passed to the object where it will be executed. In this case, the closure correctness must not be enforced at the level where it is created, but at the level where it is executed. This allows a lighter syntax without delegating any hierarchy building to a special runtime handler.

To outline the requirements of such a dynamic language design, let's have a look at the button creation in our example. The code would look like:

```
button("Click Me!") {

    onClick {
      println(s"Hello World!")
    }
}
```

The formal software design requires:

- ✓ A *Button* Class with
  - o An *onClick* method
- ✓ The *button* function:
  - o Creates a *Button* instance
  - o Runs the closure in the context of (i.e. local to) the button instance
    - ▪ The *onClick* method call is valid because it is then local to the closure evaluation context.

An important limitation is however to be highlighted in this case: it requires the runtime to be able to capture the closure code and defer its execution into another run level. This is not always possible depending on the underlying language.

2.4.2 Discussion

This section illustrated the usage that can be made of functional programming high-order functions and closures to embedded “domain-specific constructs” as part of a language. This approach is, naturally, radically different from a pure Domain Specific Language definition, as it constraints the newly defined abstraction level to a specific language. However, benefitting from the host language features, evolutions, and potential community-provided libraries is an advantage (by itself in terms of design cost) that greatly supersedes the full flexibility of a traditional DSL.

A concrete example has been provided in this section using the Scala language, which provides the adequate syntax and language features to implement an EDSL with style. This is not true for all existing languages, and the more complex the syntax will become, the higher the potential user’s acceptance level will rise.

Just like for all human languages, the acceptance criterion is usually overwhelming, as most people will use a less powerful DSL, or even create a new one, because they won’t come clear with a solution being too complex, or with a too steep learning curve, although more powerful.

To sump, some language features can be tested when evaluating a language as candidate for an EDSL:

- ✓ Functional Programming Features
  - Closures
  - High-Order Functions
- ✓ Closure definition
  - Support for real lambda functions
  - Syntax overhead of lambda definitions
- ✓ High-Order Functions
  - Support for currying
  - Syntax overhead for function passing if no currying is available
- ✓ Closure execution
  - Compiled language: Explicit typing and context enforcement at compile time
  - Dynamic language: Runtime selectable execution context

This thesis’ applications mostly focuses on hardware software design flows (Electronic Design Automation, EDA), for which the most widely spread language is TCL. In the next chapter, we are going to show how we setup a set of programming design rules to be able to create EDSL run in the TCL interpreter, with a very satisfactory semantic quality result.

3 Embedded Domain Specific Language design in TCL

So far, we mostly focused on presenting some programming principles and technics revolving around domain specific language design. An emphasis was brought on EDSL programming's advantages and drawbacks, with a concrete example using the Scala language.

However, hardware software co-design environment are in their vast majority supported by the TCL programming language, and the heterogeneity of the design flows, as outlined in the introduction, makes the development of domain specific languages attractive. Therefore, we tried to use the features of the TCL language to make the creation of EDSLs possible.

TCL being a quite minimalistic language, it does not natively support some required functional programming aspects like closures. In this chapter, we will thus introduce some important features of the TCL interpreter, and use them to enable closure programming in TCL scripts, and move-on to presenting an EDSL development methodology using object-oriented (OO) libraries. Figure 3-1 illustrates this development path to be followed.

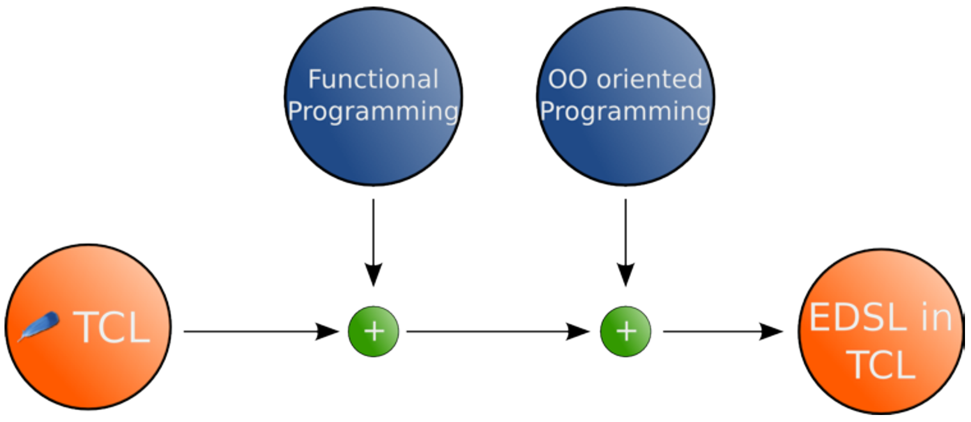


Figure 3-1 TCL to TCL-based EDSL development path

3.1 The TCL programming language

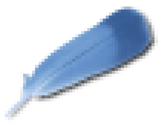


Figure 3-2 TCL logo

First presented during the 1990 Winter USENIX conference by John K. Ousterhout, the Tool Command Language (TCL [21] [42]) was created with simplicity and flexibility in mind, to allow tool writers to offer users a programming interface for customisation, without having to develop a new language for each new application. For this reason, aside from the language syntax definition, the TCL interpreter was made very easy to integrate in any software. Example 3-1 illustrates how to run a TCL script inside a traditional C application (using the latest version, but this procedure is still valid after 24 years).

```
#include <tcl.h>

int main () {

  Tcl_Interp * interpreter = Tcl_CreateInterp();

  Tcl_Eval(interpreter,"puts \"Hello World!\"");

  return 0;
}

$ gcc EmbedExample.c -o EmbedExample -I /usr/include/tcl8.6 -ltcl8.6
$ ./EmbedExample
Hello World!
$ ./EmbedExample
```

Example 3-1 TCL Interpreter embedding

Creating C based commands callable from TCL is also easy, as can be seen in Example 3-2. This is probably the reason why it has been widely adopted by the Electronic Design Automation (EDA) industry, which often relies on a programming interface (partly or totally) for the user interface.

```c
#include <tcl.h>

// Command implementation
int test_cmd(ClientData c,Tcl_Interp *i, int argc, const char *a[]) {

    printf("Hello World!\n");

    return 0;
}

int main () {

  Tcl_Interp * interpreter = Tcl_CreateInterp();

  // Linking of "test_cmd" function, under "hello_command" name
  Tcl_CreateCommand(interpreter, "hello_command", test_cmd, 0,NULL);

  Tcl_Eval(interpreter,"hello_command");

  return 0;
}
```

```
$ gcc LinkExample.c -o LinkExample -I /usr/include/tcl8.6 -ltcl8.6
$ ./LinkExample
Hello World!
$
```

Example 3-2 Command linking

Inspired by LISP languages (LISP stands for LISt Processing), one notable concept of TCL, which sometime makes it difficult to get along with during first encounters, is the reduced number of programming concepts that drive the syntax:

- ✓ List definition:
  - o Whitespace separated arguments and content placed between ***{ }*** are lists (Example 3-3 line 1)
- ✓ Command replacement (or function call)
  - o Content placed between ***[ ]*** is replaced by the result of the command specified by the first word (Example 3-3 line 5)
- ✓ Variable definitions:
  - o Variables' values can be retrieved using ***$*** (Example 3-3 line 5)
  - o Variables' names (without ***$***) are used as references (Example 3-3 line 3)

```
% set a {b c d}
b c d
% lappend a e
b c d e
% puts "List a has [llength $a] elements"
List a has 4 elements
```

Example 3-3 Basic syntax example (executed in tclsh interpreter)

To parse a script, the TCL interpreter thus simply reads the input, the first word on the first line being the name of a command, gathers the forth coming input as a list, which is then passed to the command implementation, if found in the command table. Although inspired from LISP dialects, it did not preserve the *polish prefix notation*, which makes the syntax still easy to read and avoids the readability issue mentioned in 2.1.3.

The remaining traditional imperative programming concepts, like function definition (called ***proc***) and control structures are provided by command implementations which can be called from TCL code. As illustrated in Example 3-4, this leads to a great flexibility in the command mapping (see 3.1.1 for details).

```
namespace eval a {

    ## An if needs a condition and a body to be executed
    proc if {condition body} {
        puts "Entered an if, with condition $condition"
        puts "Should we execute body: /$body/ ?"
    }

    puts "An if may not be an if"
    if {1==2} { puts "This is not reachable" }
}
Output:
#An if may not be an if
#Entered an if, with condition 1==2
#Should we execute body: / puts "This is not reachable" / ?
```

Example 3-4 Command overwritting in TCL

Like all modern scripting languages, TCL provides out of the box facilities for modular programming, introspection and self evaluation. Some of those features which are relevant for a good understanding of this thesis are detailed hereafter.

3.1.1 Namespaces and packages

Namespaces

The base concept and syntax for Namespaces (NS) resemble the one from C++. They allow to group together a set of commands and variables under a common name. A glimpse has already been given in Example 3-4, where a namespace called *a* was used to define a new *if* procedure, which was then formally registered in the interpreter under the name: *::a::if*. Calling a command or variable defined under an NS follows a simple set of rules, which are important to understand:

- ✓ Top NS: When some code runs under no specific namespace, it runs under the context called *top*, whose namespace name is *::*

```tcl
puts "Current top namespace: [namespace current]"
# Output: Current top namespace: ::
```

- ✓ NS-Call: If the call contains the *::* characters, the user is referencing a namespace.
- ✓ Absolute NS-Call: When performing an NS-Call starting with the *::* characters, the user has to pass the full NS name.

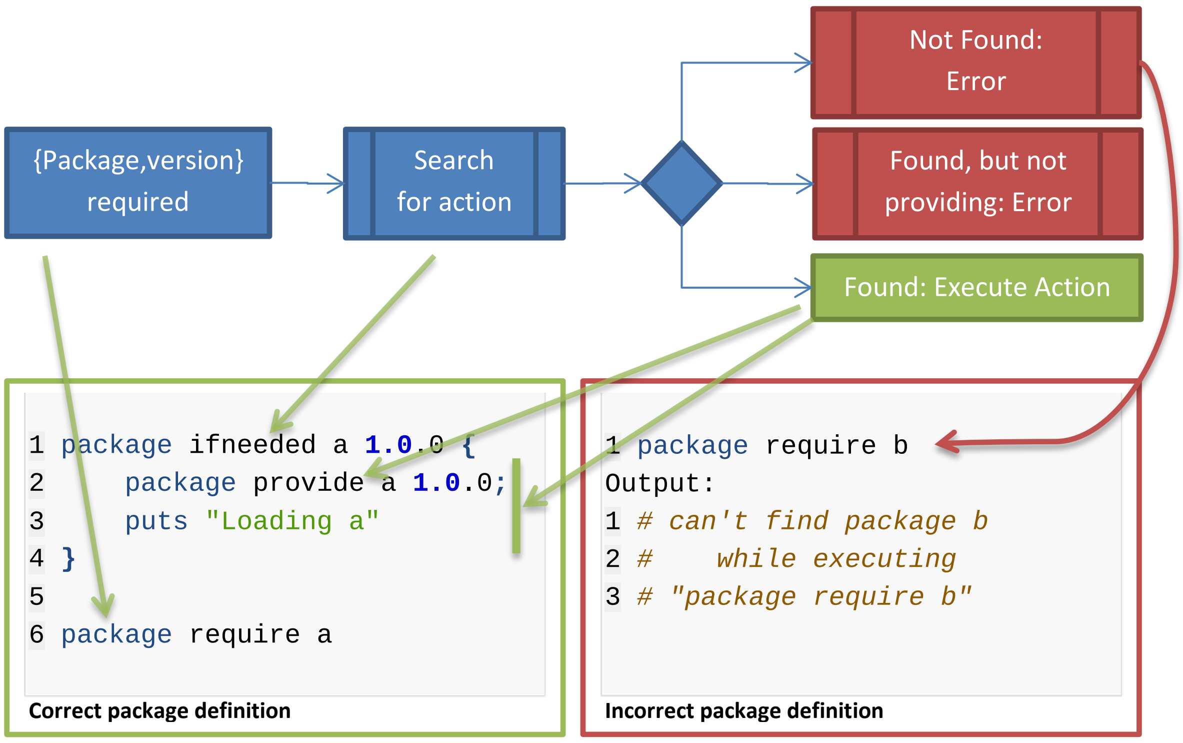
```tcl
namespace eval a {
    proc hello args {
        return "Hello World!"
    }
}
puts "Absolute call to hello in a: [::a::hello]"
```

- ✓ Relative NS-Call: During an NS-Call, if the name does not start with the *::* characters, the namespace search starts from the current namespace.

```tcl
namespace eval b {
    namespace eval c {
        proc hello args {
            return "Hello World!"
        }
    }
    puts "Relative call of hello in c in b: [c::hello]"
}
puts "Absolute call of hello in c in b: [::b::c::hello]"
```

Packages

Most modern Dynamic languages offer a mechanism to automatically load code libraries/ modules. Packages in TCL allow a user, on the one hand, to require the interpreter to find the sources for a set of functionalities, and a developer, on the other hand, to group them under a set of source file, and define a way for them to be loaded upon request. The package runtime environment does not actually require any formal packaging of source files, but it relies on finding an action (i.e. some code to execute) to take for a given *{package,version}* tuple, when a user requires it. The mentioned action will nearly always consist in loading one or more source files, but could be anything else, and must declare providing the *{package,version}* tuple, as illustrated in Figure 3-3.


{Package,version} required
Search for action
Not Found: Error
Found, but not providing: Error
Found: Execute Action
Correct package definition
Incorrect package definition

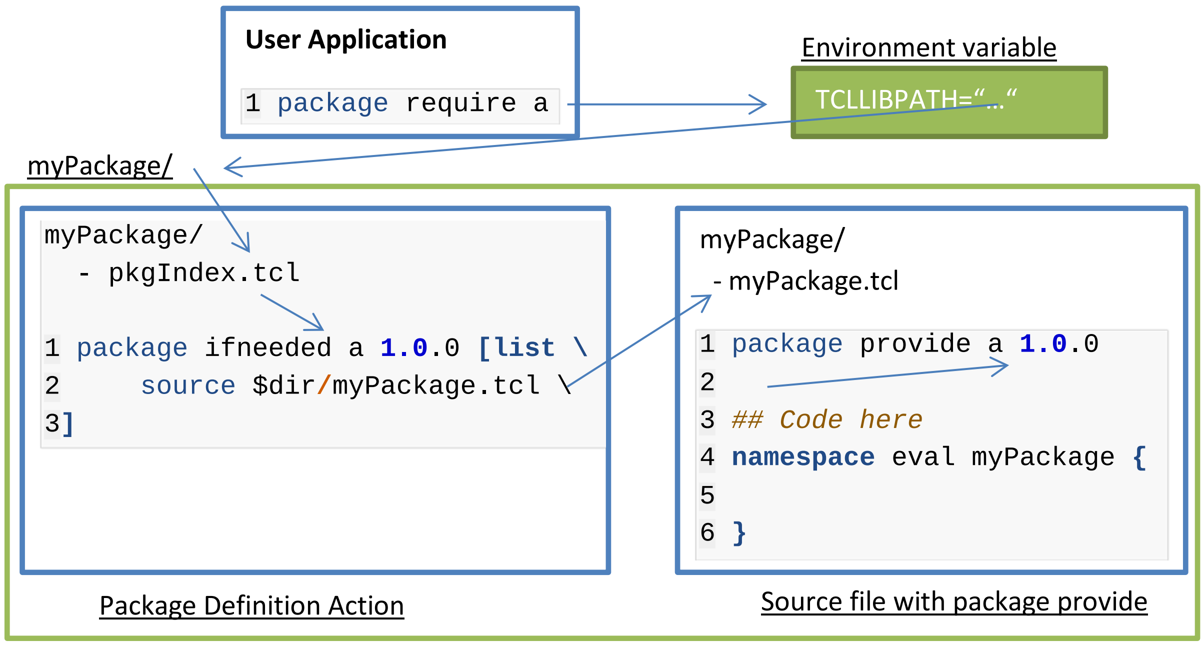
```
package ifneeded a 1.0.0 {
    package provide a 1.0.0;
    puts "Loading a"
}

package require a
```

```
package require b
Output:
# can't find package b
#    while executing
# "package require b"
```

Figure 3-3 Package search procedure

However, the package definition (i.e library) and requirement (i.e. user application) are always in separate locations, so the TCL interpreter provides some standard methodologies to find package definition actions (the ifneeded calls). The most widely used methodology relies on special files containing the package definition actions, which are called *pkgIndex.tcl*. Folders containing such pkgIndex.tcl files are located using environment variables like *TCLLIBPATH* (others exist). A Concrete package loading flow is described in Figure 3-4, where the *myPackage* package's source file is saved in a folder, with a *pkgIndex.tcl* alongside that can be auto-detected, and contains the action to perform to load the source code, in this case a simple source command call.

Note that the *myPackage.tcl* source file starts with a namespace definition. This is a very common strategy to ensure no overlapping can happen between sources loaded from other packages, and it is applied to all the modules that are presented in this thesis.



```
package require a
```

```
myPackage/
  - pkgIndex.tcl

package ifneeded a 1.0.0 [list \
    source $dir/myPackage.tcl \
]
```

```
myPackage/
 - myPackage.tcl

package provide a 1.0.0

## Code here
namespace eval myPackage {

}
```

Figure 3-4 Standard TCL Module loading using pkgIndex.tcl

3.1.2 Self evaluation

Dynamic languages commonly provide a function to evaluate some code contained in a string. It is not used very often, and also not recommended to avoid attacks like code injection, but can be very useful to run some code that is created at runtime, or fetch from an input stream. In TCL, the evaluation command is called eval, as can be seen in Example 3-5.

```
set a "Hello"
eval {
    puts "$a World!"
}
```

Output:

```
Hello World!
```

Example 3-5 TCL eval command for self evaluation

A very concrete usage of eval will be shown in the section 3.2.

3.1.3 Stack frame and execution level

Last but not least, it is very easy in TCL to follow the current execution level, called current frame. There are a few commands that enable inspecting the current stack frame, linking to variables in upper stack frames, or evaluating code in upper ones. Figure 3-5 provides a brief overview of the three main Stack Frame interactions and their matching TCL commands.

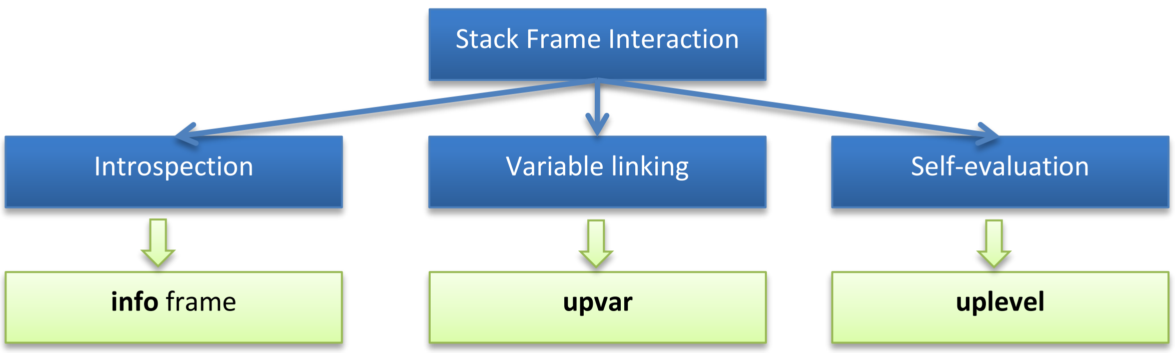


Figure 3-5 Stack frame interactions overview

Introspection

info frame ?number?

The ***info*** command provides information about the TCL interpreter state, including about the current stack frame. The user can, for example, retrieve the currently called method or the current file and line that are being interpreted. If no argument is provided, it simply returns the actual stack frame level, with 1 being the top level.

```
proc myCommand args {
    puts "My Command frame level: [info frame]"
}

puts "Top frame level: [info frame]"
# Top frame level: 1

myCommand
# My Command frame level: 2
```

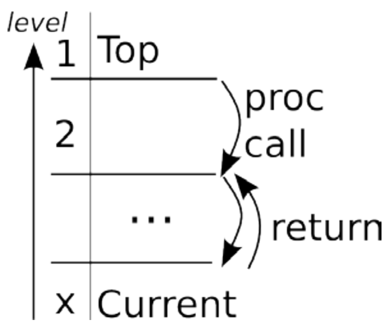
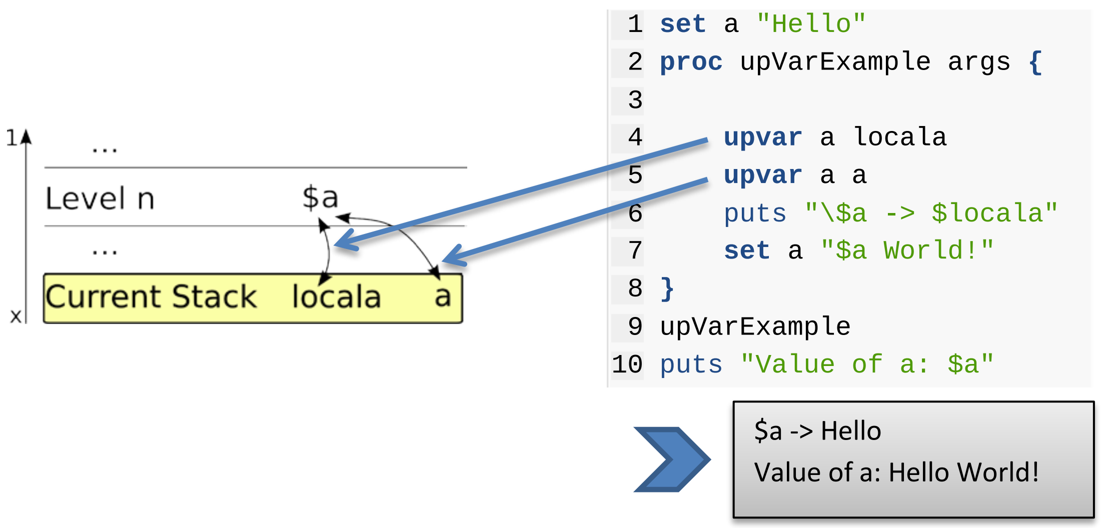


Figure 3-6 Stack frame level information in TCL

Variable linking

upvar ?level? upvar ?myVar?

Referencing variables that are declared in an upper stack frame level is possible through the use of the ***upvar*** command. Once a variable is linked in the current stack frame under a specific name, the binding is bidirectional, meaning that changes to the variable's value will be mirrored in the upper level when execution resumes. Figure 3-7 illustrates the usage of ***upvar***. The code example only links one level up, and shows that linking with the same name in both frames is allowed.



```
set a "Hello"
proc upVarExample args {

    upvar a locala
    upvar a a
    puts "\$a -> $locala"
    set a "$a World!"
}
upVarExample
puts "Value of a: $a"
```

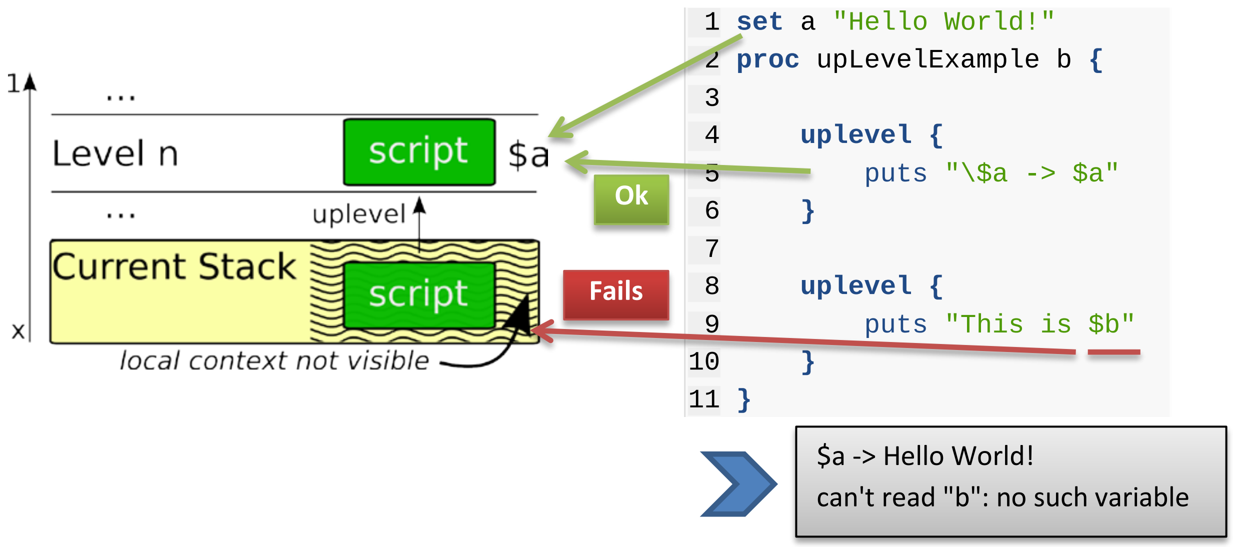
```
$a -> Hello
Value of a: Hello World!
```

Figure 3-7 Upper level variable linking

Self-Evaluation

upvar ?level? upvar ?myVar?

By combining standard self-evaluation, which we already presented, and the concept of ***upvar***, we can call for some code to be evaluated in an upper stack frame. In this case, the user must be aware that the current stack frame context won't be accessible anymore, so it is not possible to link down to variables or arguments. In Figure 3-8 two code examples are given, one of which illustrates an invalid usage of ***uplevel*** caller-level context access.



```
set a "Hello World!"
proc upLevelExample b {

    uplevel {
        puts "\$a -> $a"
    }

    uplevel {
        puts "This is $b"
    }
}
```

```
$a -> Hello World!
can't read "b": no such variable
```

Figure 3-8 Code execution in an upper level

Conclusion

Stack frame manipulation is a design pattern which is difficult and dangerous to use because it generates implicit changes of context, and the risk is high to lose sight of code runtime behaviour. Sometimes, though, it can be very useful, especially if its usage stays well encapsulated and hidden from the end user, when creating a library, for example. In the next section about closure implementation in TCL, we will show how the usage of Stack frame introspection, variable linking and self-evaluation allowed us to improve the TCL language by mimicking the behaviour of Functional Programming closures and high-order functions.

3.1.4 Pitfalls

So far we discussed some specific interesting features of TCL, however experience shows that the simplicity of TCL leads to a few issues which don't usually seem evident to new comers. Most programming language have an interpreter or compiler which performs a lot of syntax checks, and don't really allow ambiguities or valid syntax constructs which don't lead to the result a programmer would intuitively expect.

Lists are Lists

In TCL, all content placed between ***{ … }*** creates a new static list, meaning that the pure content is gathered, without any variable resolution, or even comment lines escaping. Example 3-6 illustrates this by creating two lists whose content may not seem very intuitive:

```
set a {b c $d e}
# b c $d e

set a {
    b
    # c
    d
    #e
}
# b
# #
# c
# d
# #e
```

Example 3-6 Ambiguous Static list example

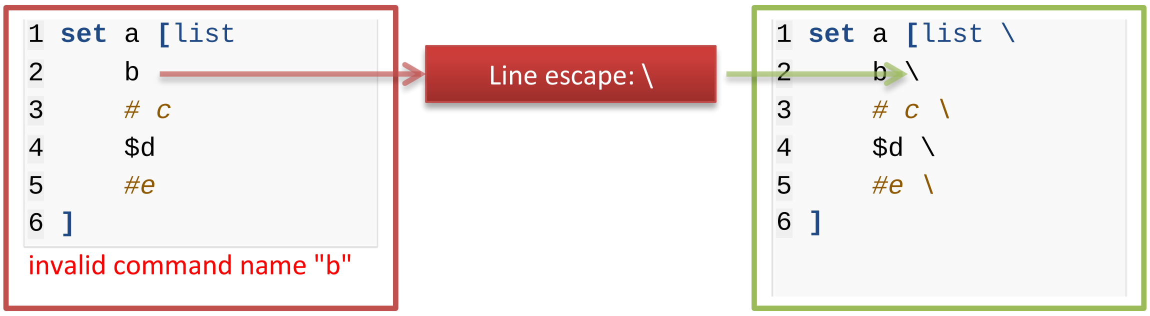
- ✓ The first list contains a variable call ($d), but it is saved as a simple string in the list. Getting the value of $d would require creating the list using the ***[list …]*** command

```
set d "hello"
set a [list b c $d e]
# b c hello e
```

- ✓ The second one contains two comment lines, which are added as content to the list. The line "# c" even generates two entries because the interpreter uses any whitespace character (space, tabulation and return to line) as separator.

Commands don't span over multiple lines

Unlike static list building, command arguments cannot be spanned over multiple lines, otherwise the arguments would be interpreted as new commands to execute. To make this easy to understand, we can just build the second list of Example 3-6 using the ***[list …]*** command, as shown in Example 3-7 and show how new lines have to be escaped to avoid being formally interpreted.



```
set a [list
      b
      # c
      $d
      #e
]
```

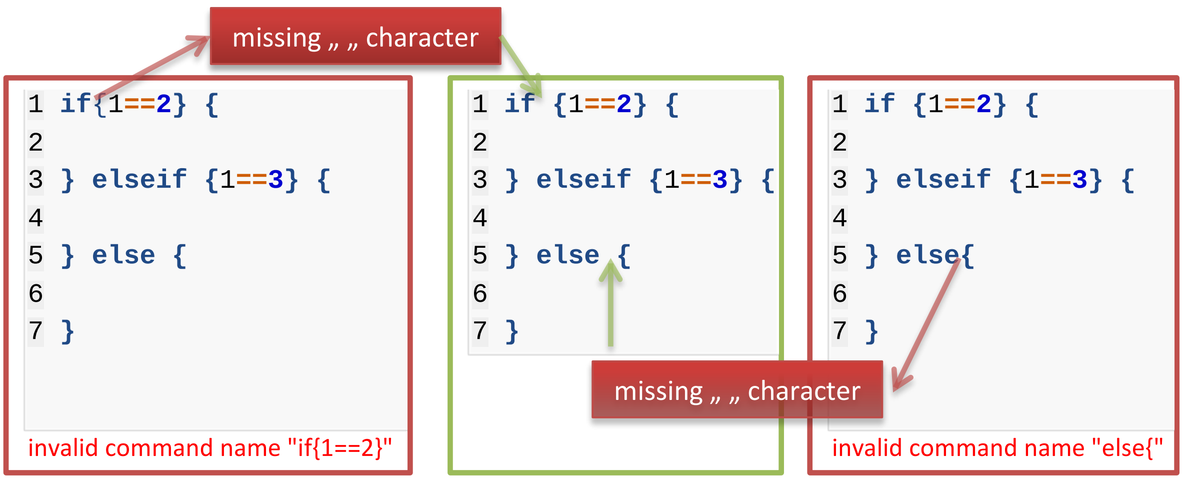
invalid command name "b"

```
set a [list \
      b \
      # c \
      $d \
      #e \
]
```

Example 3-7 Command span over multiple lines requires line escaping

Everything is a list

TCL performs very little syntax checks, as all constructs fall back to commands calls, including control structures. Knowing that the basic list's elements separator is a whitespace character, those become very important for valid syntax. Omitting one may lead to a wrong number of arguments in a command call, or even make the interpreter try to call an inexistent one. Such an extremely common error happens when writing if .. elseif .. else … construct, as illustrated in Example 3-8.



```
if{1==2} {

} elseif {1==3} {

} else {

}
```

invalid command name "if{1==2}"

```
if {1==2} {

} elseif {1==3} {

} else {

}
```

```
if {1==2} {

} elseif {1==3} {

} else{

}
```

invalid command name "else{"

Example 3-8 If-else common typo pitfall

3.2 Implementation of Closures in TCL

In section 2.4, we presented the idea of using the high-order functions paradigm from functional programming to embed a programming interface (API) in a language. This API could then be adapted to implement a semantic mirroring a standard domain specific language. Most specifically, some examples have been presented using the Scala programming language, which already includes the required features.

Since TCL is not a functional language, there is no direct support for closures. However, looking back at Definition 2.1, we already know three aspects are needed:

- ✓ Capturing the function definition (i.e. some code), and call it whenever required.
- ✓ Resolve the free variable references present in the context of the closure definition (the location in source code where it has been defined).
- ✓ Protect local variables from possible name clashes with environment variables

Moreover, given that closures are usually passed to high-order functions (Definition 2.2), they are in this case passed down in the call stack to some utility functions which will make use of them (refer to list example in 2.2.2), we can define a closure subset called "stack-down only", where free variable references would thus always be up-stack, or local.

This limitation is critical, because allowing a closure to be reused in the parent context of its definition location would require making the references to the local variables safe against context clean-up that happens after a return from a function call. As presented in Figure 3-9 on the left-hand side, this would be problematic, and require a garbage collector responsible for clean-up when the closure itself is not needed anymore. Such a mechanism is not available in TCL and is not really needed for our usage.

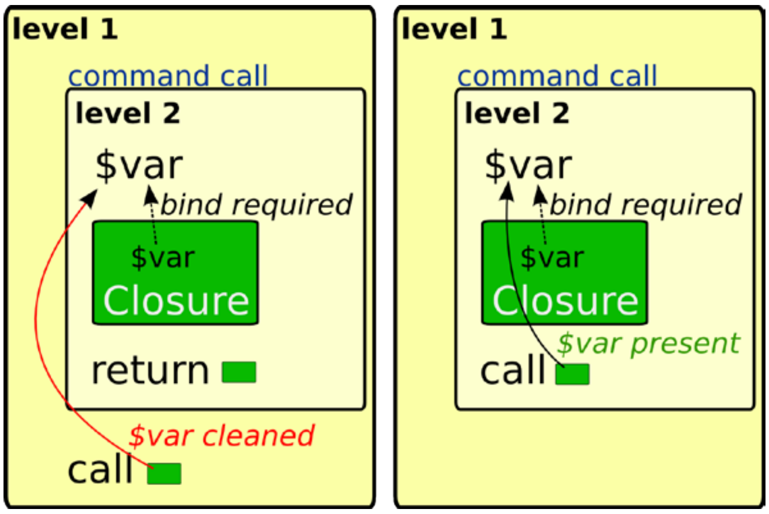


Figure 3-9 Stack-down closure subset

The right-hand side of Figure 3-9 shows a valid use case, where the closure is run in a frame level greater than or equal to the target variable to be bound. The requirements for a closure implementation can be refined to:

- ✓ Capture and call some code (i.e. an anonymous function definition) whenever required
- ✓ Resolve variables up-stack only.

Looking back at section 3.1, we notice that those two features have already been presented (we recommend going through 3.1.2 and 3.1.3 before reading further). We are going to present here how we brought them together to offer the user a closure programming interface.

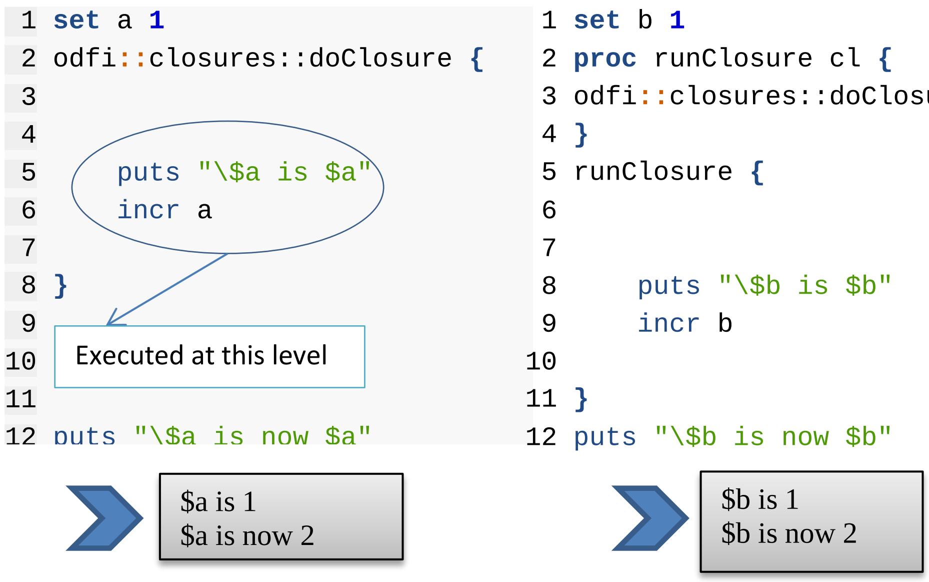
Over time, we came to two different approaches, the second one stemming from some limitations of the first try which where difficult to solve.

3.2.1 First implementation (v1 and v2)

Repository: odfi-dev-tcl, Path: tcl/closures-2.1.0.tm

In order to mirror the closure examples presented so far in other languages, we will start by analysing two very simple closure executions in TCL. The goal here is to make the run level clear, which is why both outputs are identical, but under different run contexts:

- ✓ In Example 3-9 left, a closure is run, but behaves as if the content was run without the ***doClosure*** call, because the ***doClosure*** command always evaluates the code with an up-level of at least 1, to run in the caller context (the one the user means).
- ✓ In Example 3-9 right, the closure is run inside a command, so that one execution level separates the definition of variable ***$b*** its usage. The output shows that is has been properly "closed", as the value updated in the closure can be seen in top level.



```
set a 1
odfi::closures::doClosure {


    puts "\$a is $a"
    incr a

}



puts "\$a is now $a"
```

```
$a is 1
$a is now 2
```

```
set b 1
proc runClosure cl {
odfi::closures::doClosure $cl
}
runClosure {


    puts "\$b is $b"
    incr b


}
puts "\$b is now $b"
```

```
$b is 1
$b is now 2
```

Example 3-9 left: No effect closure ; right: Closure with one bound variable

3.2.1.1 Implementation

The basic idea behind the first version implementation presented in Figure 3-10, is to prepare the closure by looking up the variable references using a regular expression (syntax: ***$variable***) in the code, then for each found variable, look-up a possible existing reference in any up-level context, and if found, prepend the binding command (***upvar***) to the closure definition before evaluation.

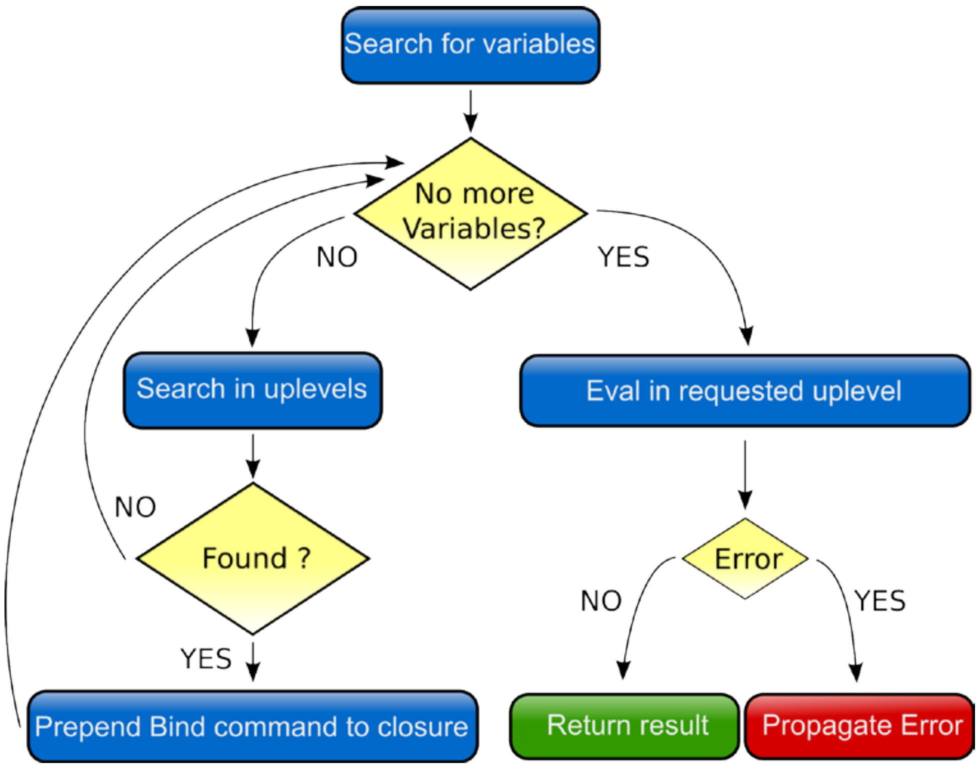
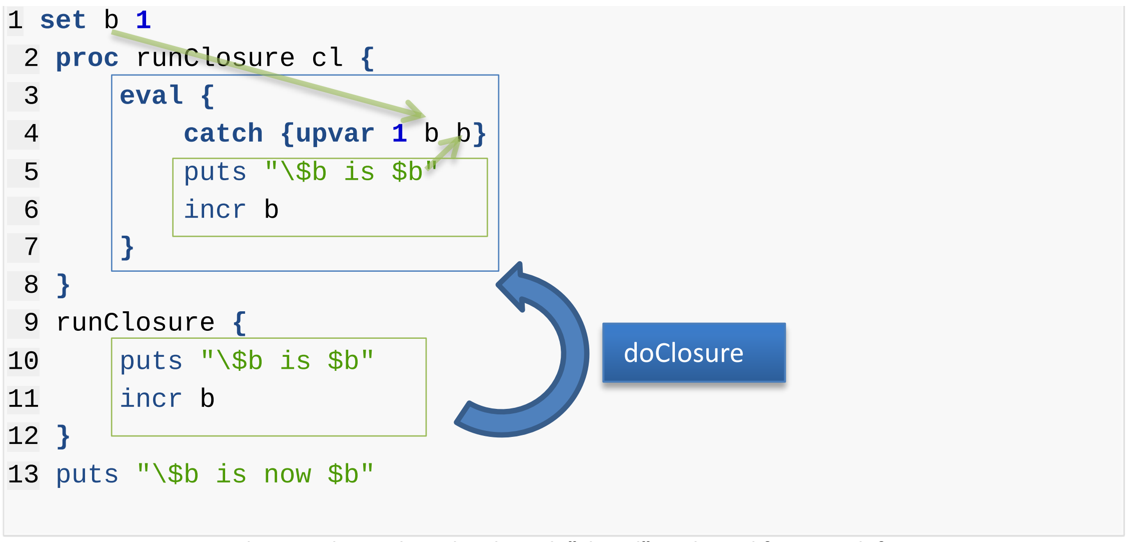


Figure 3-10 Function definition "closing" flow diagram

Applying the algorithm to Example 3-9 (right), we can have the library print the actual function definition that is going to be executed. In Example 3-10, the same code is presented, but the ***doClosure*** call is replaced by the final fully closed code which is evaluated.



```
set b 1
proc runClosure cl {
    eval {
        catch {upvar 1 b b}
        puts "\$b is $b"
        incr b
    }
}
runClosure {
    puts "\$b is $b"
    incr b
}
puts "\$b is now $b"
```

Example 3-10 Closure based code with "closed" evaluated function definition

To go into details, the ***doClosure*** command processes the function definition passed to ***runClosure*** as follows:

1. Variable search result: ***{ b }***
2. Search for variable ***b*** in levels: 2 *(call location in **runClosure**)* to 1 *(top)*
    a. Found in level 1
    b. Prepare ***upvar*** call with level reference: 1 (difference between found level and call level, $2 - 1 = 1$ in our case)
3. Evaluate the prepared ***upvar*** call and the passed closure using ***uplevel 1*** (equivalent to simple ***eval*** in ***runClosure*** as shown in line 3).

3.2.1.2 Variable detection issue

Use cases presented so far work for most cases, but an issue appears when searching for the free variables that should be closed upon. We mentioned in section 3.1 that variable values were to be referenced using the ***$*** character, whereas variable updates are performed by simply passing the name to a command. It is illustrated in Example 3-11, where our closure only changes the value of ***b***. In this case, it won't be bound to the top-level b variable if only value references are searched.

```
set b 1
proc runClosure cl {
    odfi::closures::doClosure $cl
}
runClosure {

    set b 4

}
puts "\$b is now $b"
```

```
$b is now 1
```

Example 3-11 Closure variable detection issue

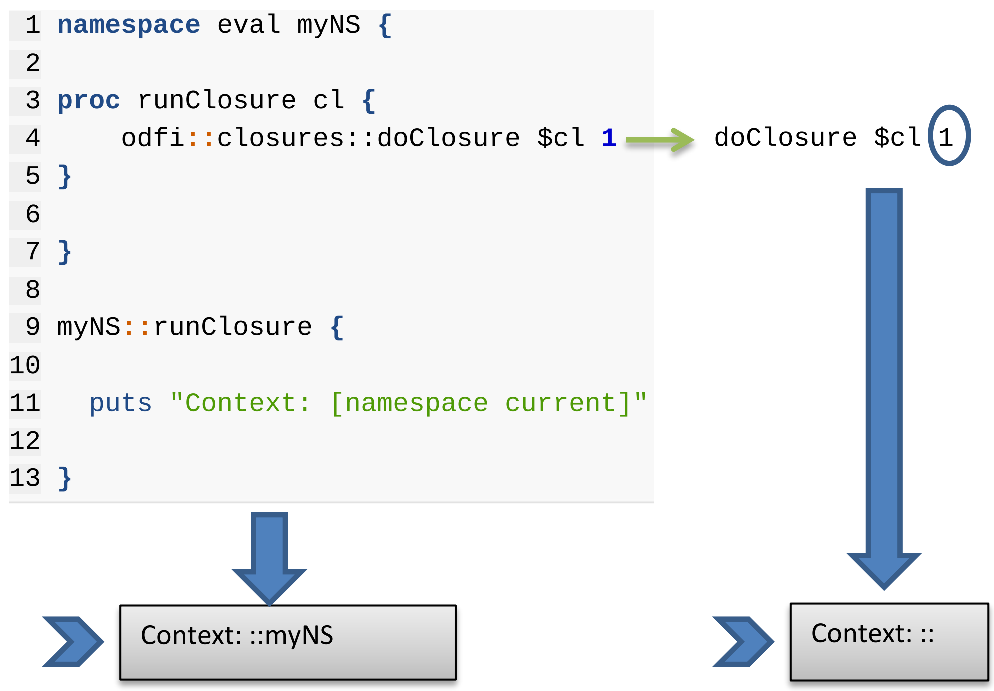
This precise case is not very common, because most closure definitions call for the values of variables to be closed, which will then be detected (because of the ***$***). But sometimes they don't, and the code will still run error-free, just not producing the expected output. To fix this issue, the variable search has to look for multiple usage patterns:

- ✓ Value call: ***$variableName*** format
- ✓ References passed to commands
    - o Needs to be statically implemented because we can't identify command string arguments that are variables references (like `set b 4` in the example).

In the current implementation, the variable search has been enriched to solve this issued if a variable is incremented using the ***incr*** command, which is a use case that can be encountered in loops.

3.2.1.3 Run level selection

Finally, it is important to be able to select the execution level of the closure. Indeed, if we modify our base example to introduce namespaces (so far everything ran in the top level, which is an "easy-life" corner case), we can see that our closure doesn't run in the top level context anymore, which is the programmer's wish, but in the ***runClosure*** procedure namespace. That is why we introduced a way to have the closure be executed in a level chosen by the user. In Example 3-12, the call to ***doClosure*** is updated on the right-hand side by adding a 1, meaning that the closure should be run one level up from the call location.



```
namespace eval myNS {

proc runClosure cl {
    odfi::closures::doClosure $cl 1
}

}

myNS::runClosure {

  puts "Context: [namespace current]"

}
```

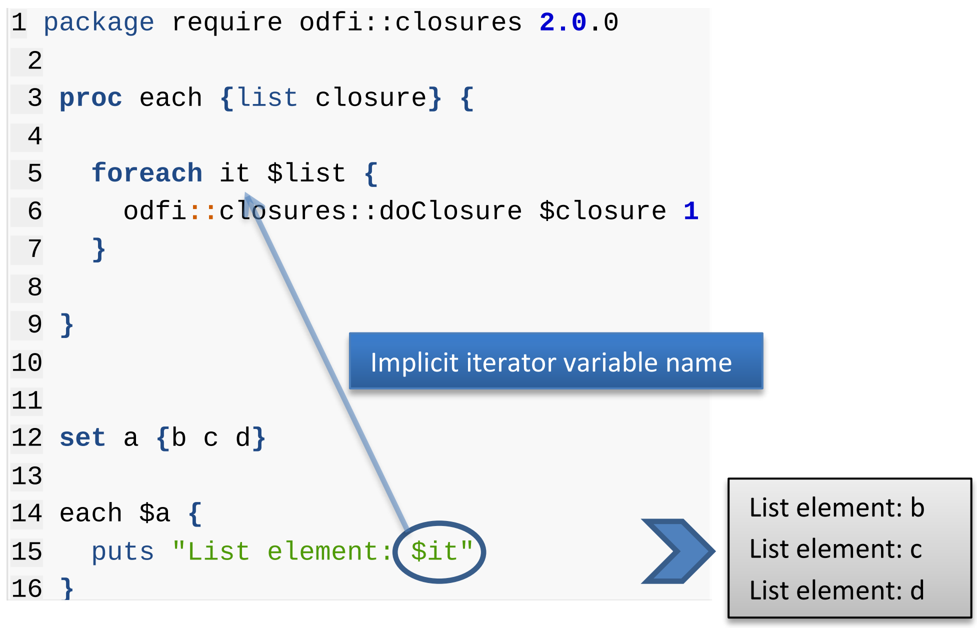
```
Context: ::myNS
```

```
Context: ::
```

Example 3-12 Closure run level selection

3.2.1.4 *Variable protection and implicit naming*

So far we have only worked with closures being lambda function without input arguments. Some language constructs may however pass arguments to a closure, like list iterators (see 2.2.2). This issue is automatically handled by the fact that the function definition is only closed in the context where it is going to be executed. A function may then set a variable value before running the closure, which may in turn use this variable, given that a specification documents the chosen name. This is illustrated by Example 3-13 which shows the implementation of a simple list iterator.

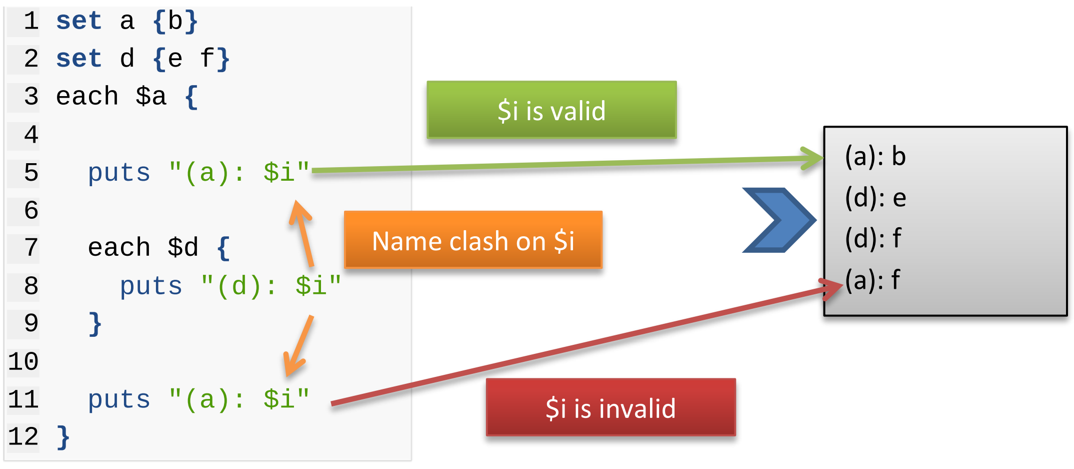


```
package require odfi::closures 2.0.0

proc each {list closure} {

  foreach it $list {
    odfi::closures::doClosure $closure 1
  }

}


set a {b c d}

each $a {
  puts "List element: $it"
}
```

```
List element: b
List element: c
List element: d
```

Example 3-13 List iterator "each" construction in TCL

This usage is valid as is, but it does not cover the case of name clashing. Indeed, the variable called ***it*** can be considered being an input parameter of the λ-expression used as closure. The counter-example presented in Example 3-14 ***($it*** has been changed to ***$i*** in the implementation of the ***each*** procedure) is simply the case where the user would call the ***each*** procedure inside the closure passed to a first each call. A name clash happens in a quite subtle way, as the dynamic execution nature of the closure leads to a wrong value for the ***it*** variable in the enclosing ***each*** call…only after the second level ***each*** invocation.



```
set a {b}
set d {e f}
each $a {

  puts "(a): $i"

  each $d {
    puts "(d): $i"
  }

  puts "(a): $i"
}
```

```
(a): b
(d): e
(d): f
(a): f
```

Example 3-14 Name clash in closures without variable protection

A name clash protection for closure input parameters has to be implemented, but as we have seen, the implementation is quite lazy and has no mechanism to support input parameters definition. Protecting per hand in the implementation of the each procedure is an option, but at the time we stumbled upon this issue, too many libraries were already implementing this kind of constructs.

For the sake of simplicity, we decided to introduce the ***it*** variable as a "variable to protect" in the implementation of ***doClosure***. The protection mechanism does not require any innovation, and simply consists of a value stack, where the variable's value is saved before running the closure, and restored afterwards. The valid result of Example 3-14 is shown on the right, when ***$i*** is replaced by ***$it***, and thus gets protected by the implementation.

```
(a): b
(d): e
(d): f
(a): b
```

Example 3-15 Valid output with variable name protection

Variable value pull

It might not be trivial at first sight, but the previous example introduced an issue linked to the run level selection. If we analyse the $it variable location, we obtain:

- ✓ $it is located in the each level
- ✓ $it is used by the closure at $each\ level - 1$

Technically, it is not possible to bind ***$it*** to its definition location, as the closure is run higher than ***each*** stack level. The implementation must thus handle the case where free variables must be bound to a variable present "down-stack" by copying the value up before running. Although this might seem tricky, we can sense that it is not really, as this behaviour is actually mirroring what is happening in standard function calls, for which input arguments values are copied and passed to the function execution's context. We join here the critic emitted previously about the lack of formal λ-expression support.

3.2.1.5 Limitations

This first implementation of our closure algorithm has been used by most of the presented applications in this thesis. However, some issues which had been overseen appeared, and although usually not very critical, can be extremely unpleasant to debug, as they usually don't produce any errors, just an incorrect result. Those issues are following:

- ✓ Incomplete support for non local free variable detection (value calls using ***$*** are detected, updates by name, as with set are not well supported)
- ✓ Closure input variables are implicitly set by execution context. The user cannot specify a state-of-the-art λ-function with input arguments specification.
- ✓ Look-ahead detection makes generic implicit variables protection not possible (the iterators, for example, must be named ***$it***). The user cannot customize the variables to be protected as input arguments.
- ✓ Recursive closure calls will need to search and resolve multiple times the same variables to propagate bindings at each call level.

3.2.2 Second implementation (v3)

Repository: odfi-dev-tcl, Path: tcl/closures-3.0.0.tm

Simply fixing the issues from the first closure implementation by handling corner cases per hand would make the code complexity rise, and make ensuring stability difficult. Although a unit-testing strategy is used to make sure this low-level library doesn't break, we needed to think about a more meaningful way to proceed than the brute-force solution presented previously, especially considering that some formal closure definition issues appeared.

One more time, three main phases are executed when running a closure:

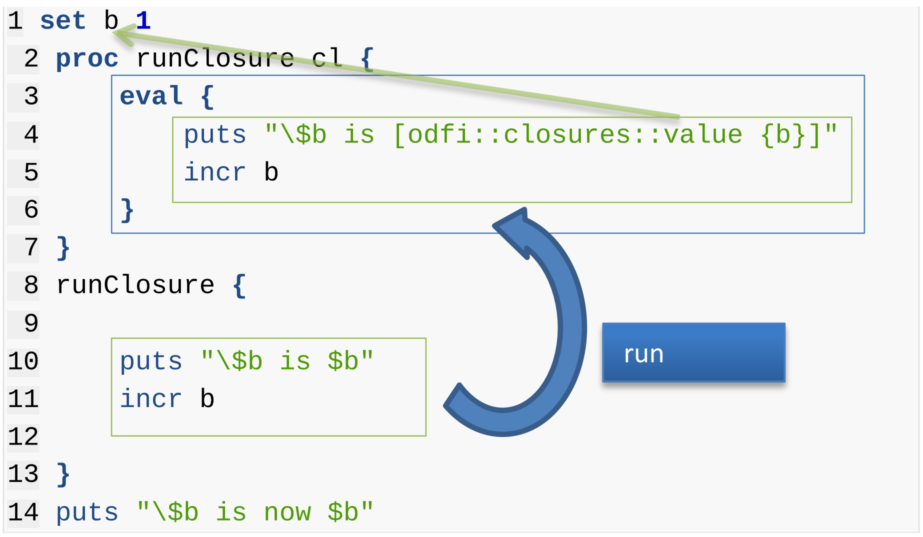
1. Find the free variables usages (value call using ***$*** or updates by name)
2. Protect and restore the local variables that are subject to name clash with the environment
3. Bind the variables to the environment when necessary, or pull them up.

The first point is not execution context dependent (the code is written once, and not changed dynamically), but the two last rely on the context. This is why we explored the idea of changing the phases order in the following way:

- ✓ First implementation:
  1. Detect variables
  2. Search and bind
  3. Run
- ✓ Second (new) implementation:
  1. Detect variables
     - ➔ Replace them with runtime value search procedure call
  2. Run
     - ➔ Variables calls are resolved and bound to environment

3.2.2.1 Variable value resolution

First, we are going to focus on running closures, ignoring the issues which are related to the definition of formal anonymous λ-expression. Delegating the variable binding to runtime is an easy task, like in the first implementation we just look for the variable value calls, but replace them by a procedure call which will try to resolve the variable. Example 3-16 shows the actual internally run code based on the user input. The old ***doClosure*** name has been replaced by ***run***, and we can see that ***$b*** is going to be bound and retrieved only when evaluating the call to ***odfi::closures::value***.

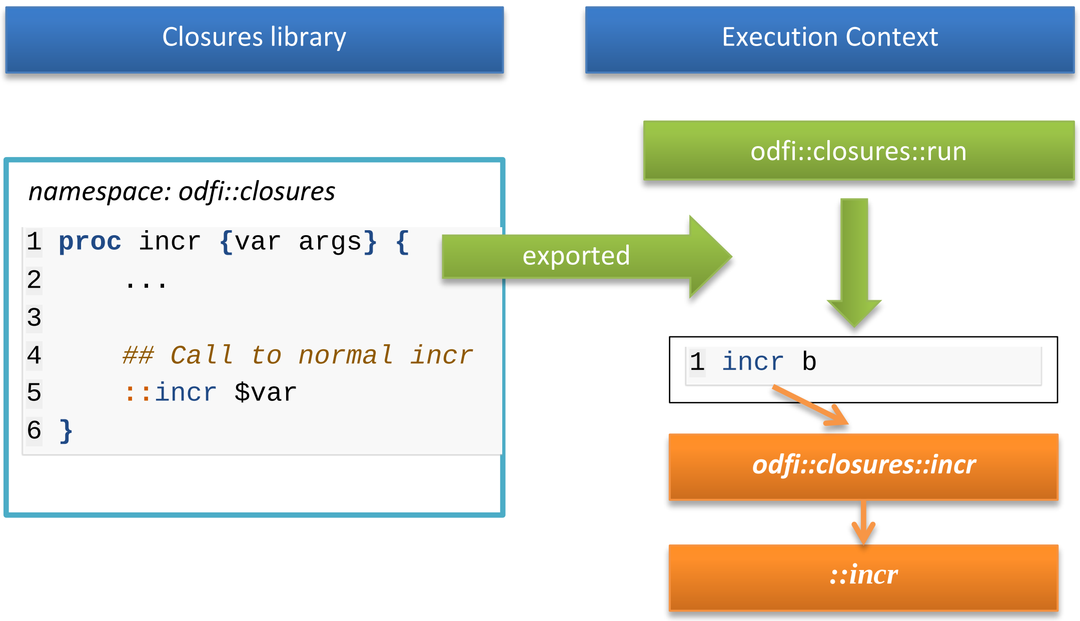


```
set b 1
proc runClosure cl {
    eval {
        puts "\$b is [odfi::closures::value {b}]"
        incr b
    }
}
runClosure {

    puts "\$b is $b"
    incr b

}
puts "\$b is now $b"
```

Example 3-16 Runtime variable binding in closure

3.2.2.2 Variable update resolution

Once again, we stumble on the problem of variable updates, which are performed by passing variable names without the ***$*** character, like the call to ***incr*** in previous example. The solution must be implemented using a runtime alternative as for the value resolution, as we have no look-ahead at all anymore. The solution has actually already been presented in section 3.1, Example 3-4. We can simply replace the implementation of the commands that should update variables, and have them perform resolution before calling on the standard implementation. The example for ***incr*** is illustrated in Figure 3-11, and this mechanism can be reproduced for all required commands.



```
proc incr {var args} {
    ...

    ## Call to normal incr
    ::incr $var
}
```

```
incr b
```

Figure 3-11 Runtime value updating procedure call model

There is no need to go deeper into details here, as the implementation mainly focuses on exploiting TCL interpreter features, making sure error cases are correctly handled to avoid leaving the closure execution context unclean etc… The reader can refer to the source code for further details.

3.2.2.3 Lambda support

While presenting our first implementation, we mentioned two issues being related to the lack of nice state-of-the-art support for λ definitions:

1. Variable protection
2. Variable value “pull-up” depending on the closure target run level

To solve those issues, we will first try to better formalise our closures as λ functions, and the solution should naturally come out.

Basically, we need to be able to provide a way to specify an anonymous function definition, which requires:

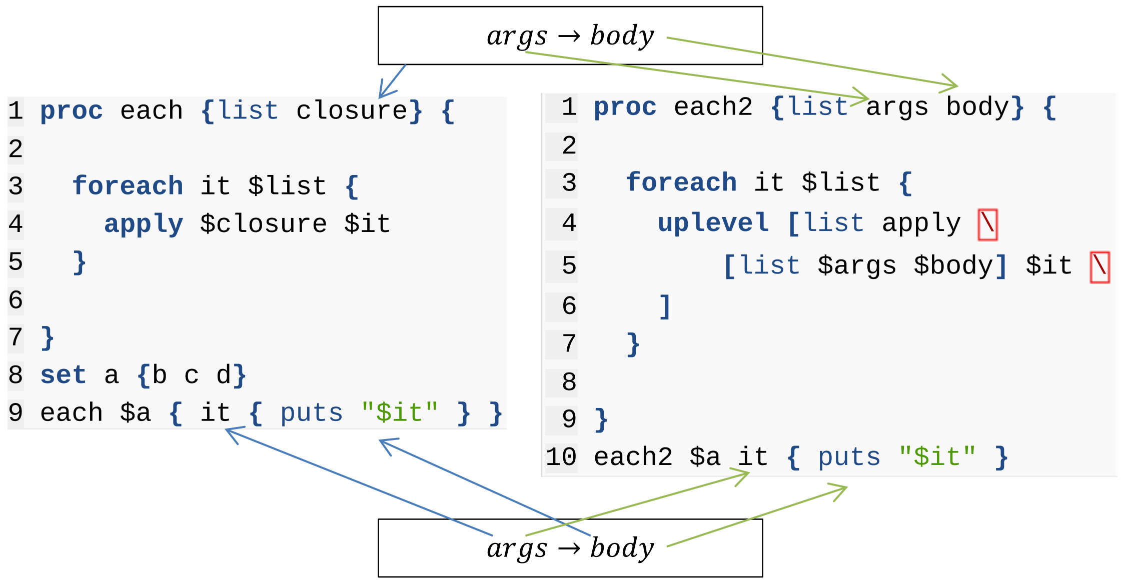
1. Input arguments specification
2. An expression, of function body

$$\boxed{args \rightarrow body}$$

Following a strictly formal design path, we could implement a solution that would look like a LISP λ definition, by creating a procedure in the TCL interpreter, based on input arguments and body, and return a reference so that it can be called. Examples can be found on the TCL wiki (example: http://wiki.tcl.tk/519), or using the TCL 8.5 ***apply*** command, but no precise literature reference could be found. The pure lambda creation and binding as a real procedure would not really bring any specific advantages, and introduce unattractive syntax. The ***apply*** function is a bit more interesting, but presents some important issues:

- ✓ No support for closures
- ✓ The syntax is not very natural
- ✓ Natively Supported from TCL 8.5, while most of this work has been constrained to TCL 8.4 because of third-party software limitations

As illustrated in Figure 3-12, the usage of apply is quite acceptable. The basic syntax requires gathering input arguments and the body in a list, which makes a list inside of a list (on the left side). This can be overcome by splitting the input arguments of the each function (on the right side), which produces a nicer looking result although arguments and body are separated from each other. The limitation on closure support could be overcome by passing the body through the variables replacement algorithm, and surrounding the call to ***apply*** by the necessary variable update procedures overrides (incr, set etc…).



```
proc each {list closure} {

  foreach it $list {
    apply $closure $it
  }

}
set a {b c d}
each $a { it { puts "$it" } }
```

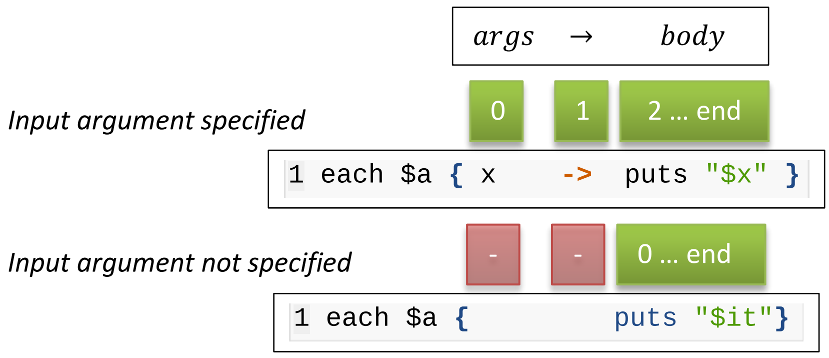
```
proc each2 {list args body} {

  foreach it $list {
    uplevel [list apply \
        [list $args $body] $it \
    ]
  }

}
each2 $a it { puts "$it" }
```

Figure 3-12 "each" implementation using TCL 8.5 apply

In this light, the benefits of ***apply*** become quite limited, and it lacks the flexibility of implicit variables. These, although presented in 3.2.1.4 as the consequence of an issue, are quite convenient for lambda functions which will always be called using the same format. For example, iterator functions, like our example's ***each***, are nearly always going to see the closure's input parameter named ***$it***. This feature also exists in the Groovy language, where the input arguments are assigned to an implicit variable name if not defined in the closure's arguments list.

3.2.2.3.1 Implementation

The syntax choice made for our implementation is very simple. One just has to look at the formal λ definition we have been using in this section, and note it is exactly the one used in the Scala programming language (see 2.2.2). Moreover, there is no parsing required to support it in TCL, as it takes the form of a list, which is the base input format of everything in this language. Figure 3-13 presents the two possibilities: closure definition with input arguments specification or without. The "end" keyword refers to the end of the input list.



```
each $a { x    ->  puts "$x" }
```

```
each $a {          puts "$it"}
```

Figure 3-13 List based Lambda definition

Based on the use cases from Figure 3-13, two format definitions have been set for both:

✓ λ-expression list definition:

```
{ {arg0 ... argn} ->  body }, single argument: { arg ->  body }
{ {arg0 ... argn} =>  body }, single argument: { arg =>  body }
{                     body } (Unspecified)
```

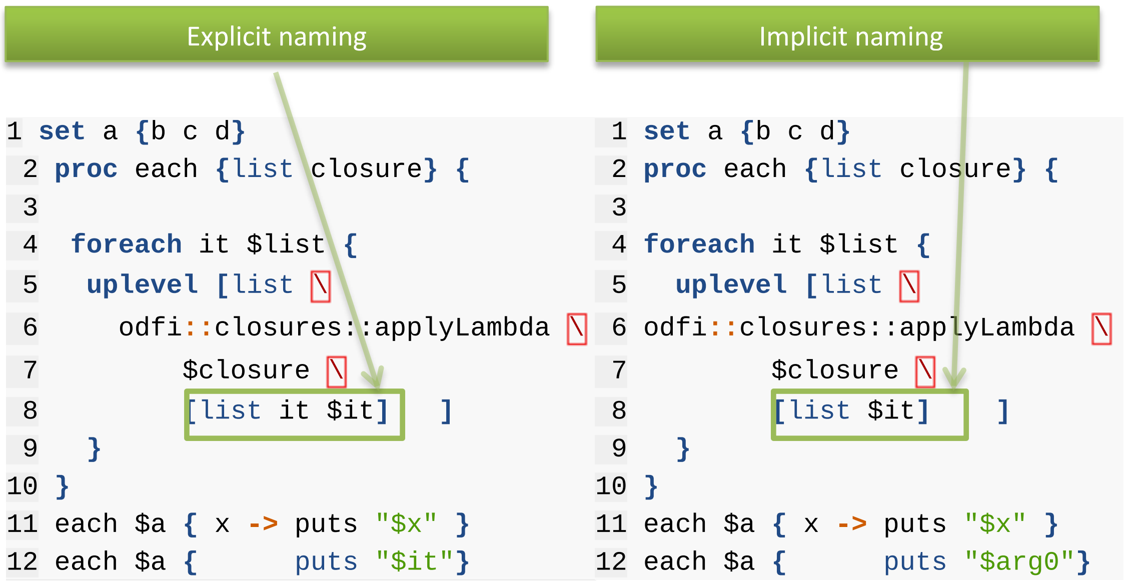
1. Optional:
   - ➔ A List of input arguments (no { } required if only one argument)
   - ➔ The symbol string "->" (Groovy Syntax) or "=>" (Scala Syntax)
2. Mandatory:
   - ➔ The expression's body, as remaining list content (not enclosed in a sub-list)

✓ λ-expression call:

```
odfi::closures::applyLambda lambda arguments?
```

1. *lambda*: A λ-expression according to previous definition
2. *arguments*: A list of values to bind to the λ-expression's input arguments. Each element of the list has the following format:
   ➔ 1-Element: the value
   or
   ➔ 2-Elements: a {*name value*} tuple. The *name* will be used as implicit variable name if the λ-expression's lacks input arguments specification.

Finally, Figure 3-14 presents the ***each*** example ported to this new format, in all use cases. It is to be noted that if an implicit definition is required, but the caller omits the name specification in the ***applyLambda*** arguments list, the runtime generates the name, with format "arg$_n$" depending on the argument's position.



```
set a {b c d}
proc each {list closure} {

 foreach it $list {
  uplevel [list \
    odfi::closures::applyLambda \
        $closure \
        [list it $it]    ]
   }
}
each $a { x -> puts "$x" }
each $a {      puts "$it"}
```

```
set a {b c d}
proc each {list closure} {

foreach it $list {
  uplevel [list \
odfi::closures::applyLambda \
        $closure \
        [list $it]    ]
  }
}
each $a { x -> puts "$x" }
each $a {      puts "$arg0"}
```

Figure 3-14 applyLambda usage examples, with and without implicit naming

It is interesting to note that the call scheme slightly changed from the previous implementation. Indeed, in version 2, the target execution level was selected when calling the ***odfi::closures::doClosure*** procedure (see 3.2.1.3). In this new implementation, the applyLambda call is fully defined with all required arguments, and can thus be executed in the target execution level at user's discretion using a simple ***up-level*** call.

3.2.3 Discussion

In this section, we have set the base for supporting λ-expressions and closures in TCL. Although a good quality implementation has been reached in version 3, most of this thesis' presented work is based on the version 2. The vast majority of encountered use cases were well covered by this first implementation, and most users would not notice or stumble on some of the limitations.

The feedback from experience and the fixes work done on this library will allow future developments to be more solid and less prone to implicit errors. Still, we explored two different implementation solutions, and the closure resolution logic has to be kept at the runtime level. As we will see later, this is not such an issue, because a lot of use cases involve running a closure only once.

However, in case of multiple usage of one single closure, as for list processing or special constructs like the presented ***::each*** example, the variable search and replace is run every time, but this can be easily optimised by separately preparing the closure before running it. Variable binding through ***upvar*** at each invocation is to be expected, as each closure invocation relies on a procedure call to ***uplevel***, causing the variables to be unbound at end-of-call context clean-up.

The next section will present the usage of closures to create Embedded Domain Specific Languages in TCL and give a concrete path to follow for future developments. All of the examples will be based on the latest closure support implementation (v3), as the applications presented in chapter 4 are using v2.

3.3 Embedded DSL in TCL

Now that we have implemented support for closures and high-order functions (Definition 2.2) in TCL, we are fulfilled some of the requirements to be able to create EDSLs. In regard to the checklist we set up in 2.4, it seems we are close to reaching the destination. What is missing in the default TCL interpreter is support for named data structures, like classes. Indeed, the examples presented in 2.4 relied on a simple systematic process to create hierarchies:

1. Create a data structure
2. Configure by passing a closure
3. Update the hierarchy

or

1. Create a data structure
2. Update the hierarchy
3. Configure by passing a closure

Fortunately, TCL is well equipped with libraries, and numerous object-oriented frameworks exist which we can use for this purpose. We are going to present the one we used the most: incrTCL [43] and a next generation one which is worth of interest, although more complex for non-specialists: the Next Scripting Framework [44].

3.3.1 Introduction with the incrTCL library

We want to focus here on applying the EDSL creation methodology to the TCL language. We chose the incrTCL framework for object-oriented programming (OOP) because it is very simple, presents a semantic very similar to the OOP features of well-known languages like C++/Java/Scala, and is even included in the standard TCL distribution since version 8.6. The presented development however is not limited to incrTCL, and can be easily ported to any other library of the user’s choice.

To avoid being too abstract, we are going to analyse a concrete use case. Our goal will be to create a simple image using the Scalable Vector Graphic (SVG) image format. An SVG file is an XML document, which contains graphic object definitions (position, style etc…), as presented in Figure 3-15.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"  version="1.1"
    width="40" height="40">

    <rect fill="orange" opacity="0.4"
       width="40" height="40"
       x="0" y="0" />

    <circle fill="green" opacity="0.4"
       r="10" cx="20" cy="20" />
</svg>
```

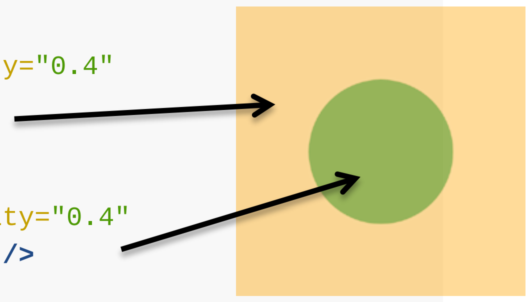


Figure 3-15 Simple SVG definition with a rectangle and a circle with exported Bitmap on the right

Unfortunately, the SVG format has no support for object layout (like column, row and grid). This could be achieved by binding JavaScript calls to place objects dynamically, but it would make the picture output depend on the viewing application capabilities. What we need to achieve creating an EDSL for SVG are both:

1. A Language to create the objects structure
2. A layout engine to easily define objects placement information

We are going to focus here on creating the language. The layout engine will be presented later with a concrete application in 4.2.

When creating an EDSL using our methodology, the workflow is always the same:

- ✓ Define the data structure hierarchy
- ✓ Define the matching classes in TCL using the incrTCL library
    - o Each class will take a configuration closure as last argument of the constructor
        - ➔ The provided closure is then called inside the constructor
    - o Alternatively, the class will have a method called apply, which takes a closure as input argument and runs it as a configuration closure.
- ✓ In each <u>container</u> class (a data structure containing another one), for each <u>aggregated</u> data structure type, create a method which:
    - o Is named accordingly to the aggregated class name
        - ➔ Example: The method in an SVG to add a ***<rect>*** will be called: ***rect***
    - o Takes as input arguments:
        - ➔ The required parameters to create an instance of the aggregated class
        - ➔ A configuration closure to pass to the new instance's constructor
    - o Creates an instance of the aggregated class by passing to the constructor: the parameters and optionally the configuration closure
    - o Optional: Stores it in the appropriate class field if necessary
    - o Optional: Passes the configuration closure to the configuration method, if it is not the constructor

Now we just need to apply these rules to our SVG API (We only focus on general properties and definitions here, the details are available in the source code).

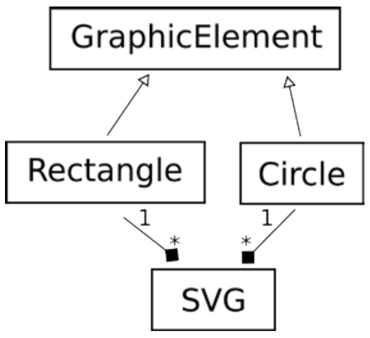


Figure 3-16 SVG simple class hierarchy

The class hierarchy is presented in Figure 3-16 :

- ✓ The *GraphicElement* class holds common properties to *Rectangle* and *Circle*, like the position, filling colour, or opacity.
- ✓ Both *Rectangle* and *Circle* are a *GraphicElement*
- ✓ The *SVG* class can hold multiple instances of *Rectangle* or *Circle.*

Afterwards, we can prepare the source code. We are going to only give code extracts here, the full source can be accessed from the repository (see Appendix A for more details). Following our rules, the SVG class needs:

- ✓ A constructor with a configuration closure input.
- ✓ A list to hold content.
- ✓ Width and Height dimensions.
- ✓ Methods to create *Rectangle* and *Circle* (only *Rectangle* is presented here), which will become elements of our language.

Repository: thesis, Path: sources/3.3-TCLEDSL/ svg-impl.tcl

```
itcl::class SVG {

    odfi::common::classField public width  0
    odfi::common::classField public height 0

    public variable content {}

    ## Runs the closure to configure at creation
    constructor closure {
        odfi::closures::run $closure
    }

    public method rect closure {

        ## Create
        set newRect [::new Rectangle %auto]
        $newRect apply $closure

        ## Save
        lappend content $newRect
    }
}
```

Figure 3-17 SVG language top class implementation extract (full source in repository)

The code extract from Figure 3-17 shows the method called *rect* which acts as constructor for the *Rectangle* class, as well as semantic keyword in the language.

Now we are ready to use our language. We just need an extra function definition to create the SVG instance, which is our top level class. It is presented in Figure 3-18, and will set the resulting variable for the user, to avoid a call in the form of ***[svg { … }]*** which is not comfortable to use for the end-user.

```
## keyword is useless, only for the semantic beauty
proc svg {varName keyword closure} {
    uplevel set $varName [::new SVG %auto $closure]
}

## After this class, the variable $mySvg contains the result
svg mySvg is {

    width 40
    height 40

    rect {
        width 40 ; height 40
        opacity 0.4 ; fill orange
    }

    circle {
        r 10 ; x 20 ; y 20
        fill green ; opacity 0.4
    }
}
```

Figure 3-18 SVG language user view

Finally, we add a method called ***toString*** on each class, which produces the SVG XML (*SVG* produces the ***<svg …>*** mark-up, and calls ***toString*** on the *Rectangle* and *Circle* instances). The results are presented in Figure 3-19. On the right side the produced picture is the same as in Figure 3-15, but is the result from the generated SVG.

```
puts "[$mySvg toString]"
```

```
<svg xmlns="http://www.w3.org/2000/svg"
    version="1.1" width="40" height="40">
<rect x="0" y="0" fill="orange" opacity="0.4"
      width="40" height="40"/>
 <circle cx="20" cy="20" r="10"
      fill="green" opacity="0.4"/>
</svg>
```

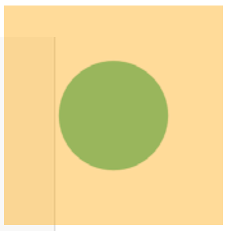


Figure 3-19 Result of generated SVG

This implementation flow was repeated for each of the libraries presented in 4.1, 4.2 and 4.3.

3.3.2 Improved extensibility with the Next Scripting Framework (NSF)

We have just presented a methodology to create EDSLs using the incrTCL framework. As we already mentioned, there are no reasons to limit ourselves to this setup, so we can open the design space to profit from other frameworks' features.

The main advantage of incrTCL is its simplicity, however after developing a few applications some weaknesses appear. We present two of them here:

1. Variable naming confusions can appear between local variables and class fields
2. Language abstraction level enrichment is limited and requires good TCL knowledge

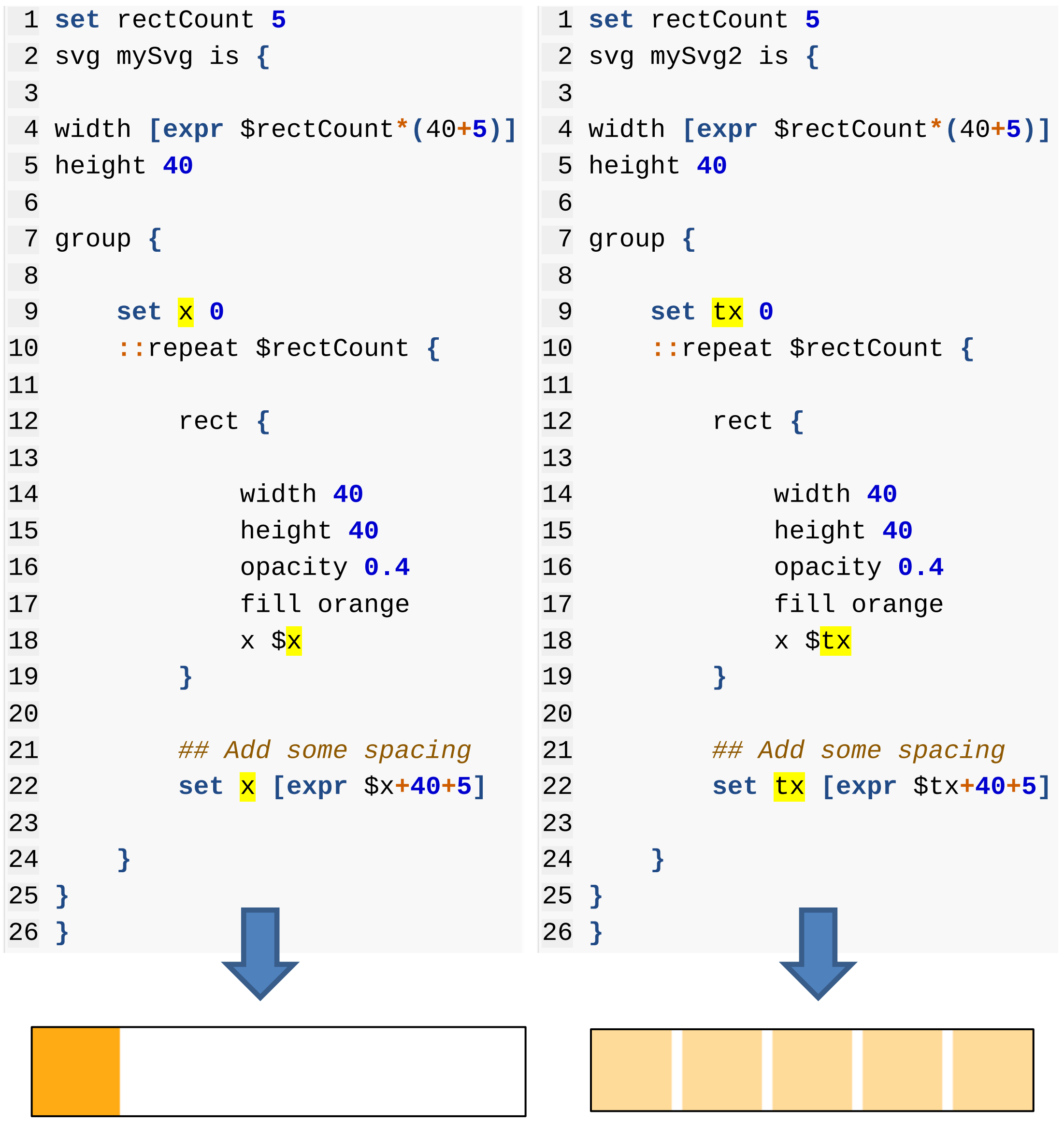
Variable naming issue

Repository: thesis, Path: sources/3.3-TCLEDSL/svg-variable-issue.tcl

The incrTCL framework does not distinguish class field naming from local variables when updating values. This can lead to unnoticed confusions, leading to writing algorithms that modify class fields when the developer intended to work on local variables. This is a very classical problem, faced by nearly all object-oriented programming interfaces in all languages. A gold-rule of object-oriented programming is to always clearly modify class fields by using the self-pointer variable, but in the case of a TCL script, it is not very convenient, and the text editors usually don't provide coloured highlighting of class fields indicating possible confusions.

Following-up on our SVG example, a developer is very likely to create a script that generates a picture, and generate positioning errors. In Figure 3-20 we present two variants of the same code, which generates a row of rectangles. We introduced the SVG ***<g>*** element here, with coordinates ***(x,y)*** although they are not allowed and won't have any effect, they are here only to support our demonstration:

- ✓ On the left, the developer used the ***$x*** variable as temporary coordinate storage. In this case, the ***x*** variable update will only increase the position of the group, and leave the rectangles positions to 0 , as reading from ***$x*** refers to the class-field, and not the parent ***$x***. The result is five rectangles on top of each other.
- ✓ On the right, the variable has been renamed to ***tx*** to avoid conflicts. The script produces the expected result of five rectangles next to each other.



```
set rectCount 5
svg mySvg is {

width [expr $rectCount*(40+5)]
height 40

group {

    set x 0
    ::repeat $rectCount {

        rect {

            width 40
            height 40
            opacity 0.4
            fill orange
            x $x
        }

        ## Add some spacing
        set x [expr $x+40+5]

    }
}
}
```

```
set rectCount 5
svg mySvg2 is {

width [expr $rectCount*(40+5)]
height 40

group {

    set tx 0
    ::repeat $rectCount {

        rect {

            width 40
            height 40
            opacity 0.4
            fill orange
            x $tx
        }

        ## Add some spacing
        set tx [expr $tx+40+5]

    }
}
}
```

Figure 3-20 Variable naming confusion example

Abstraction level improvement

Repository: thesis, Path: sources/3.3-TCLEDSL/svg-improveabstraction-issue.tcl

To demonstrate the issue linked with abstraction level improvement, we can reuse the previous rectangle row generator script, and replace the rectangles by our base example rectangle + circle building block. To improve reusability, we wish to wrap the two-elements set inside a function, to be reused by the end-user, as presented in Figure 3-21.

```
proc rectCircleSet {ix iy} {

    rect {
        width 40 ; height 40
        x $ix ; y $iy
        opacity 0.4 ; fill orange
    }

    circle {
        r 10 ;
        x [expr $ix + 20]
        y [expr $ix + 20]
        fill green ; opacity 0.4
    }
}
```

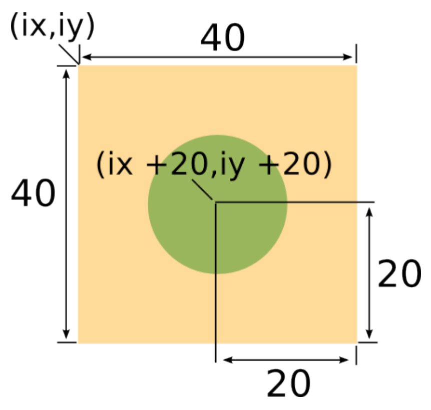


Figure 3-21 Circle in rectangle building block function definition

Using this procedure in our previous generator will produce an error as shown in Figure 3-22:

```
set rectCount 5
svg mySvgwrong is {

width [expr $rectCount*(40+5)]
height 40

group {

    set tx 0
    ::repeat $rectCount {

        rectCircleSet  $tx 0

        ## Add some spacing
        set tx [expr $tx+40+5]

    }
}
}
```

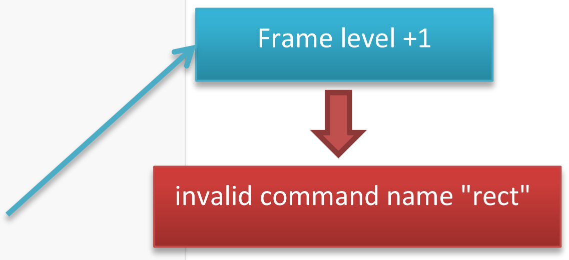


Figure 3-22 Simple function definition used as building block produces an error at runtime

Indeed, the ***rect*** or ***circle*** methods are not visible inside ***rectCircleSet*** because of the extra call level. To solve this issue, we need to call the ***rectCircleSet*** body in following context:

- ✓ Using ***uplevel*** to be in the Group class
- ✓ Using the closures ***applyLambda*** function to pass the procedure input arguments which are not visible anymore once in ***uplevel*** context

The ***rectCircleSet*** procedure can be then rewritten as presented in Figure 3-23, yielding correct results:

```
proc rectCircleSet {ix iy} {

set lambda {

    rect {
        width 40 ; height 40
        x $ix ; y $iy
        opacity 0.4 ; fill orange
    }

    circle {
        r 10 ;
        x [expr $ix + 20]
        y [expr $iy + 20]
        fill green ; opacity 0.4
    }

}

uplevel [list \
    odfi::closures::applyLambda  \
        $lambda [list ix $ix] [list iy $iy] \
]

}
```

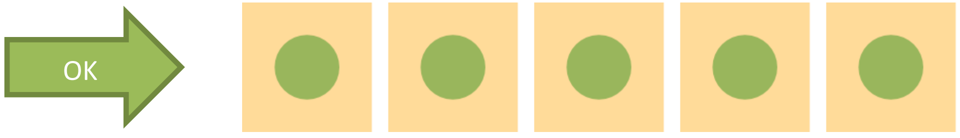


Figure 3-23 User-defined language procedure, with valid call-scheme

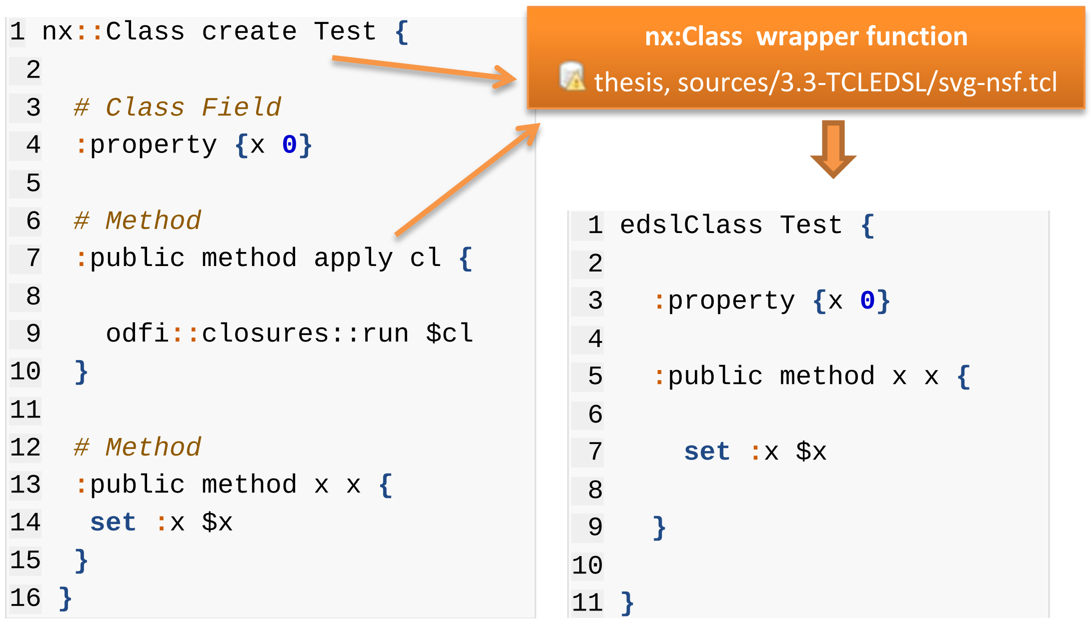
Both of the issues that have been presented can be solved by using another framework than the incrTCL one. We are going to present how the Next scripting framework (NSF, https://next-scripting.org) can be used as a more powerful tool than incrTCL. It is the continuation work of the XOTcl framework [45], and offers a lot of powerful features, which we don't want to detail fully here , but we invite the reader to read the tutorials on the Next Scripting website (A next framework installation is provided in one of the thesis software package, see Appendix A) .

3.3.2.1 *Switching frameworks: semantic and feature issues*

Repository: thesis, Path: sources/3.3-TCLEDSL/svg-nsf.tcl

The first challenge we have to face when switching the underlying Object-Oriented framework is the change in the semantics and features. The NX framework does not handle constructor definition with arguments in a satisfactory way, which is an issue if we follow back our SVG EDSL implementation flow. However, it was defined in the workflow listing that objects could be build without using a configuration closure as constructor argument, but instead by creating an apply method to be called later.

In NX, this "class + apply" implementation flow would be mandatory for all classes, but we can be a little bit creative and define a utility function that would do that for us. To introduce the basic syntax of NX, Figure 3-24 presents the definition of a simple Class, with an appropriate "apply" method. On the left using standard NX syntax, on the right using a utility function (see source code for details).


nx:Class wrapper function
thesis, sources/3.3-TCLEDSL/svg-nsf.tcl

```
nx::Class create Test {

  # Class Field
  :property {x 0}

  # Method
  :public method apply cl {

    odfi::closures::run $cl
  }

  # Method
  :public method x x {
   set :x $x
  }
}
```

```
edslClass Test {

  :property {x 0}

  :public method x x {

    set :x $x

  }

}
```

Figure 3-24 Class definition with configuration closure in NX

Variable name conflict

The class-field name conflict issue is solved explicitly by NX. All class fields and class methods in NX must be called using the ***':'*** character prefixed to their name. This way, all class-level references are made crystal clear. Thus, the user would have to knowingly create the confusion by using variable names starting with ***':'***, and calling their values using the ***${:varname}*** special syntax, as presented in Example 3-17.

```
# NSF style naming with ':'
set :v "Hello"

# Natural usage -> invalid
puts "Res: $:v"

# Explicit name wrapping -> valid
puts "Res: ${:v}"
```

```
Res: $:v
```

```
Res: Hello
```

Example 3-17 Variable and method calling scheme in NX

This clarification comes however to the price of the ***:*** prefix. In Figure 3-25, the first SVG example has been reproduced with an NX implementation. It could be debated if this ***:*** prefix would become a show stopper for the end-user acceptance.

```
svg mySvg is {

    :width 40
    :height 40

    ## Create the rect (x and y to 0 per default)
    :rect {
        :width 40
        :height 40
        :opacity 0.4
        :fill orange
    }

    :circle {
        :x 20
        :y 20
        :fill green
        :opacity 0.4
        :r 10
    }
}
```

Figure 3-25 Simple SVG picture EDSL using the NX framework

3.3.2.2 *Dynamic API Enrichment*

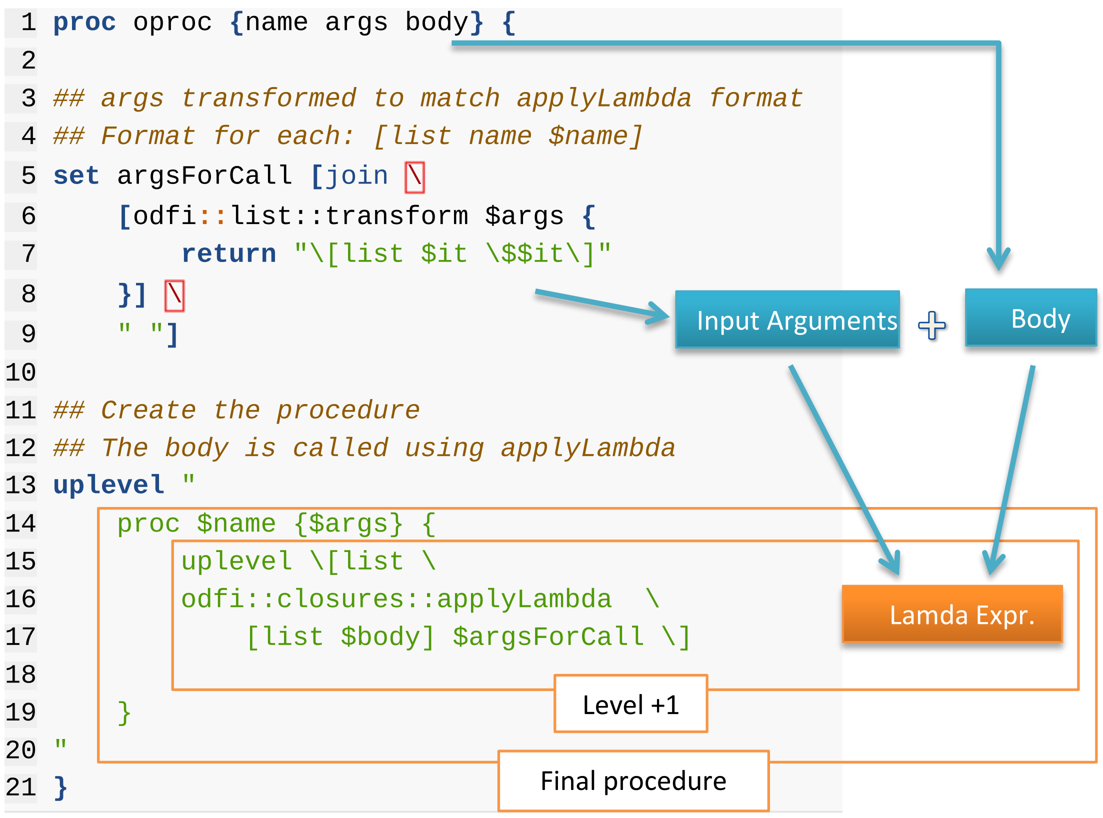
Repository: thesis, Path: sources/3.3-TCLEDSL/svg-nsf.tcl

The second issue we presented in introduction to this section treated the extensibility of the EDSL. In other words, how a user can easily add custom language elements to add abstraction levels to the existing language. We are going to illustrate two ways to proceed here to spare the developer the necessity to add some black-magic boiler plate code because of execution context:

1. One can easily wrap the advanced function calls to hide them, and keep the methodology as presented before.
2. NX allows defining class mixins, which allows the user to write a class, whose content will be added to another one.

3.3.2.2.1 Special procedures

It is enough here to simply wrap the code presented in Figure 3-23 to hide the ***uplevel*** and ***applyLamba*** calls from the developer. Figure 3-26 presents the results.



```
proc oproc {name args body} {

## args transformed to match applyLambda format
## Format for each: [list name $name]
set argsForCall [join \
    [odfi::list::transform $args {
        return "\[list $it \$$it\]"
    }] \
    " "]

## Create the procedure
## The body is called using applyLambda
uplevel "
    proc $name {$args} {
        uplevel \[list \
        odfi::closures::applyLambda  \
            [list $body] $argsForCall \]

    }
"
}
```

Figure 3-26 Procedure definition wrapper for automatic uplevel execution

This approach presents the main advantage of being framework agnostic, and can be reused if the same kind of execution level issue is encountered in another context. The user only needs to know to replace the ***"proc"*** keyword by ***"oproc"***, as can be seen on the right.

The main drawback is that it only runs code in a different level, but does not define real new methods for any class. Standard Object-Oriented features like overriding, method chaining etc… are not available.

```
oproc rectCircleSet {x y} {

 :rect {
    :width 40 ; :height 40
    :x $x ; :y $y
    :opacity 0.4
    :fill orange
 }

 :circle {
    :r 10 ;
    :x [expr $x + 20]
    :y [expr $y + 20]
    :fill green
    :opacity 0.4
 }
}
```

Example 3-18 Rect + Circle using oproc

3.3.2.2.2 NX mixins

An alternative to the previous "oproc" solution is using a class mixin. A mixin is a class, whose methods and properties can be mixed in the definition of another class. It differs from standard inheritance, because a mixin acts like a copy, and only shares its type definition with the target class, but does not alter the inheritance hierarchy.

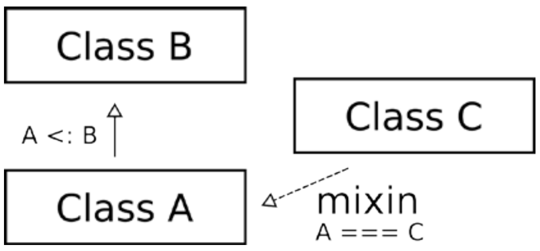


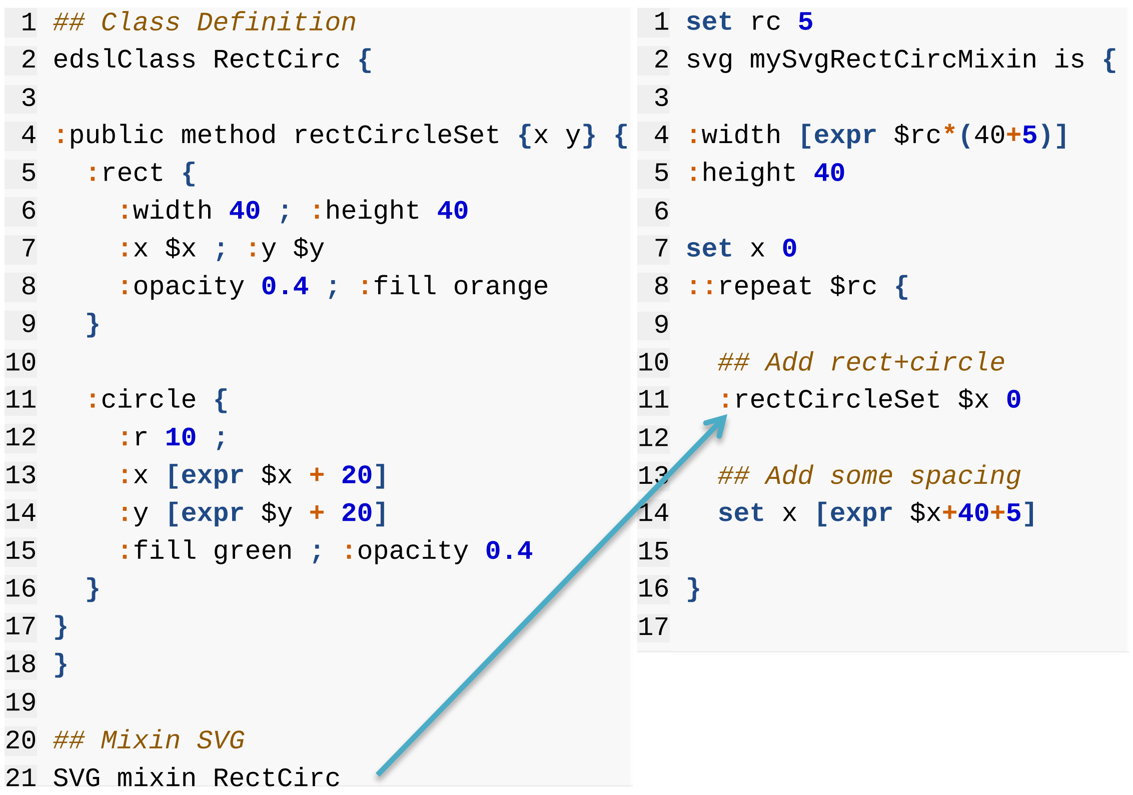
Figure 3-27 Class hierarchy and mixin

In Figure 3-27, class A inherits from class B, and includes the properties, methods and type of class C, but does not inherit from class C (as opposed to multiple inheritance allowed in some languages like C++).

Mixins exist in other programming languages under different names sometimes. They are for example called traits in Scala. To adapt our “Rectangle+Circle” example using a class mixin, the developer must proceed as following:

- ✓ Create a Class with the new method/properties.
- ✓ Call the target Class to mix the new class definitions in its own.
- ✓ Adapt the source code. We now have a full-featured method, its invocation must thus be prefixed with ***“:”***.

Figure 3-28 presents the implementation results, with the new configuration on the left, and its usage to produce a picture on the right. You can note that the source code has been updated with local variables being named ***x*** and ***y*** , without any problem as the class-field can only be updated using the variable named ***:x*** and ***:y***.



```
## Class Definition
edslClass RectCirc {

:public method rectCircleSet {x y} {
  :rect {
    :width 40 ; :height 40
    :x $x ; :y $y
    :opacity 0.4 ; :fill orange
  }

  :circle {
    :r 10 ;
    :x [expr $x + 20]
    :y [expr $y + 20]
    :fill green ; :opacity 0.4
  }
}
}

## Mixin SVG
SVG mixin RectCirc
```

```
set rc 5
svg mySvgRectCircMixin is {

:width [expr $rc*(40+5)]
:height 40

set x 0
::repeat $rc {

  ## Add rect+circle
  :rectCircleSet $x 0

  ## Add some spacing
  set x [expr $x+40+5]

}

```

Figure 3-28 Rectangle + Circle method as class mixin

3.3.3 Discussion and outlook

In this last section, we presented a standard methodology to use lambda functions implemented in TCL to easily create an embedded Language. To illustrate our purpose, we decided to create a mini-framework to draw SVG pictures. We will see in 4.2 a more complex extension brought to this SVG framework to enable object layout through algorithms like column, row, grid etc…

To emphasise on the generic workflow, two object-oriented programming libraries have been selected as support for the implementation:

- ✓ The incrTCL framework proved to be easy to use, but lacks clarity in some cases, as well as some features which would bring flexibility to the designer
- ✓ The NX framework presents less issues and more interesting features than incrTCL (e.g. class mixin is one of them), but forces the usage of ":" prefixes when calling class fields and methods of the EDSL.

Chapter 4 focuses on some applications that are backed by the presented EDSL workflow, and they all use the incrTCL backend variant. For future versions and applications, it would be interesting to consider switching to the NX framework. We believe the acceptance issue caused by the introduction of ***":"*** characters can be balanced by the assurance the users won't get caught by implicit name clashes. As developers of this library, we ourselves encountered this issue a few times, and had a hard time finding the error. We can only imagine the consequences for a simple EDSL developer who might not think about this possibility and us providers also not thinking about documenting this trap properly.

An interesting experiment with the NX framework would be to try to modify the implementation, and introduce an option at class definition, or method definition time, to enable calls without the ***":"*** character. This way, we could keep the incrTCL colon-free call scheme, along with the NX naming protection. This would require in-depth analysis of the sources, and the first basic tries did not look very encouraging (because of NX architecture, not the sources quality).

On the other hand, the same could be done for the incrTCL library, and try to add a mixin feature. The variable naming issue can be solved easily by creating a wrapper function for class-field definitions, which can already be encountered in some examples' sources.

4 Components for Hardware Software co-design

Up to this point we have only setup a basic set of programming principles, which should help us to better formalize our design flow issues from a top-level perspective. We are now going to try to apply our new skills to a Hardware design flow, and show that we can:

- ✓ Bridge the gap between the top-down and bottom-up views of a design.
- ✓ Integrate design steps and abstraction levels with each other along the design flow

The presented developments stem from working on the next generation high-performance network Extoll, whose main components are described in [46] [47] [48]. The Extoll design presents the specificity of being ported to various different target technologies in different configurations. The generic architecture of the Extoll network controller is presented in Figure 4-1. It outlines two generic components combined together:

3. The system interface features a host interface, an HTAX [49] network on chip (NOC) to connect to some hardware functional units and a register file for control and status registers.
4. The network controller holds the hardware which units provides the network functionalities to the system, like low-latency messaging or shared memory support, and a certain number of network links to connected nodes with each others.

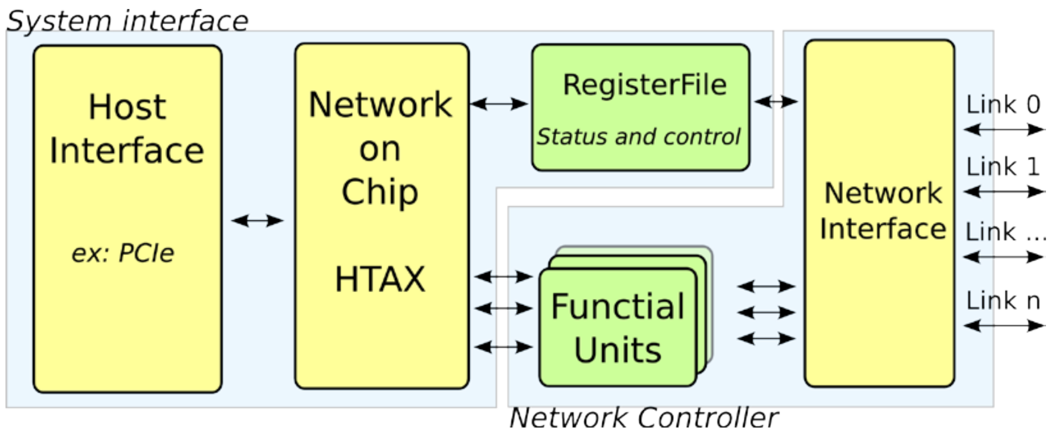


Figure 4-1 Extoll NIC generic architecture

An overview of the possible Extoll configuration is provided in the following table. It does not include the use cases for which only the system interface was reused, or the whole configuration integrated in a special context.

Target Name	Host	NOC width	Links Count	Link Size	Technology
Ventoux	Hypertransport	64bit	4	X4	Xilinx Virtex 6
Galibier	PCIe Gen2	64bit	4	X4	Xilinx virtex 6
Aspin	PCIe Gen3	128 bit	4	X12	Xilinx virtex 7
Gan Ainm	PCIe Gen2	128bit	6	X8	Altera Stratix 5
Tourmalet	PCie Gen3	256 bit	7	X12	ASIC TSMC 65 nm
	Hypertransport				

We can see that depending on the target technology, the capabilities of the network vary. The less network links available, the less network topologies can be supported. Also, the width of the network on chip limits the network communication available bandwidth.

We chose to focus in this chapter on a set of applications used in the design flows applied to Extoll's various configurations. They have the advantage of being highly reusable across various architecture specifications, and span from hardware definition to software interface. The chosen set is presented in Figure 4-2:

- ✓ RFG is a register file generator used to create the "RegisterFile" component on the architecture picture.
- ✓ An ASIC Floorplanning programming interface as created for the implementation of the Extoll Tourmalet configuration
- ✓ A part definition language was created to support Extoll printed circuit board design.
- ✓ The Object to XML library (named OOXOO) is a high level XML data binding technology used in various design flow related applications. It features a special RegisterFile interface enabling easy support on the software side for the various Extoll controllers.

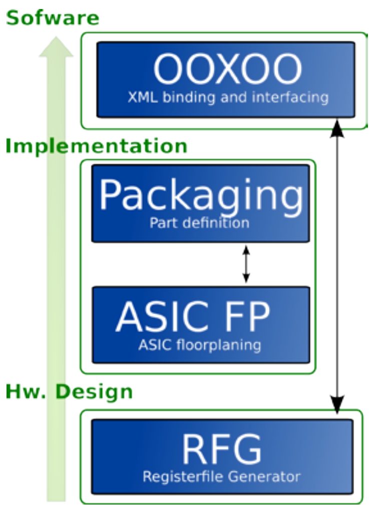


Figure 4-2 Chapter 3 components overview

4.1 Register file generator

An important component of most digital hardware designs is a register file. Indeed, the various functional units in a hardware hierarchy feature special registers, which are used for configuration, control and status reporting. Such registers provide for example:

- ✓ Performance counters (like how many data packets have been processed)
- ✓ States
- ✓ Control bits to start/stop hardware functions, and poll for completion

More specifically, they will be accessed by the software which is driving or monitoring the hardware function. However, this software can only access data through Read or Write operations (also called load and store) issued to addresses. Those addresses can point, depending on the memory mapping, to various locations: process memory space, I/O address space to external devices.

Important here is to notice that a software runs inside a continuous virtual address space, while the hardware registers can be dispatched in a hierarchy, as presented in Figure 4-3. Both Read or Write at location A and B in the address space target two different locations in the hardware hierarchy. One could argue about the option of maintaining by hand a simple address decoder in the hardware, but some issues would appear very fast:

- ✓ A register file is usually used in multiple designs; do we want to repeat this operation every time?
- ✓ Every change (like inserting a new register) will require modifying the address information in the software, as well as maintaining the documentation.
- ✓ Some registers have to be mapped to special resources, like counters, or SRAM blocks (for registers which handle data loads).
- ✓ Registers can be grouped and distributed among physically distinct hardware units

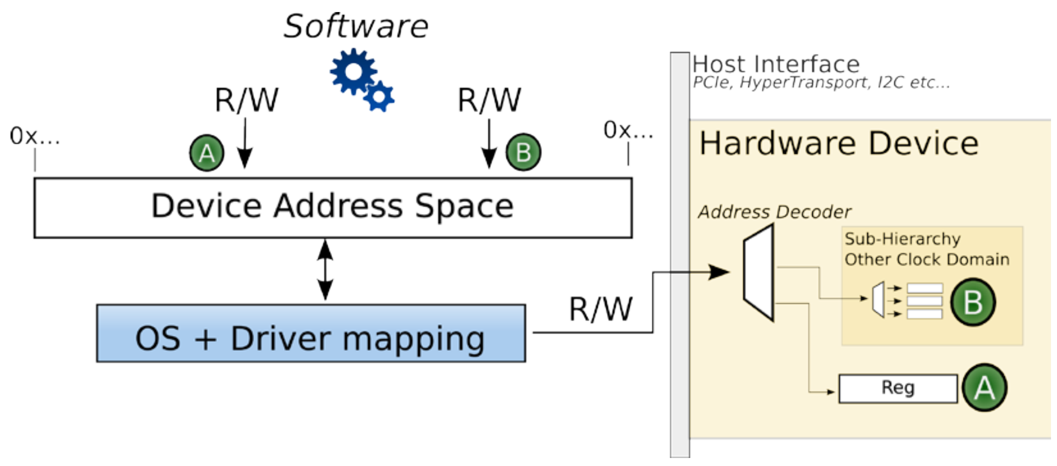


Figure 4-3 Linear Address space to Hierarchical registers dispatching

As the tasks are generic and repetitive for all designs, they are very likely to be handled by a tool that would generate all the required outputs. Such generators exist already, mostly available as commercial software, thus closed to improvements, which makes them difficult to adapt to any specific requirements. Flexibility is critical for large designs requiring deep customisation, like the Extoll project, which is why an alternative implementation called RFS was developed by C. Leber in [50]. We invite the reader to consult the thesis for a state of the art reference.

We present in 4.1.2 a novel highly flexible register file generator implementation, called *RegisterFileGenerator* (RFG). It is publicly available to use and open to contributions under a GPL License (see Appendix A).

4.1.1 RFS: Workflow and limitations

Before presenting the architecture of RFG, we want to highlight a few issues encountered while working with RFS. The workflow of RFS was based on an XML configuration document describing the register file hierarchy (i.e. groups) and its elements (registers, rams etc…). RFS would then generate various outputs like Verilog sources, HTML documentation, or a result XML file with calculated addresses information (see Figure 4-4).

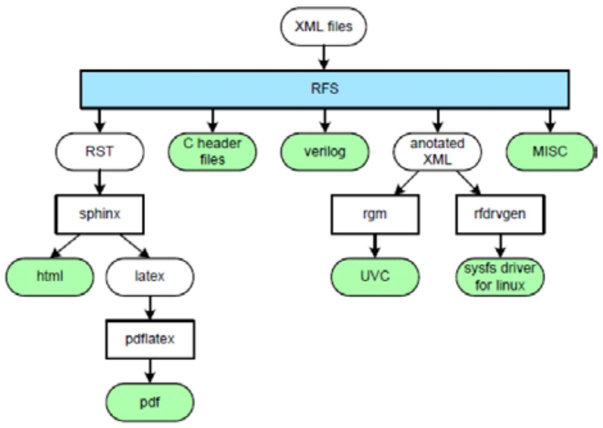


Figure 4-4 RFS Outputs

4.1.1.1 XML Format issues

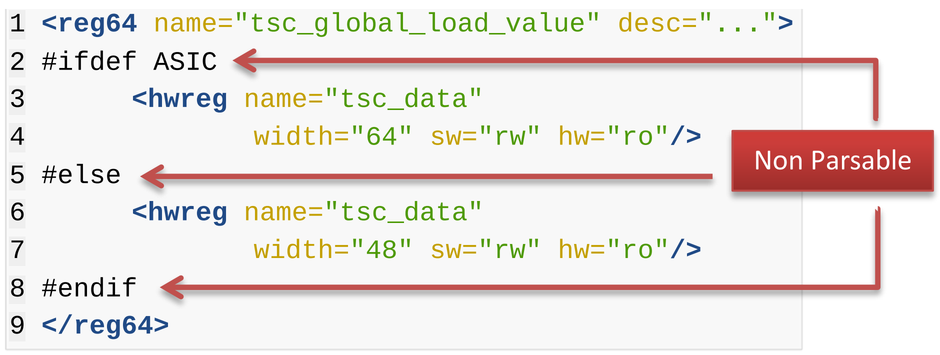
Repository: thesis, Path: sources/4.1-RFG/info_rf.xml

The XML format specified presents some issues. Indeed, an XML tree is static, and some special control structures are required to make a register file definition flexible, for example:

1. Some elements need to be redefined multiple times to the identical. A <repeat> loop element was therefor introduced:

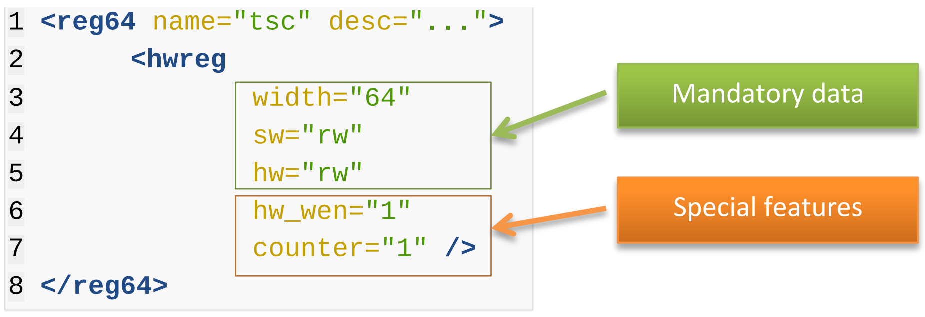
```
<repeat loop="8" name="scratchpad">
  <reg64 name="scratchpad" desc="...">
      <hwreg name="data"
                        width="64"
                        sw="rw"
                        hw=""
                        reset="64'h0" desc="..."/>
      </reg64>
</repeat>
```

2. A single RFS specification may be shared by various designs, with slight difference. As no control structures are available in XML, and none have been implemented, the workflow relied on using C pre-processor calls to prepare the XML files….which forces the user to write non-conform XML (!):



```
<reg64 name="tsc_global_load_value" desc="...">
#ifdef ASIC
      <hwreg name="tsc_data"
              width="64" sw="rw" hw="ro"/>
#else
      <hwreg name="tsc_data"
              width="48" sw="rw" hw="ro"/>
#endif
</reg64>
```

3. The special features, like marking a register as being a counter, or defining read/write rights are specified with XML attributes, and are not distinguished from mandatory data like register names. This is an obstacle for full flexibility and user customisation:



```
<reg64 name="tsc" desc="...">
      <hwreg
              width="64"
              sw="rw"
              hw="rw"
              hw_wen="1"
              counter="1" />
</reg64>
```

An XML document is very flexible and well-suited as an exchange format. The annotated XML output provided by RFS appeared a very useful feature, as it allowed the team to write special software build on top of this XML output, among which: a Linux SYSFS driver, some UVM System Verilog for verification environments, a generic software access interface in Scala (see 0) etc…

However, XML as an input language for end-users is not a very good choice, both because of the issues we just presented, and of the syntax overhead it presents.

4.1.1.2 Implementation in C

Finally, RFS was implemented using the C language, which requires compilation for each target system. The XML parsing was implemented per hand to avoid any library binding, which makes the source code quite cumbersome to understand. The C language also lacks flexibility, and does not provide any intuitive and ready-to-use infrastructure for users to add custom functionalities, like a plugin mechanism.

Considering the relative simplicity of the task, and the high requirement for flexibility, it is not justified to use a compiled language, even less a low-level one like C. A dynamic script language, such as Python, Ruby, Perl would have been better suited, not to mention the TCL language which is very present in EDA software.

4.1.2 RFG implementation

Repository: odfi-rfg, Path: tcl/rfg.tm

One of the very first requirements for a register file generator's user interface is to be able to clearly map the hierarchy of registers. An XML format was a meaningful solution, as it natively provided a tree mapping of the register file structure, but with the drawbacks that have been highlighted previously.

If we want to lighten the input format, we first need to find a solution that will still allow a clear understanding of the register file's hierarchical structure. Based on the work presented in 3.3, it appears that we can reach this goal by creating an EDSL for the TCL language. By doing so, we gain following advantages:

- ✓ The language naturally expresses the hierarchy
- ✓ All standard programming control structures (conditions, loops, variables etc…) are available.
- ✓ No need to maintain an input format parser
- ✓ The user can structure the code as he wishes:
    - o Separated files can just be read using the TCL source command
    - o Special TCL functions which create register sets can be created and reused at wish

We will present here the base elements of our new language. The presented data structure hierarchy focuses on the main classes, the details can be consulted in the source code. We also made the choice to purposely separate all processes of the tool chain in distinct components: register file specification, address calculation and output generators of all kind are all implemented independently from each other to maximise flexibility.

4.1.2.1 Language elements

Defining the language is fairly easy. We globally want be able to find back all the elements that were supported by RFS [50]. Registers, RamBlocks and hierarchies (i.e. groups that were named ***regroot*** in RFS) stay somehow the same. The parameters applied to each element, like read/write rights and special features are now supported through a generic data structure called *Attributes*. Each of those elements can hold multiple instances of the *Attributes* class, which acts as a named container (like "sw" and "hw", for the software and hardware attribute groups), each containing some attribute definitions, which are *{name value}* tuples.

To sum-up, we need support for:

- ✓ RegisterFile, Groups (ex-regroot), Register and RamBlock.
- ✓ Attributes containers on each structural element (The class is named *Attributes*).
- ✓ Attribute tuples stored in the *Attributes* container.

We also made the choice to delegate the base register width to a global configuration parameter, so that the software stays independent from the target host architecture. It can be adapted to support special devices, like small Integrated Circuits, which for example often have 8-bit wide registers reachable over an I^{2}C interface. The ***<reg64>*** in RFS element thus became a ***register***.

We now simply repeat the workflow presented in 3.3, and specify the class hierarchy as presented in Figure 4-5 (simplified version, refer to source code for full details).

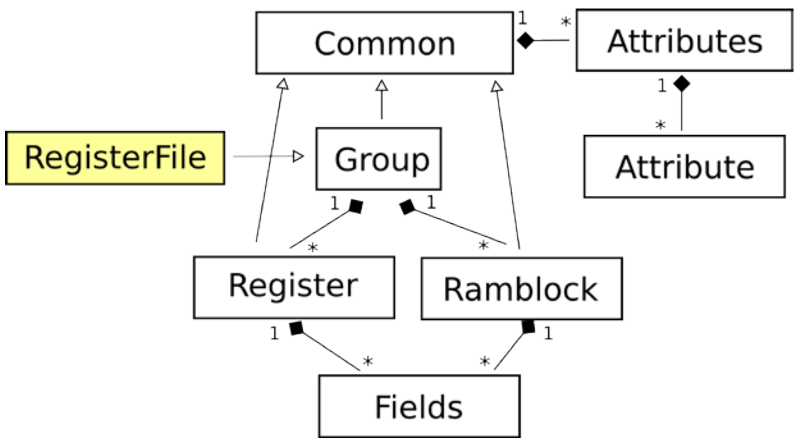
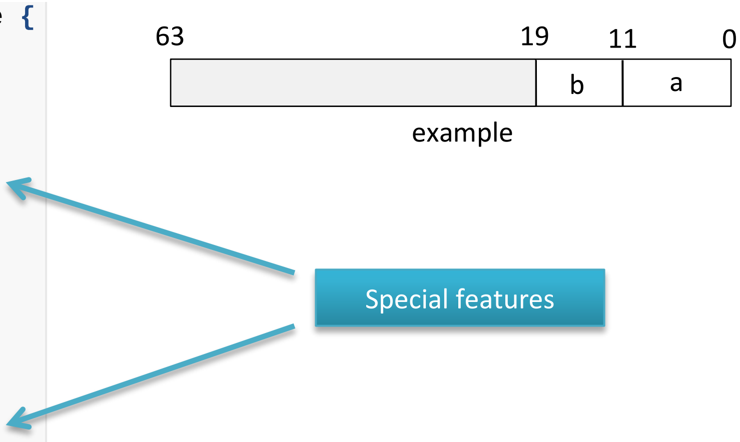


Figure 4-5 RFG simplified class hierarchy

Using proper method naming, and considering that all elements in the hierarchy must have a name, the resulting implementation allows us to now write our register file specification using following inputs:

Register with field

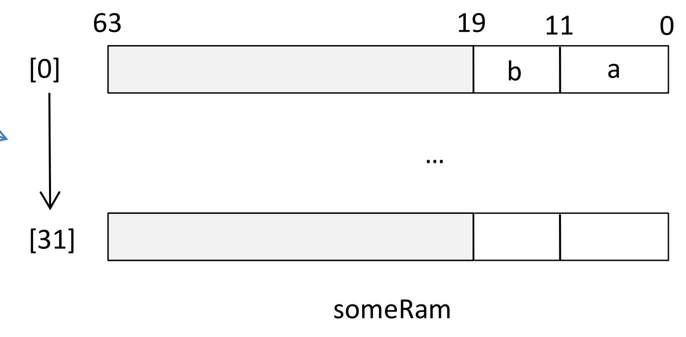
```
register example {

  field a {
    width 12
    software ro
    hardware wo
  }

  field b {
    width 8
    software rw
    hardware wo
  }
```



This example shows how to create a *Register* named “example”, with two fields “a” and “b”, and a default width of 64 bits. The ***software*** and ***hardware*** methods are shortcuts for creating *Attributes* groups named *software* and *hardware*.

RamBlock

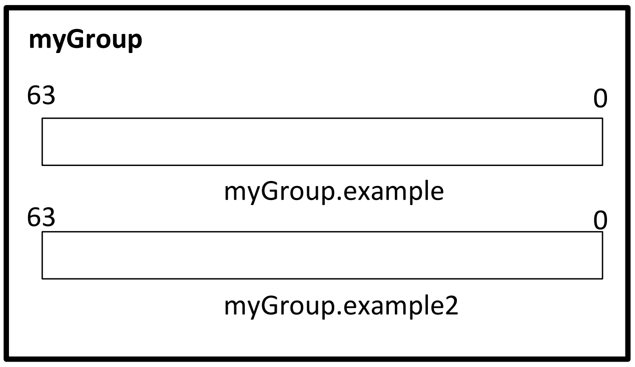
```
ramBlock someRam {

  depth 32

  field a {
    width 12
    software ro
    hardware wo
  }

  field b {
    width 8
    software rw
    hardware wo
  }

```



A RamBlock definition resembles a register definition, but must specify a depth which characterizes the number of elements hold by the memory array. RamBlock is a fully supported class definition in RFG, but it could be argued that it is equivalent to a group definition that would replicate a register definition “depth” times, and have a special attribute to signal it should be seen as a Ram. Because RamBlock definitions are widely used as memory arrays in register files, we decided to keep their support distinct from the Register element.

Group

```
group myGroup {

  register example {

  }
  register example2 {

  }

}
```



A *Group* is a purely virtual structural element. It shares its name with its descendants and helps organising the data structures. It might also share its special features with its descendants, but this behaviour would have to be specified by the software component supporting the concerned features.

Top Register File

```
package require osys::rfg

osys::rfg::registerFile toptmp {

  group myGroup {

  }

  register example {

  }

}
```

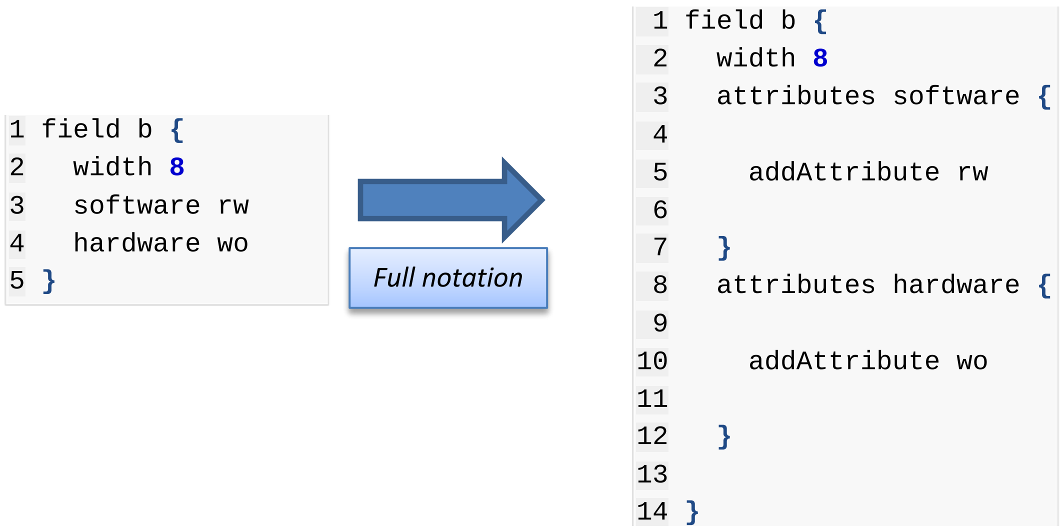
A *RegisterFile* instance is a start-point to create a register file definition. It may be used standalone, or as part of another register file. It shares the behaviour of a group but is considered a real structural element.

4.1.2.2 *In-depth customisation: Attributes specification*

We have previously defined two special data structures called *Attributes* and *Attribute* to support special features. One of the critics aimed at the RFS format, was that those features (like marking a register as being a counter) were hard-coded in the configuration XML. In RFG, the user can set attributes on all elements. The attributes must be part of a named group, so that the special features can be logically sorted depending on their application context.

For example, RFS required read/write rights for both software and hardware. This specification was used when producing the Verilog output to remove the unnecessary logic. For a register marked as non-writable by the software, the address decoder for packets coming from the software would not feature the logic to handle a write request. This read/write specification can differ between the hardware or the software side, which is the reason why we need the named attribute groups.

In the register example presented previously, we introduced a shortcut attribute specification for hardware and software. The standard API usage however expands to the following:



```
field b {
  width 8
  software rw
  hardware wo
}
```

```
field b {
  width 8
  attributes software {

    addAttribute rw

  }
  attributes hardware {

    addAttribute wo

  }

}
```

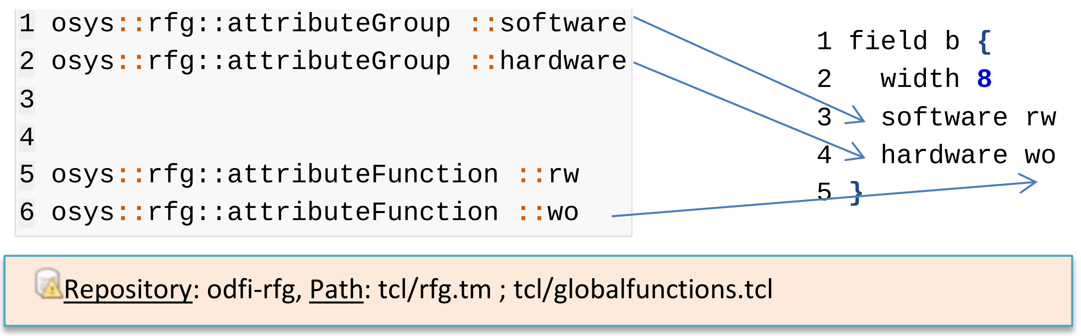
Figure 4-6 Attributes group definition syntax

Supporting attributes

To simplify the notation, it is enough to follow the procedure wrapping methodology presented in 3.3:

- ✓ The ***software*** and ***hardware*** methods wrap the calls to ***attributes "software" closure*** and ***attributes "hardware" closure***.
- ✓ The ***rw*** and ***wo*** calls wrap ***addAttribute rw*** and ***addAttribute ro***.

To ease the definition of attribute functions, the RFG API offers two procedures that create the “attributes name closure” (for groups) and “addAttribute” (for attributes) wrapper using a simplified interface. Some of the core supported features are listed in the following extract:



```
osys::rfg::attributeGroup ::software
osys::rfg::attributeGroup ::hardware


osys::rfg::attributeFunction ::rw
osys::rfg::attributeFunction ::wo
```

```
field b {
  width 8
  software rw
  hardware wo
}
```

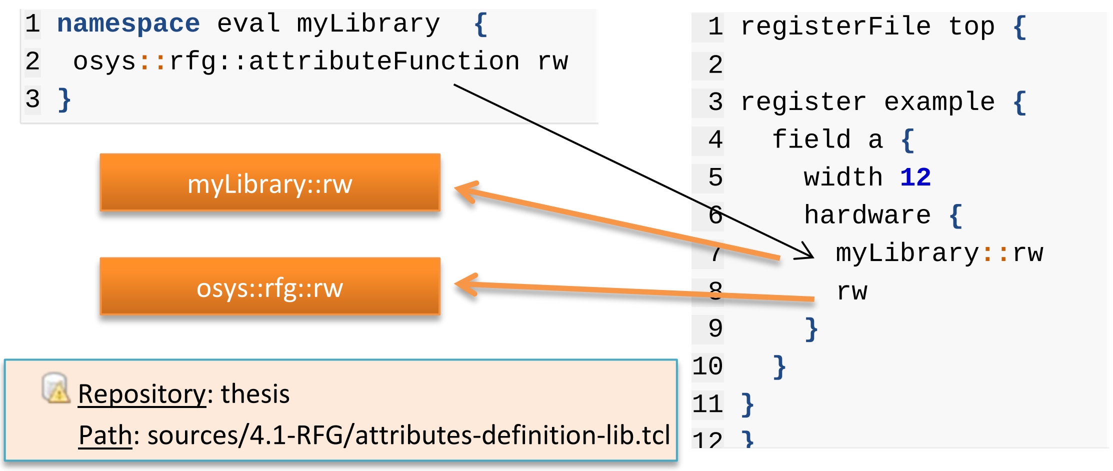
Repository: odfi-rfg, Path: tcl/rfg.tm ; tcl/globalfunctions.tcl

Figure 4-7 RFG attributes group and attribute specification using wrappers

Finally, we have to consider the issue of name confusion that might occur if different software components specify new attributes. The ***attributeFunction*** and ***attributeGroup*** both take an input argument, which is used to name the procedures they create. Two issues have to be considered:

1. Naming should not overlap. In the presented example, the input names lead to global functions creation, because of the leading ***::*** prefix. This is acceptable for the core API attribute functions, but a library developer should be careful to keep his functions under a namespace.
2. Attribute names cannot overlap as well. The attribute’s name which is initialised to the ***attributeFunction*** input argument will therefore automatically be prefixed with the name of the namespace under which the ***attributeFunction*** is called.

To be more concrete, Figure 4-8 presents the naming output for the standard core attributes, and for an attribute defined in a namespace, showing no name overlapping occurs although the chosen base names are identical.



```
namespace eval myLibrary  {
 osys::rfg::attributeFunction rw
}
```

```
registerFile top {

register example {
  field a {
    width 12
    hardware {
      myLibrary::rw
      rw
    }
  }
}
```

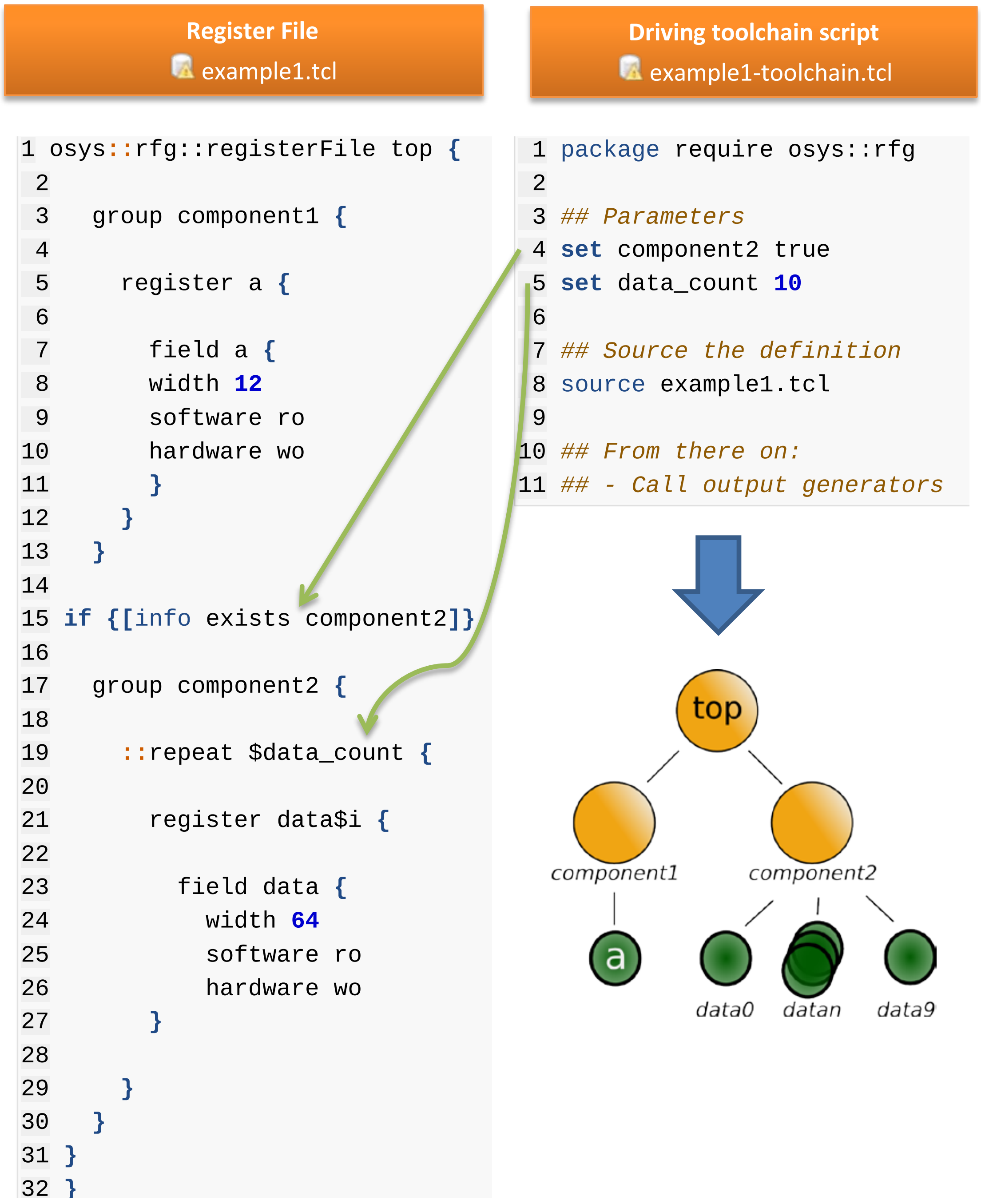
Repository: thesis
Path: sources/4.1-RFG/attributes-definition-lib.tcl

Figure 4-8 Attribute naming is secured using namespaces

4.1.2.3 *An example*

Repository: thesis
Path: sources/4.1-RFG/example1.tcl ; sources/4.1-RFG/example1-toolchain.tcl

We can now write a simple complete example. Figure 4-9 presents the source code, in which we can see that repetitions and conditions which were supported by special XML constructs and pre-processor directives in RFS are now delegated to standard TCL control structures.



```tcl
osys::rfg::registerFile top {

  group component1 {

    register a {

      field a {
      width 12
      software ro
      hardware wo
      }
    }
  }

if {[info exists component2]}

  group component2 {

    ::repeat $data_count {

      register data$i {

        field data {
          width 64
          software ro
          hardware wo
      }

    }
  }
}
}
```

```tcl
package require osys::rfg

## Parameters
set component2 true
set data_count 10

## Source the definition
source example1.tcl

## From there on:
## - Call output generators
```

Figure 4-9 Simple RFG example with conditional processing and a repeated register

4.1.3 RFS backward compatibility

An interesting property of both XML and our new programming interface is the hierarchical nature of the input language. In other words, both input format are structured trees. When considering switching from RFS to RFG, we want to convert from an XML format, to a structured text output, that is to say transform the XML tree to a text tree. This task can be easily handled using the existing and well-supported Extensible Stylesheet Language Transformations (XSLT) technology [51][1].

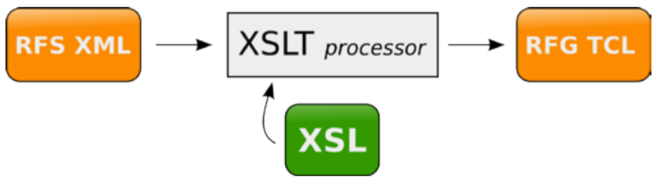


Figure 4-10 RFS to RFG conversion using XSLT

Repository: odfi-rfg, Path: xsl/rfs-to-rfg.xsl, Tool: bin/rfs_to_rfg

4.1.4 Processing chain components

Now that we defined the basic input format and features to specify a register file, we can call the components which will process the structured tree. Those additional software pieces can perform various tasks like optimisation, analysis or more commonly generate outputs. We are going to present and discuss some vital components for RFG to be usable in a hardware design: Address Calculation for the register file elements, a Verilog HDL output, and a documentation output example.

4.1.4.1 Hierarchical address calculation

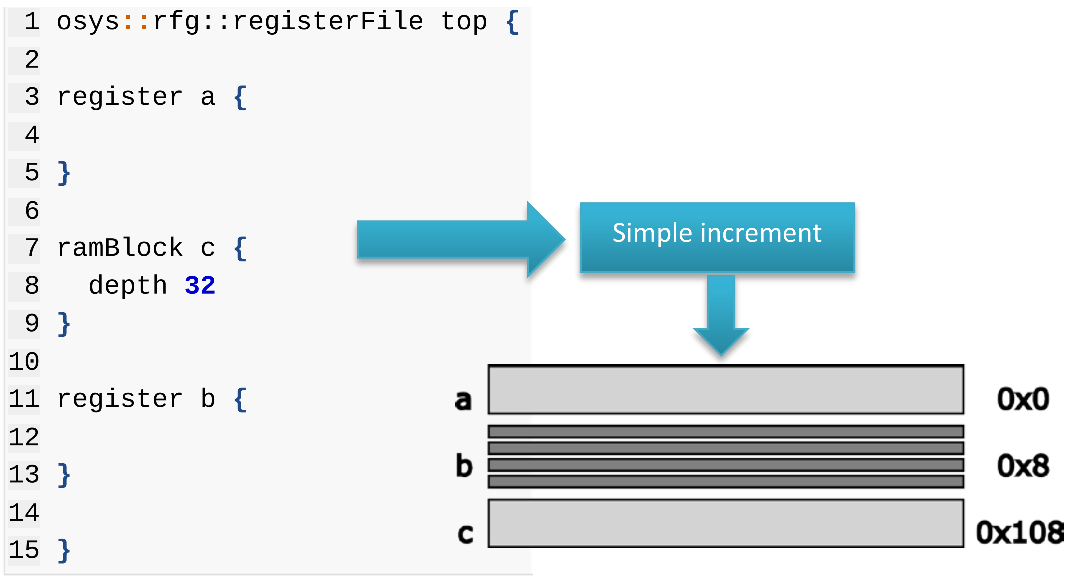
Please note that this work does not make any assumption on the base register bit width. All presented addresses refer to a generic “number of elements”.

To be able to map the register file to hardware, and let the software know how to access the register file elements, the hierarchy must be mapped into a continuous address space.

[1] We invite the reader to consult sources and XSLT tutorials widely available on the web for details .

Intuitively, one would walk the element tree using an In-Order depth first search, so that all the leaf elements are encountered in their order of specification, and increment an address counter by the amount of bytes occupied. We can write such a simple function, along with an SVG view generator based on work presented in 4.2, to obtain the result presented in Figure 4-11. The input format has been purposely kept very simple.

⚠ Repository: odfi-rfg , Path: tcl/address-linear/address-linear.tm
⚠ Repository: thesis , Path: sources/4.1-RFG/address-linear.tcl

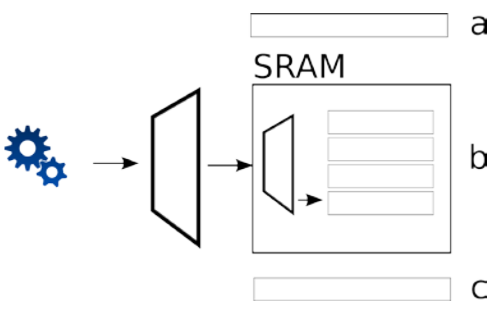


```
osys::rfg::registerFile top {

register a {

}

ramBlock c {
  depth 32
}

register b {

}

}
```

Figure 4-11 Simple increment address calculation

This method presents the drawback of ignoring hierarchies, which leads to a useless consumption of resources when mapping to hardware. Indeed, a RAM represents a hierarchy because the register file logic delegates the read/write to the physical RAM memory component, when its calculated address range is matched. If we draw the circuit leading to interfacing with the RAM, it appears that we have two address decoders:

- ✓ One decoder (a < b < c) to differentiate the RAM from the registers
- ✓ One decoder inside the RAM to select the correct word line.



Because a top-level register file may be an aggregation of hierarchical sub register files (as described in [50]), the same issue would be encountered in that case. A hierarchical register file can be seen as an address range, whose precise address-to-element decoding is delegated to another physical component (in that case another register file).

We thus need to perform a hierarchical address calculation, where the addresses can be efficiently tested for belonging to a sub-hierarchy, which we will call *Region* subsequently as well as in the source code. We propose here to analyse the implementation of such a calculation.

The idea is to create a base address for a region, which can be tested using only a minimal amount of most significant bits in an address. Each region having a certain size, it will consume a certain number of address bits: nb. By rounding up the bits size to the next power of two, we get a bit value of 1 at position $(nb + 1)$, which can be used to test the region destination.

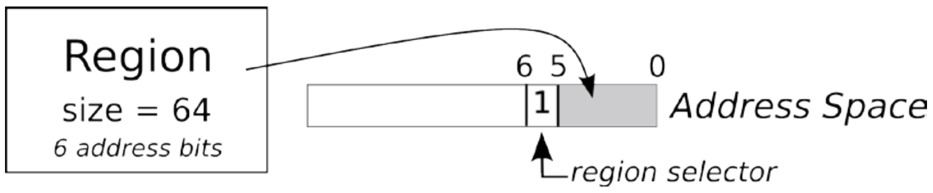


Figure 4-12 Hierarchical Address for a single region

By repeating this process for all addressable elements in the register file definition (registers, ram blocks or sub register files), we obtain for each an address aligned to the size, which can be splint in two parts:

- ✓ The selector bits are the most significant bits which can be tested to match an address to the region.
- ✓ The offset bits are the least significant bits which address the actual addressed element in the matched region.

Two elements have a specific behaviour in front of this address definition:

- ✓ Registers are equivalent to a region containing only one element. Thus the address is only made of selector bits, which when match only select this precise register.
- ✓ Sub register files have no size calculated when building the hierarchy definition.

To overcome the fact that register files don't record their size per default, the address calculation algorithm first needs to determine all the sizes of the hierarchy's regions and then dispatch the addresses. This can simply be implemented using two recursion steps: size and address, but a more elegant way to avoid recursion is to opt for a three-passes map-reduce-dispatch algorithm:

1. Map all the hierarchical components (virtual groups are ignored) to their discrete components (registers, rams and sub register files).
2. Reduce all hierarchical components' mapped content to their size, and annotate.
3. Distribute addresses by just walking the tree.

Figure 4-13 presents an example map-reduce outcome applied to a register file definition with multiple hierarchies (the source code is available in the repository). The Rams have a depth of 2048 bytes and the registers a width of 8 bytes.

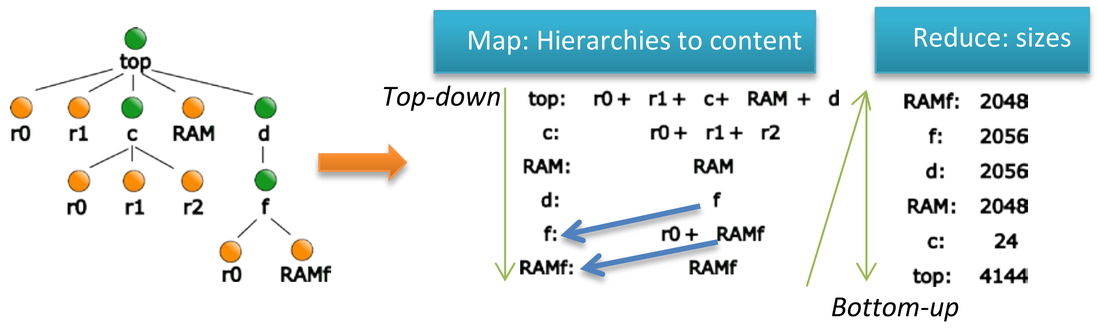


Figure 4-13 Map-reduce sizes outcome

After the map-reduce step, all the elements are annotated with a size, and a simple function walking through the hierarchy can apply the address calculation equations:

$$address = \left(\frac{currentAddress + (size - 1)}{size}\right) * size$$

$$currentAddress = address + size$$

The two equations set the addresses then in the following way:

1. First, the size is rounded to the next valid power of two.
2. Second, the address is simply the current address incremented by $size - 1$ because addresses start at 0. The result of $currentAddress + (size - 1)$ represents the last address of the region (because of the increment by the size). The divide by and multiple operations reset the least significant bits to 0, which leads to the base address of the region.
3. Finally, the $currentAddress$ for the next element is set to the address of the region, incremented by its size, which leads to the next valid address.

4.1.4.1.1 Addressing strategies selection

The address calculation we just presented is only one possible option and does not perform any intelligent work. The strategy adopted in the RFG API definition shows that various strategies could be tested to optimise the address space usage, or the address selector calculation to improve the hardware mapping output.

For example, one could think about grouping all consecutive registers into virtual Regions only used during address calculation, so that such register groups can have a global address selector. A hardware description language generator could then have the possibility to create an implementation which respects nice convention for proper clock gating detection.

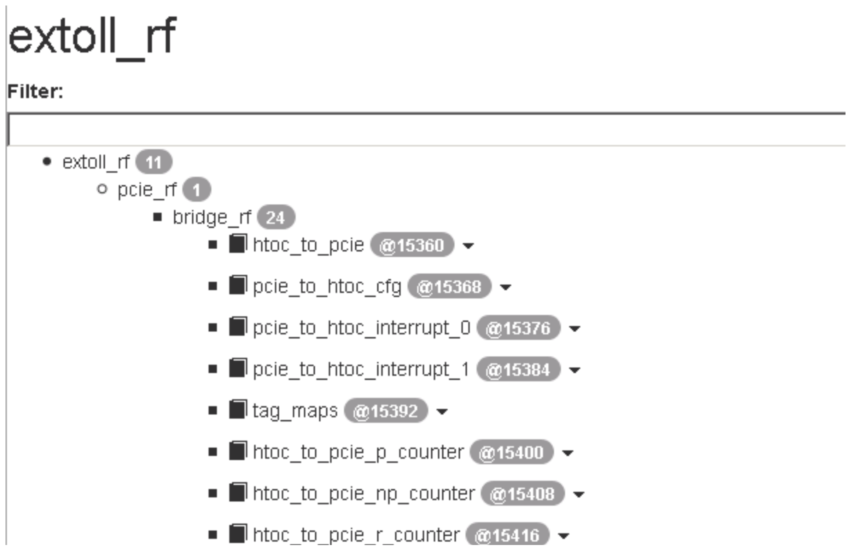
4.1.4.2 *Verilog HDL*

For compatibility reasons, the Verilog output generator was implemented to reproduce the RFS output format and features, which is described in [50] . However, at least two hardware construction features, which are added per hand case-by-case to designs, could be integrated to the generator:

- ✓ Hierarchical signal synchronisation between clock domains.
- ✓ Hierarchical signal serialisation for long wires

4.1.4.3 *Documentation*

A documentation output should allow a fast access to the hierarchy description and the attributes set by all the components in the design chain. Many options can be explored for this purpose, but we chose for a started to implement a simple html browser contained in one file, featuring a tree view of a register file. A screenshot is presented in Figure 4-14, but the sources for this thesis contain a generated html file ready to be opened in any web browser in: liverun-project/src/main/webapp/rfg-doc/example_doc.html.



extoll_rf

Filter:

- extoll_rf 11
  - pcie_rf 1
    - bridge_rf 24
      - htoc_to_pcie @15360
      - pcie_to_htoc_cfg @15368
      - pcie_to_htoc_interrupt_0 @15376
      - pcie_to_htoc_interrupt_1 @15384
      - tag_maps @15392
      - htoc_to_pcie_p_counter @15400
      - htoc_to_pcie_np_counter @15408
      - htoc_to_pcie_r_counter @15416

Figure 4-14 HTML documentation for an Extoll hardware register file (screenshot)

The virtual groups are represented by bullet nodes in the hierarchical list, while registers are nodes featuring a special book-looking icon. A filter box on the top hides the HTML elements whose name don't match the user input. This allows a fast search in large register file definitions.

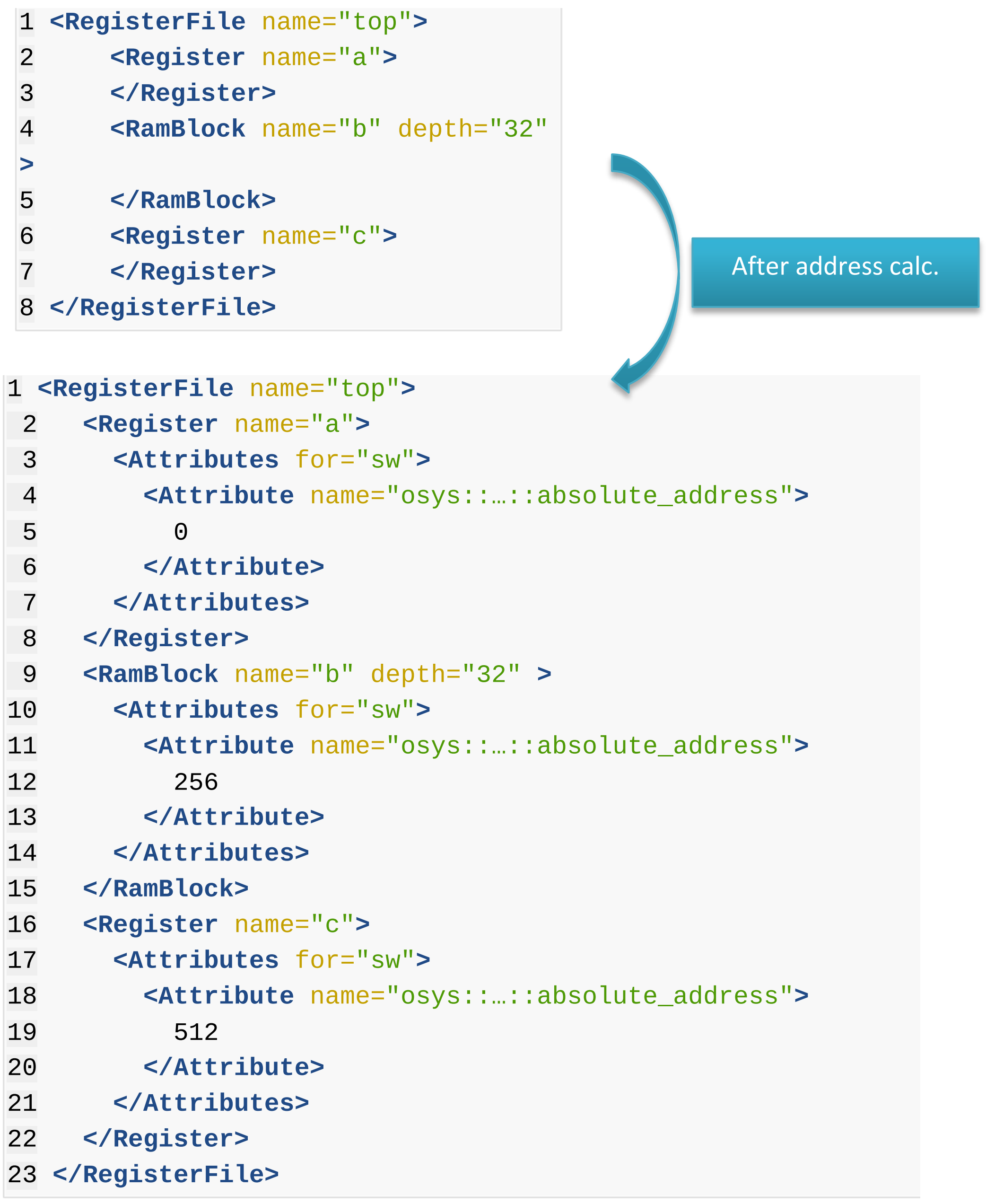
4.1.4.4 XML output

An XML output is of great help to exchange a register file description in a language agnostic format, while not loosing the structure. As presented earlier, RFS was solely based on XML both as an input format, and as an output format (called annotated XML) where calculated addresses would be mirrored.

For RFG, we reworked the XML to make it closely mirror the TCL data structures. This way, the XML output provides a state of the register file structure along with all the attributes set on the elements at a given moment. The implementation can be totally generic, and an input path to recreate the TCL data structures from the XML could also be imagined in case various TCL tools would need to access the RFG flow output without the possibility to integrate correctly with each other.

Figure 4-15 presents the XML output before and after address calculation, for a very simple register file with two registers and a RAM block.

⚠ Repository: thesis, Path: sources/4.1-RFG/xml-output.tcl



```
<RegisterFile name="top">
    <Register name="a">
    </Register>
    <RamBlock name="b" depth="32"
>
    </RamBlock>
    <Register name="c">
    </Register>
</RegisterFile>
```

```
<RegisterFile name="top">
   <Register name="a">
     <Attributes for="sw">
       <Attribute name="osys::…::absolute_address">
         0
       </Attribute>
     </Attributes>
   </Register>
   <RamBlock name="b" depth="32" >
     <Attributes for="sw">
       <Attribute name="osys::…::absolute_address">
         256
       </Attribute>
     </Attributes>
   </RamBlock>
   <Register name="c">
     <Attributes for="sw">
       <Attribute name="osys::…::absolute_address">
         512
       </Attribute>
     </Attributes>
   </Register>
</RegisterFile>
```

Figure 4-15 RFG XML before and after address calculation

4.1.5 Software interface for the Java Virtual Machine using Scala

The software interfaces which can be generated depend on the usage context. Some Linux kernel interfaces and special drivers are available, mostly for the C language, as described in [50], but have only been improved at the margin.

During this work, we tried however to find a convenient setup to access the register file in a managed environment like the Java Virtual Machine. Java-based applications have the great advantage to be faster to develop than low-level C/C++ applications, while offering instant portability, which makes them a good candidate for all software developments which don't have strong requirements in terms of performance predictability and fancy architecture support.

There are two main issues to solve to create a complete interface set:

- ✓ First we must be able to bind with the device, whose driver interfaces are only available as platform specific shared libraries or through system calls.
- ✓ Secondly, a programming interface must be offered in the host language for the user to write an application.

4.1.5.1 Simple device interfacing using mmap

In the context of Extoll hardware, multiple options are available to access register files of various hosts using the special functional units offered by the network controller. If we only focus on accessing the register file of a PCI device plugged in the local host, the architecture becomes quite simple.

The software view on the register file is a simple continuous address space, where it issues read and writes (see Figure 4-3). Therefore, the device driver only needs to map the memory region matching the size of the register file up to the user space. A mapped memory region is simply obtained in C as a pointer which can be manipulated using the + operator to navigate into the memory area to the location which match the desired elements.

The memory mapping for a register file in a PCIe device is presented in Figure 4-16. From the hardware to the user space, the memory mapping stages are:

1. The kernel driver maps inside the kernel space the I/O memory region from the PCIe BAR which matches the register file address space. The location of this region inside the BAR is known to the driver based on the specification of the PCIe interface inside the hardware (hidden from picture).
2. A character device driver is loaded inside the Linux kernel. It can access the register file memory region mapped by the kernel driver, and exposes it through a character device, which is presented to the user space as a file.

3. A user application can open the character device file, and request to map its content to the application's virtual address space. This operation is called *mmap*.
4. The character device driver maps the kernel space memory region to the user application virtual address space.
5. The application receives a pointer to a memory region as return value to the *mmap* call. It can then navigate in this memory region and issue read and writes by using the = operator.
6. Steps 3 to 5 can be repeated by different application instances.

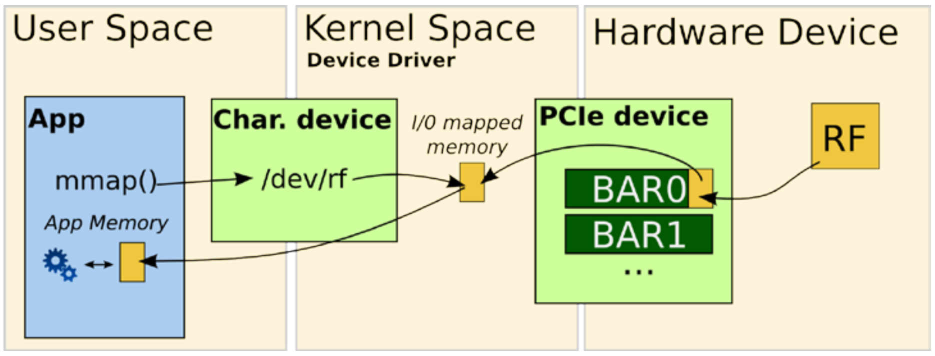


Figure 4-16 Register file address space mapping

4.1.5.2 Native function binding in the Java Application Space

We have shown a register file can be mapped into a user space application using a simple *mmap* call. The same can be done from within an application running inside a JVM. However, the system calls like *mmap()* are not available as standard Java calls, therefore the user must make the C code accessible to a Java class. This is possible using the Java Native Interface (JNI), which allows the implementation of some methods marked using the *native* keyword, to be located in native code available as a shared library.

Using the JNI interface is not very complex, but requires the native functions to respect a special API contract. A faster method was discovered by using the BridJ [52] library, which hides the JNI API from the user, by automatically mapping standard C function definitions to Java Classes. Figure 4-17 summarises the two main components of the BridJ library:

1. At compile time, it parses C header files and generates a Java class containing the appropriated native method definitions. The user then only needs to compile the standard C code in a shared library.
2. At runtime, BridJ loads the previously compiled shared library, and connects its function definitions to the Java native methods.

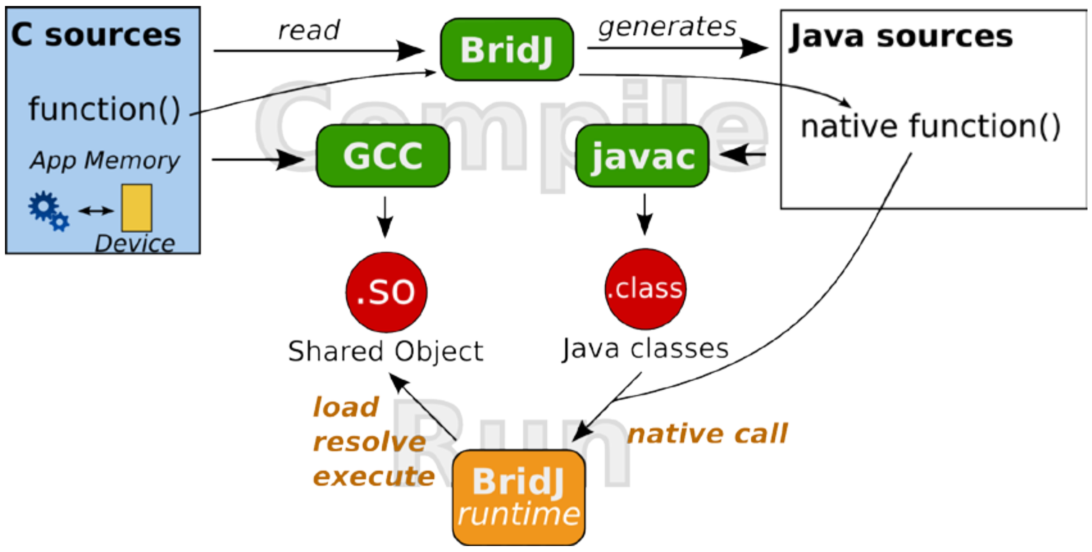


Figure 4-17 Automatic native code binding using BridJ

Using the BridJ library, a very simple piece of C code can thus be written to expose the register file memory region to the Java application.

4.1.5.3 Scala API for RFG

Once the device is accessible to be issued read-writes, we need to create a programming interface for the Scala language. This interface could be generated, with classes and method definitions for all hierarchies, but we decided to create a generic interface that would initialise itself by reading the XML output.

For this purpose, we used the OOXOO binding library, whose functionality is presented in 4.2.7.1, along with the created generic Register file interface in 4.4.4.

4.2 Hierarchical floorplanning for Integrated Circuits

Once a digital hardware design has been specified using an HDL language like Verilog or VHDL, with the help of specific tools like the previously presented register file generator, it can be mapped to a real physical circuit. This process is automated and follows various steps, as presented in Figure 4-18, from code to logic function translation, down to transistor wiring.

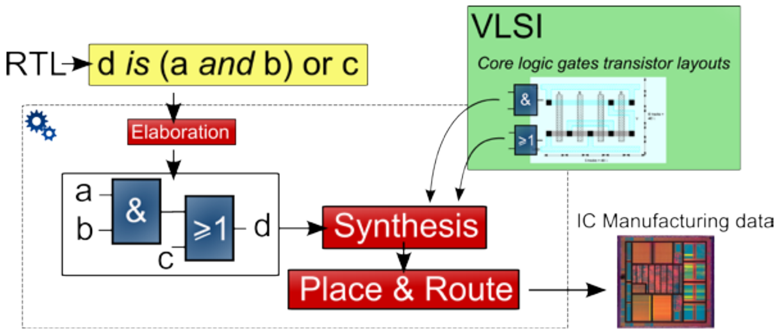


Figure 4-18 Digital circuit HDL technology mapping overview

During this process, the tools permanently check that the resulting circuit implementation respects the user constraints, especially in term of reachable clock frequency for the synchronous logic. More specifically, during the Place and Route phase, the software component called placer must determine the physical location of all the design elements. Two main categories of such elements must be distinguished:

- ✓ Special Resources like RAM memories, Phased-Locked-Loops, input/output Serialisers/Deserialisers etc...
- ✓ Logic gates

The logic gates typically have a size in the order of magnitude of a few transistors, and are dispatched on the available circuit area. The special resources however are usually bigger in size, and are therefore called macro blocks, hard macros, or macros for short. Together with the external inputs and outputs, the macros structure the logic gates placement depending on their locations.

Figure 4-19 illustrates this fact by showing how the logic of a circuit featuring RAM memory macros would be dispatched, depending on the location and orientation of both RAMs and input/outputs. Three possible options were drawn to show various considerations during placement:

- ✓ Option A: Some area could be wasted in this configuration because of a over-usage.
- ✓ Option B: Some area could also be wasted because the macros are close to each other, and the gap between them may not be usable for any logic.
- ✓ Option C: Here the placement and orientation of the macros is sub-optimal.

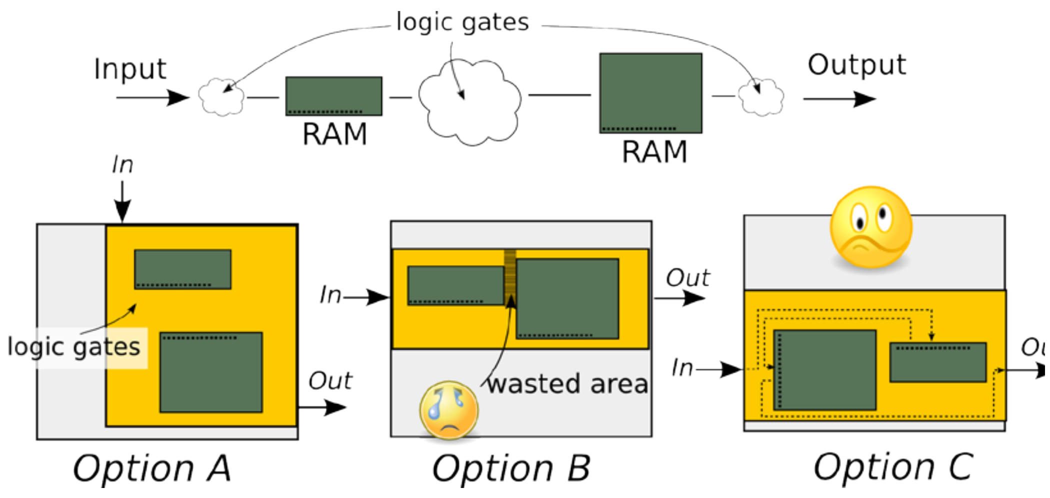


Figure 4-19 RAM macro block placement options

For small designs with relaxed constraints, the placer can find a solution on its own. When the design size scales up, the complexity of the task can quickly grow out of acceptable bounds, because moving around macros impacts logic placement, and logic placement may lead to moving around macros…

For large designs, a user defined process called floorplanning must be run. Floorplanning, as illustrated in Figure 4-20, consists in feeding the software tool with instructions to pre-set the physical location of part or all the structuring elements of the circuit: Input/Outputs, macros locations, bus guides to force set of wires to be routed in a specific area etc…

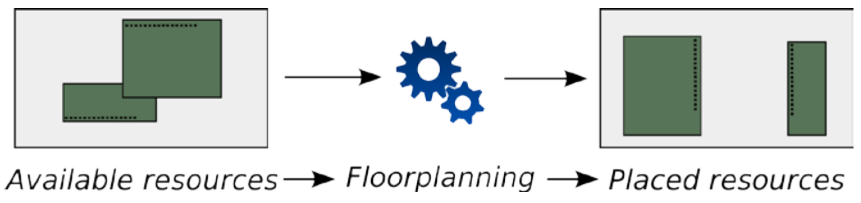


Figure 4-20 Floorplanning places resources

Generally, when implementing a design in a circuit, two main target technologies categories are available:

1. Field Programmable Gate Arrays (FPGA), which are integrated circuits featuring a vast amount of ready-to-use logic gates and macro resources. An FPGA mapping process translates a circuit to an array of bit, which once loaded in the device reconfigure it internally to wire the desired circuits.

2. Application Specific Integrated Circuits (ASIC), which are full-custom manufactured integrated circuits. The mapping software must be configured with all circuit constraints: available resources and logic gates definitions (bought from specialized vendors) for a specific manufacturing process (example: 180nm IBM, 65nm TSCM, 28nm STM etc…), die size, input output types etc… The die area is free to use at whish to place and wire the circuit.

In this section, we are going to describe a programming interface designed to help floorplanning a design for an ASIC production. FPGA floorplanning is a specific case, and usually requires little configuration, because the resources are already placed somewhere in the integrated circuit.

4.2.1 Hierarchy-centric macro placement

The Extoll network ASIC implementation, whose architecture was briefly presented in introduction to this chapter, represents a case for a large design. It features hundreds of RAM memory macros, and a total of roughly 279 Million transistors. To implement such a design, it is necessary to partition it. In other words, some of the sub-designs in the hierarchy can be implemented concurrently, and connected together as part of an abstracter higher hierarchy level. The partitions are then seen as huge macro circuits which can just be connected together.

In Figure 4-21, we can see a view of the Extoll ASIC partition macros with their interconnection scheme. On the bottom, a screenshot of the physical layout of *extoll_asic_top* is shown, where each of the partitions are represented as macro areas. They are placed on the circuit die, and wired with each other, but the detailed circuit present inside each of the macros is hidden.

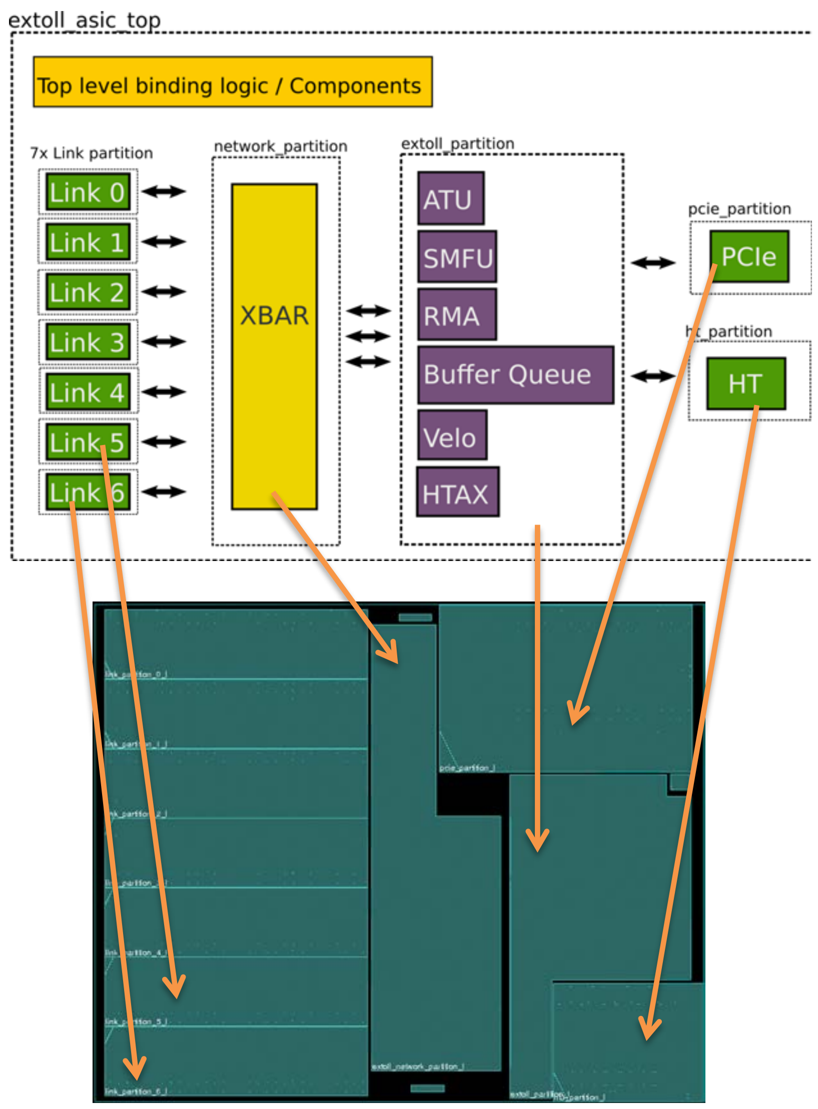


Figure 4-21 Extoll ASIC Toplevel partitions and die placement

Each of those partitions, which also contain some sub-hierarchies, are small enough to be implemented as standalone circuits, without facing overwhelming tool runtimes.

Generally speaking, no matter which level we look at, we always face a hierarchy tree, whose nodes represent distinct sub designs (or logical groups) which are connected with each other, sequentially or in parallel (with some synchronisation signals or not) and this down to the lowest level. In Figure 4-22, such a hierarchy tree is represented, in which two kinds of nodes can be seen:

- ✓ Hierarchy nodes which contain sub-hierarchies, logic gates and macros.
- ✓ Macro nodes, which are abstract blocks and thus terminal.

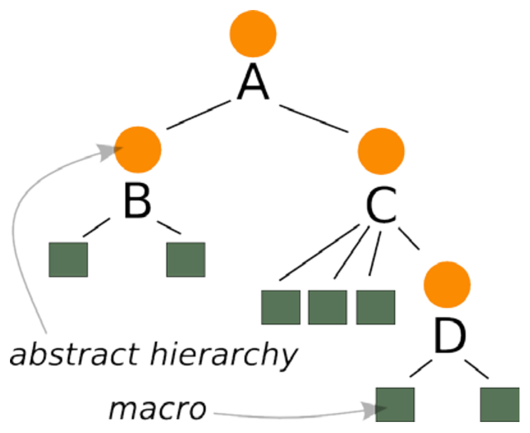


Figure 4-22 Hierarchy tree with sub-hierarchies and macros

This hierarchical structured view on the design is the one that is defined in specification documents, and also the one present in the architects' minds.

During floorplanning, the goal is to try to find a placement configuration for all the macros, which correctly maps to the logic structure, and can be optimised based on physical constraints (like macro sizes). Sub-hierarchies, although they are not single discrete components, may also be related to each other. This means that during floorplanning, the macros have to be placed based on constraints inside their local hierarchy, but also in respect to their parent hierarchy's logical placement.

The global placement of all objects is thus clearly a top-down process, whilst the placement of the single objects is a bottom-up process. Figure 4-23 shows a prepared bottom level for hierarchy B, C, D, and two options for level A. For each B, C and D leaves, we can see that the logical placement of the nodes was defined locally, independently from the parent hierarchy, and then reordered in the global context.

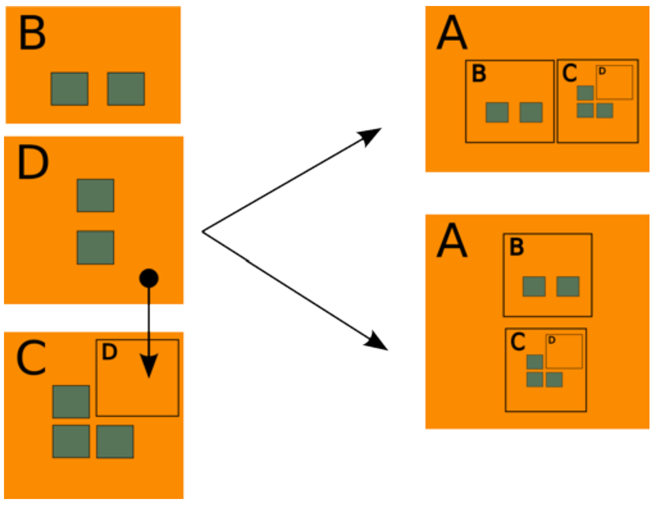


Figure 4-23 Placement options for a hierarchy tree

This placement strategy can be described as being hierarchy relative, and follows two main steps:

1. Floorplanning of all hierarchy levels based on their first-level content, each in its own local coordinate space.
2. Resolve the absolute coordinates of all objects when the full top-down hierarchy is known.

Figure 4-24 shows the placement coordinates of a macro in C, and of C inside A. Both coordinate spaces for C and A are independent. The table that follows describes an example of absolute coordinate resolution when top-down placement is run.

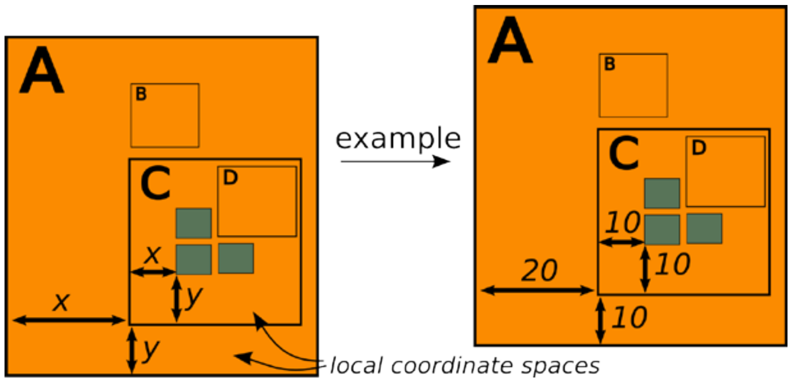


Figure 4-24 Hierarchies floorplanning in separate coordinate spaces

Hierarchy	Local Coordinates	Absolute Coordinates
A	(0,0)	**(0,0)**
C (in A)	(20,10)	**(20,10)**
C	(0,0)	**(20,10)**
MACRO (in C)	(10,10)	**(30,20)**

4.2.2 Motivation for a generic programming interface

Concretely, the previously described floorplanning process is driven by an input passed to the software environment. This input is typically a TCL script, which issues API calls provided by the tool environment to place objects.

All tools have their own set of programming interfaces, with more or less adequate functionalities to create groups of objects and help define placement. Sometimes those helper functions are not very useful, and make working with the software environment difficult and time consuming. Moreover, the design must already be in a quite advanced stage, so that the environment sees all the final macros that will have to be placed.

Additionally, if multiple alternative design solutions are explored, they have to be brought far enough in the implementation process so that floorplanning can be done.

Based on the hierarchical placement concept we just presented, we can say that there are not reasons for a floorplanning process to be concealed to the programming interface of any specific vendor software. The placement of objects can be specified in an abstract way, and then outputted to the target environment by mean of a configuration file, direct API calls or anything else required.

In 4.2.3 we present a novel, generic and environment-agnostic programming interface for objects floorplanning. The implementation we propose targets the TCL language, and relies on the EDSL design rules presented in 3.3.

4.2.3 A scene graph programming interface

We introduced in 4.2.1 the basic requirements of floorplanning. We can now reformulate the concept using a more generic semantic, by saying that we are trying to model and layout a hierarchy of objects in a two dimensional coordinates space.

Scene graphs are a well known modelling concept often used for the rendering of 3D scenes, as described for example in [53] or [54]. The name “scene graph” does not refer to any specific programming standard, or existing data representation specification, but more generally to the idea of using a tree of nodes to model an object hierarchy. Parent nodes are container for children nodes and properties applied to those containers can in turn affect the children. Applied to a generic scene representation as presented in in Figure 4-25, hiding a hierarchy level would for example mean hiding all the children.

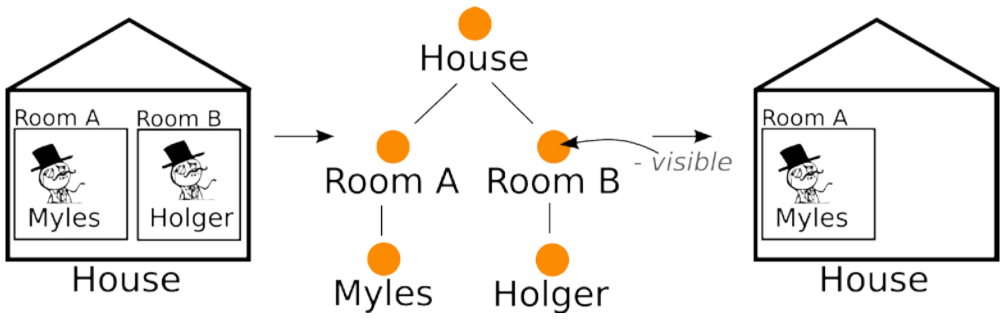


Figure 4-25 3D Scene representation using a scene graph

The similarities between common scene-graph applications and the floorplanning problem presented in 4.2.1 lead us to borrowing the semantic of scene graphs to develop our programming interface. The software architecture follows three main axes, which we are going to be detailed in the next sub-sections:

1. The properties of each node, as required for floorplanning.
2. The scene graph tree nodes definition.
3. The integration of the scene graph programming interface into specific target environments.

4.2.3.1 Floorplanning properties requirements

Each node in the scene graph must have a set of properties to enable object placement calculation. The first two natural properties which come to mind are position and size (width and height) of the nodes. This is true for any kind of nodes, may they be pure container (hierarchy levels) or macro blocks. The size of a pure container will vary depending on its contents’ positions and sizes, while the size of a macro is an immutable constant.

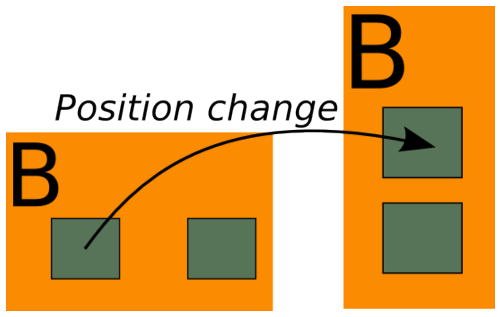


Figure 4-26 Pure container B shape variation based on its content

Figure 4-26 shows how the shape of the container B (from Figure 4-23) changes to reflect a modification in its content's positioning.

The third property is the orientation for macro blocks. Indeed, they have input and output connections spread on their sides, and thus must be oriented during floorplanning to be adapted to the orientation of their neighbours. For containers, the orientation is not considered relevant, as it would require moving the content around.

Depending on the target application, the supported orientation values applicable to a macro may vary. The scene graph programming interface however defines a set of standard values:

- ✓ R0: The standard macro shape (no rotation)
- ✓ R90: A rotation of 90 degrees counter-clock-wise
- ✓ R180: A rotation of 180 degrees counter-clock-wise
- ✓ R270: A rotation of 270 degrees counter-clock-wise
- ✓ MX: Mirror over the X axe
- ✓ MX90: MX then R90
- ✓ MY: Mirror over the X axe
- ✓ MY90: MY then R90

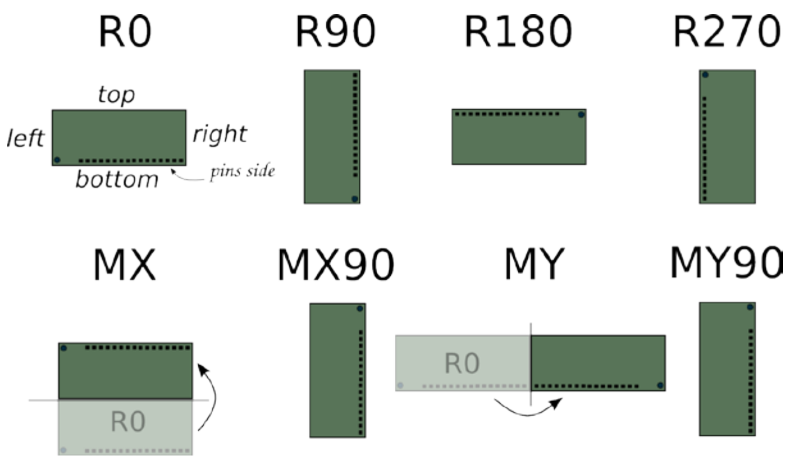


Figure 4-27 Orientation options for macros

4.2.3.2 *Abstract API in TCL*

The scene graph programming in TCL was designed as an embedded language for the TCL language, as presented in 3.3. It supports the requirements we have defined so far:

- ✓ Scene graph construction: Create a tree of nodes in memory
- ✓ Floorplanning properties: Nodes must define position (x,y), size (width and height) and orientation.
- ✓ Coordinate space resolution: Nodes have an Absolute position property, which is resolved by the scene graph

4.2.3.2.1 Abstract class hierarchy

The base domain specific language for this API is limited. The reason is that we are focusing here on defining generic object placement behaviour. The macros, which are the physical elements in the scene graph, have only been described in an abstract way so far.

We cannot at the moment define their actual implementation in a class hierarchy, because their definition must be provided by an interface layer to the application which is targeted. All what can be said is that they are Nodes in the tree.

This is why the final user-ready domain specific language is partially defined here for the generic scene graph behaviour, and partially at the application layer level. A simple example is presented in 4.2.3.3.1.

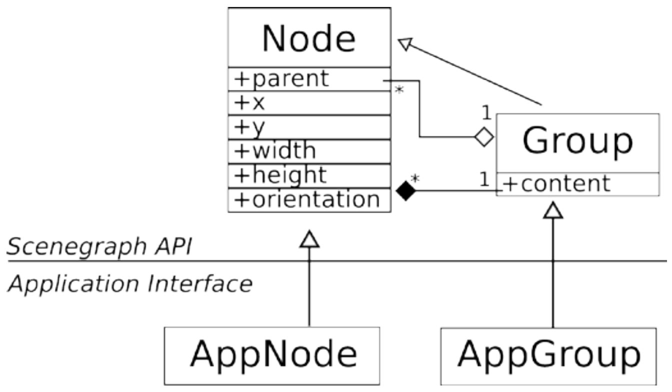


Figure 4-28 Base scene graph class hierarchy

Figure 4-28 represents the base scene graph *Node* class, and the container node type called *Group*. A *Group* owns a relation to its content, and a *Node* to its parent container. The application interface layer shows an inheritance example that could be provided by an application-specific programming interface based on the generic scene graph API.

The source code is a good reference for the implementation details, concrete application are described later.

Repository: odfi-dev-tcl-scenegraph , Path: tcl/scenegraph/scenegraph-1.0.0.tm

4.2.3.2.2 Container shape and orientation

The nodes representing macros have an orientation parameter. Additionally, the container nodes derive their shape from their first-level content. Depending on their orientation, the macro nodes have thus a different shape in the same coordinate system.

To avoid forcing the container nodes to explicitly keep track of their content's orientation, which would increase the complexity of the implementation, it has been decided to let the Nodes return a size which depends on the orientation.

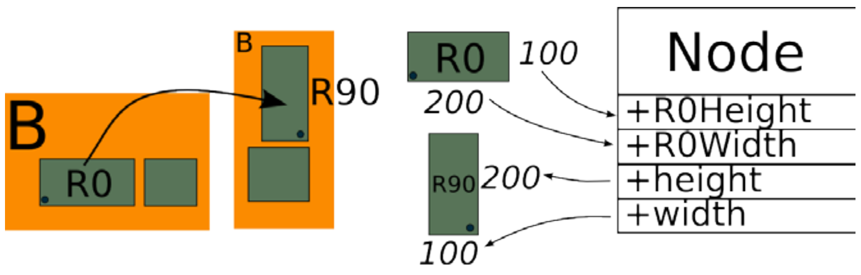


Figure 4-29 Size and orientation methods contract

In Figure 4-29, the *Node* instance representing the macro oriented to R90 returns a view of its shape by inverting its initial width and height. The default contract for the *Node* class therefore specifies two special methods called *R0Width* and *R0Height*, and automatically adapts the returned *width* and *height* based on the orientation.

4.2.3.2.3 Absolute coordinates resolution

Finally, resolving the absolute coordinates of objects is a simple operation which is implemented using the bottom-up recursive function $position() \ = position(parent) + \{x, y\}$.

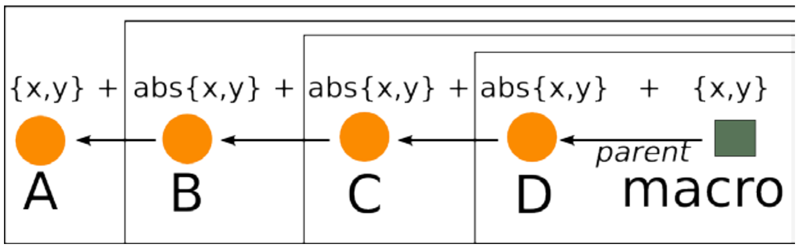


Figure 4-30 Recursive absolute position resolution

Figure 4-30 illustrates the successive absolute position resolution for one of the macro under the D hierarchy level from Figure 4-22. The $abs\{x, y\}$ expression refers to the absolute coordinate of the nodes, which is obtained by a recursive call issued from the child node. Both the *macro* and the *A* containers have a predefined position: The *macro* element because it is positioned in the relative coordinate space of its parent, and *A* for it is the top container and thus does not have a parent. It is the origin.

To improve performances, the absolute position is cached for each node. Care must be taken by the user or a specialized application layer to invalidate the value of the caches absolute position if a change in local position or shape occurs. For example, changing the position of a container node must lead to invalidating the absolute positions of the complete sub-tree.

4.2.3.3 Application interface

The scene graph API is application-agnostic, as mentioned in 4.2.3.2. This means that no input-output connection to produce any result is provided. We have indeed so far only presented the base functions to place objects in a coordinate space, but not how to actually floorplan any macro on an integrated circuit.

An application interface layer is required to construct node instances which can be floorplanned using the scene graph API, and commit the results to the application. Figure 4-31 presents the general concept. This part of the application's design only consists in defining subclasses of *Node* and *Group*, if needed, and implement application-specific behaviour. It is therefore purely classical software design.

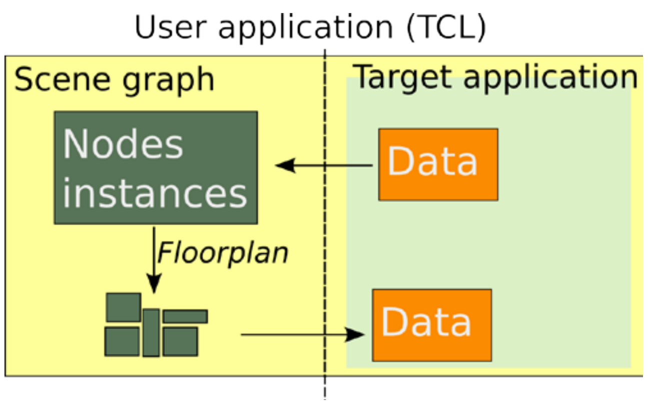


Figure 4-31 Application interface and scene graph interfacing

4.2.3.3.1 Example: Floorplan prototyping using Library Exchange Format files

Now that we have covered the main aspects of the floorplanning programming interface, we can illustrate our purpose with a small example, which reproduces the application flow of Figure 4-31. The source code with annotated flow is presented in Figure 4-32 and features four major steps:

Input Data

First, a macro definition is going to be loaded from a Library Exchange Format (LEF) file. LEF files are industry standard text files which hold layout information about physical components. The LEF file we are using contains a named macro definition, associated with its R0 width and height. The size of the macro is *333.42x 105.28* microns.

Create application data

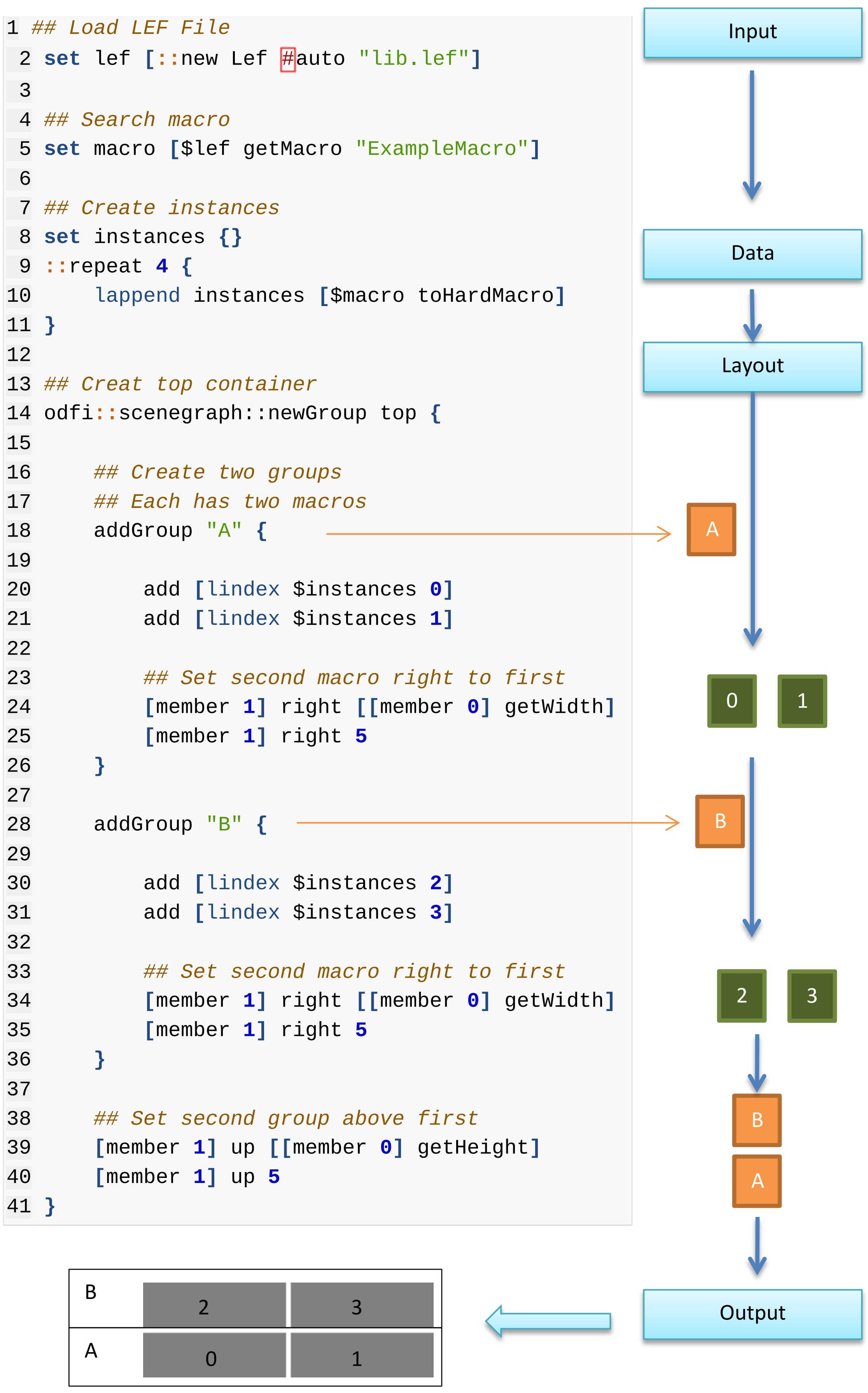
Secondly, a real application is simulated by creating 4 instances of a *Node* class which represent instances of macros matching the description extracted from the LEF file. The class name is *HardMacro.*

Layout

The node set that was created before can be floorplanned. To illustrate how containers work, the fours macros are split into two groups. Both groups are placed on top of each other, and each macro next to each other in their respective groups. Some spacing is added when positioning for output clarity.

Output

As we are not really floorplanning any real integrated circuit, we can just output a scalable vector graphic XML file, which will mirror the results.



```
## Load LEF File
set lef [::new Lef #auto "lib.lef"]

## Search macro
set macro [$lef getMacro "ExampleMacro"]

## Create instances
set instances {}
::repeat 4 {
    lappend instances [$macro toHardMacro]
}

## Creat top container
odfi::scenegraph::newGroup top {

    ## Create two groups
    ## Each has two macros
    addGroup "A" {

        add [lindex $instances 0]
        add [lindex $instances 1]

        ## Set second macro right to first
        [member 1] right [[member 0] getWidth]
        [member 1] right 5
    }

    addGroup "B" {

        add [lindex $instances 2]
        add [lindex $instances 3]

        ## Set second macro right to first
        [member 1] right [[member 0] getWidth]
        [member 1] right 5
    }

    ## Set second group above first
    [member 1] up [[member 0] getHeight]
    [member 1] up 5
}
```

Figure 4-32 Simple Floorplanning application

To close the example, we can have a look at the output generation code to show the usage of the automatic absolute coordinates calculation.

```
1 ## Output SVG
 2 set svg "<svg
 3         xmlns=\"http://www.w3.org/2000/svg\"
 4         version=\"1.1\"
 5         width=\"[top getWidth]\"
 6         height=\"[top getHeight]\">"
 7
 8 ## Only output macros
 9 top eachRecursive {
10     if {[$it isa HardMacro]} {
11         set svg "$svg
12         <rect x=\"[$it getAbsoluteX]\"
13             y=\"[$it getAbsoluteY]\"
14             width=\"[$it getWidth]\"
15             height=\"[$it getHeight]\"
16             fill=\"gray\"/>"
17     }
18 }
19
20 ## Write
21 set svg "$svg</svg>"
22 odfi::files::writeToFile "lef-example.svg" $svg
```

Figure 4-33 Example application SVG output

In Figure 4-33, we can see how the automatic size and absolute positions were derived from the relative floorplanning of each hierarchy level in the previous step:

1. The main *<svg>* element is sized using the top container ***getWidth*** and ***getHeight*** methods, which return a size based on the content layout.
2. The groups are not considered (filtered by line 10) because we are interested in the final placement of the macros.
3. The *<rect>* elements which represent the macros must be placed on the picture at their exact global location. Therefore the ***getAbsoluteX*** and ***getAbsoluteY*** methods are used.

To sum-up, we have presented here a simple methodology to prototype floorplanning of macros. By using an adequate application interface, it is possible to keep the same layout source code, and replace the input/output parts to interface with Integrated Circuit implementation software.

4.2.4 Generic building blocks for floorplanning

Repository: odfi-dev-tcl-scenegraph ,
Path: tcl/scenegraph/layout-functions-1.0.0.tm

The presented floorplanning programming interface is generic for all applications. This means that it is possible to write some functions to layout objects in a certain way (row, column, grid etc…), and reuse them.

To do so, there are not special rules to follow and the user is free to define its own methodology. However a small set of functions have been introduced in the core API to simplify this process. It features two aspects:

1. Defining a layout function: A name and an implementation must be provided. The API passes a ***$group*** reference to the container which must be processed, and a ***$contraints*** reference to a value map containing some parameters (like spacing, or special function features selectors).
2. Calling a layout function: The user can use the "layout name constraints" method available on the *Group* class and pass it a name matching a previously defined layout function, along with a values list as constraints.

Figure 4-34 schematises a layout function use for the floorplanning example from Figure 4-32 and uses a Grid layout function to structure the four macros. Two constraints are passed:

1. The *rows* constraint orders objects on two rows. The function creates as many columns as required (our example is a corner case where two columns could also have been requested for the same result).
2. The *spacing* constraint adds spacing in all dimensions by adapting the object's positions.

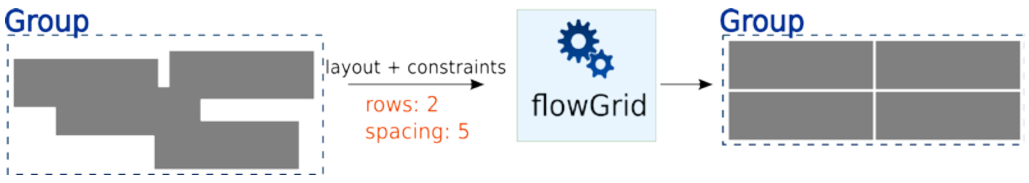
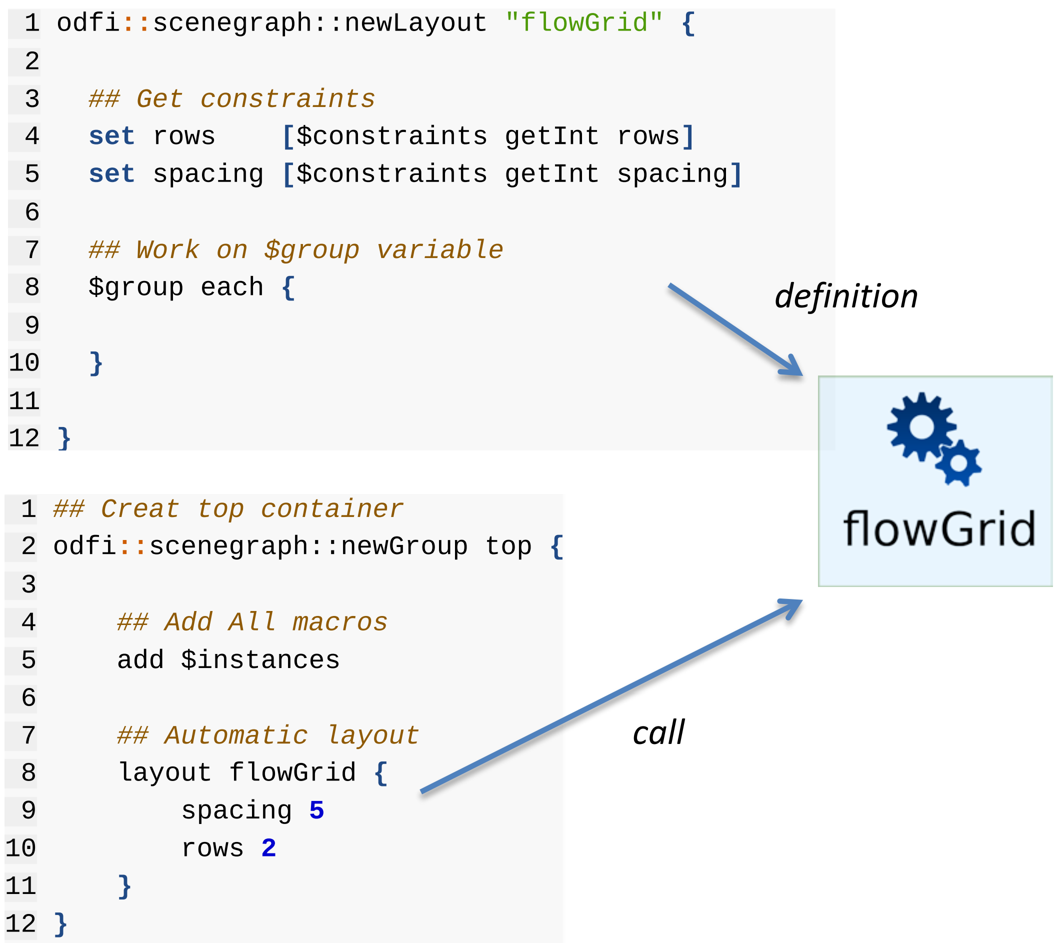


Figure 4-34 Objects floorplanning using a layout function

Finally, the definition and usage of layout functions is illustrated in Figure 4-35 which shows a part of the code from 4.2.3.3.1 adapted to just add the macros to the top-level group and call the *flowGrid* function. Some more layout functions like *column*, *row*, *mirrorX*, *mirrorY* are available and can be consulted in the library's project.



```
odfi::scenegraph::newLayout "flowGrid" {

  ## Get constraints
  set rows    [$constraints getInt rows]
  set spacing [$constraints getInt spacing]

  ## Work on $group variable
  $group each {

  }

}
```

```
## Creat top container
odfi::scenegraph::newGroup top {

    ## Add All macros
    add $instances

    ## Automatic layout
    layout flowGrid {
        spacing 5
        rows 2
    }
}
```

Figure 4-35 Definition and Usage of layout functions

4.2.5 Generic data representation using SVG

SVG pictures are an example of an application candidate to be integrated with the scene graph API. SVG was previously used in this work to produce graphical outputs, quite often by producing the XML manually like in Figure 4-33.

A basic SVG language was presented in 3.3, which was integrated in the scene graph API class hierarchy. The layout functions presented in 4.2.4 can thus be used to place graphical elements on the picture.

The scene graph integration details can be consulted in the source code, but to illustrate the API usage we can refine once again the macros floorplanning example from 4.2.3.3.1. Figure 4-36 presents new the SVG picture production from Figure 4-33 converted to the SVG language usage.

```
## Output SVG
odfi::scenegraph::svg::createSvg svgview is {

  ## Only output macros
  ::top eachRecursive {

      if {[$it isa HardMacro]} {

        addRect {
          setX [$it getAbsoluteX]
          setY [$it getAbsoluteY]
          width [$it getWidth]
          height [$it getHeight]
          fill gray
        }
      }
  }

}
```

Figure 4-36 Simple SVG scene graph integration example

4.2.6 Real placement in Cadence Encounter

The Cadence Encounter tool is an Integrated Circuit physical implementation software environment for digital designs (Figure 4-18 Place and Route). Floorplanning of ICs can be done using this application.

The presented library was used in Encounter to perform floorplanning of the Extoll's IC partitions. Not only SRAM memory macros were placed, but also other blocks required for the top-level integration like:

1. Area I/O cells groups in I/O Rows.
2. Power I/O cells.
3. ElectroStatic Discharge (ESD) protection cells, etc…

Some more structures could be supported, but given the size of the Extoll IC, these already represent multiple hundreds of structures across the various partitions, the interface was not used further.

To give an example, Figure 4-37 shows the RAM placement of the Extoll Crossbar from Figure 4-21. The crossbar contains a certain number of {Input + Output} port tuples (eleven in the Extoll ASIC) connected together through an arbiter. All the structures are identical, only repeated according to the number of IO port the crossbar features. The floorplanning for the crossbar is easy to implement using the scene graph API, and features two steps:

1. Create a function that performs the floorplanning of an input port + output port group. This can be prototyped and multiple alternatives can be tested.
2. Create the Top level crossbar Floorplan group, which creates the input + output port groups, pass them to the floorplanning function, and then places the groups on the partition area.

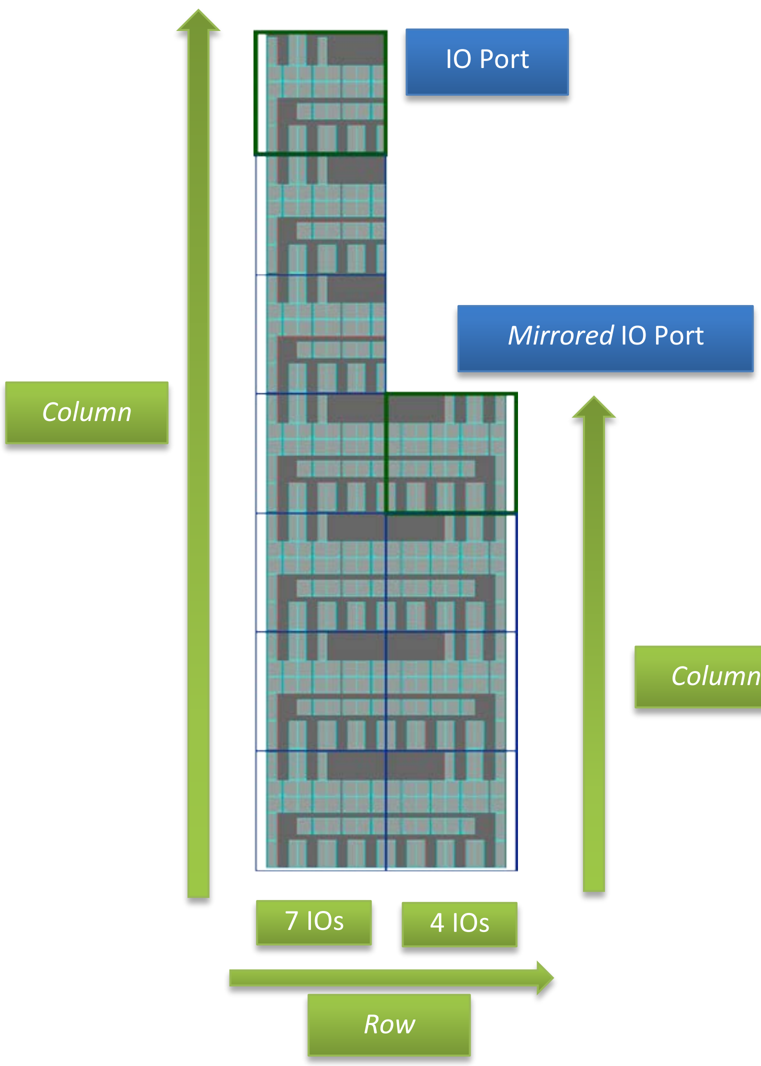


Figure 4-37 Extoll crossbar floorplanning example

Additionally, we can note that the defined *Column*, *Row* and *Mirror* floorplanning functions can be defined in a generic way. The four right and IO groups were mirrored because they are structured the same way as the ones on the left, but the external pins are on the right. We could also have written the IO group Floorplan respective to the right side, and applied the mirror function to the left ports.

4.2.6.1 Application interface

The Cadence Encounter application interface exactly follows the description provided in 4.2.3.3. It provides two main interfaces:

- ✓ A set of *Node* and *Group* subclasses providing interfacing to the various object types encountered. For example, the *EncounterFloorplan* class inherits the Group generic node type, and returns as *R0Width* and *R0Height* the size of the die configured in the Encounter design.
- ✓ A special *applyToEncounter* function which walks the scene graph tree and places the physical elements at their absolute locations.

Additionally, the floorplanning library and its dependencies must be loaded in the Encounter's TCL interpreter. Some details are provided in Appendix A.

4.2.7 Outlook

In this section we presented a generic 2D scene layout programming interface. It offers the user the advantage of hierarchy-level relative placement for objects, which enables coordinates calculation to be independent from the final absolute coordinate space. The absolute coordinates are automatically resolved only when required.

Two main applications were found so far: Floorplanning of macro blocks in ASIC designs and SVG drawing. Both can be used in the context of digital hardware design, either for real implementation, or for prototyping purpose.

However, floorplanning prototyping using an SVG picture is only convenient if the digital design input (Verilog or VDHL format for example) can be parsed or read to feed the prototyping environment with correct information about macro blocks. This was not the case in the context of the Extoll project, so SVG was only used in very early prototyping phase in the same way as presented in 4.2.3.3.1.

4.2.7.1 Multiple tree-view support

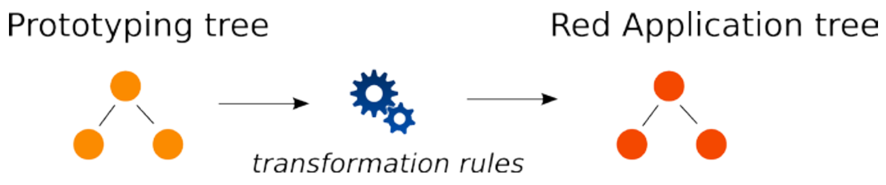
By looking at the macros floorplanning prototyping example from 4.2.3.3.1, it is interesting to note that we are dealing with two trees representing the same data. Indeed, the macros floorplanning was performed, and then an SVG tree was produced by merely only translating *HardMacro* objects to SVG *Rect* ones.

This example shows that it would be interesting to develop a methodology to improve the integration of multiple application views inside the scene graph API. In our case, the two application views could be for example:

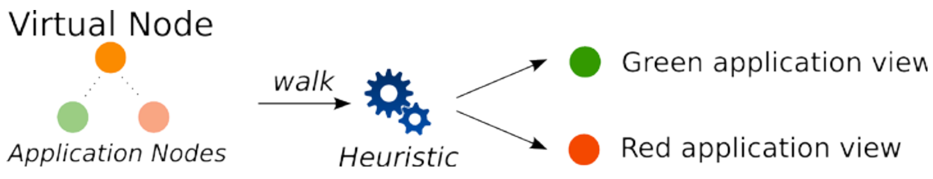
- ✓ An SVG view to be used when prototyping
- ✓ An Encounter node view to be used when committing the macros tree into a real Physical Implementation software

This issue can be solved in multiple ways, which might also depend on the programming language capabilities. We through about three different approaches which could be explored and even combined:

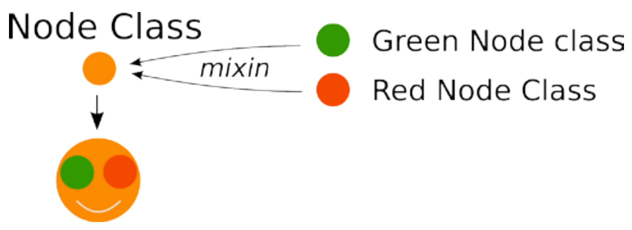
- ✓ Tree transformation: XML transformation technologies like XSLT [51] define the concept of tree-to-tree transformations. A usage example was presented in 4.1.3 to convert an XML document to a TCL EDSL script. A similar programing interface could be developed to define transformation scenarios. This kind of transformation can be very generic and does not deeply interacts with the source tree, but yields separate tree instances, which can bring confusion to the developer.



- ✓ Virtual Nodes: A “virtual node” would be a simple generic node, which would hold a set of application nodes without being a group. The generic scene graph interface, like placement and orientation would be forwarded to the contained node, so that floorplanning can be done in a classical way. A specific application could then walk the tree and not see the virtual nodes, but one of their contained application node based on a selection heuristic.



- ✓ Class mixins: Section 3.3.2 introduced the concept of class mixins to add functionality to classes outside of their initial definition. This could also be a way for applications to add their required properties and methods to the *Node* or *Group* classes.



4.3 Part description language

Once an integrated circuit has been manufactured, it is packaged to be physically soldered on a printed circuit board (PCB). This packaged circuit is usually called a Part and is at the boundary between the internal IC and the external world. A part description should, for each input or output pin, include various information, which are used by various tools like PCB design, on board signal trace analyses or documentation generator. Examples of such useful data are: physical pin location on the package, logical grouping of pins, input or output, buffer strength and type for a pin etc…

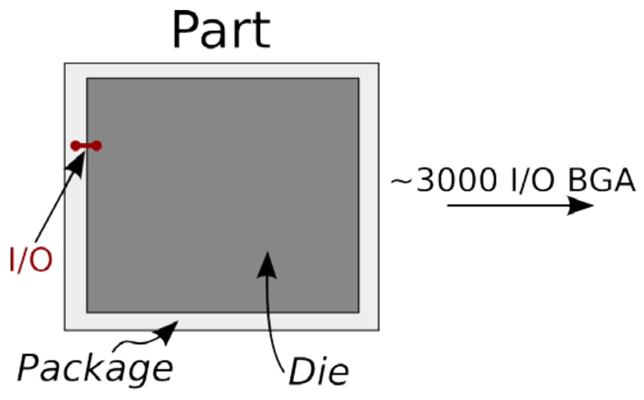


Figure 4-38 Part definition with Ball Grid Array (BGA) view

There are a lot of integrated circuit vendors, as well as tools for PCB design and all kind of analysis, but no flexible solution is available to access the raw data describing a package. Some vendors have their own internal features to ease part definition importing in their software chain, but it is usually limited to the set of provided circuits, and not really extensible. In the end, the designers often end-up reading a datasheet and importing the required data per hand in the target software environment.

In this section, we present a novel part definition language for TCL, which follows the same definition methodology as the languages presented in 4.1 and 4.2. It integrates in the design flow methodology by allowing to fully specifying a part in a tool-agnostic format. A few use cases that have been encountered are introduced to illustrate the part definition reuse depending on the target application.

4.3.1 Language description

The part language is similar to the register file generator one. It is simpler and only consists in a Part class definition which contains some Pin definitions.

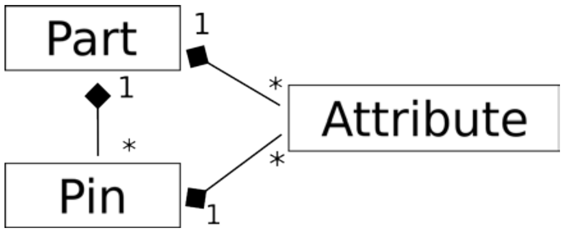


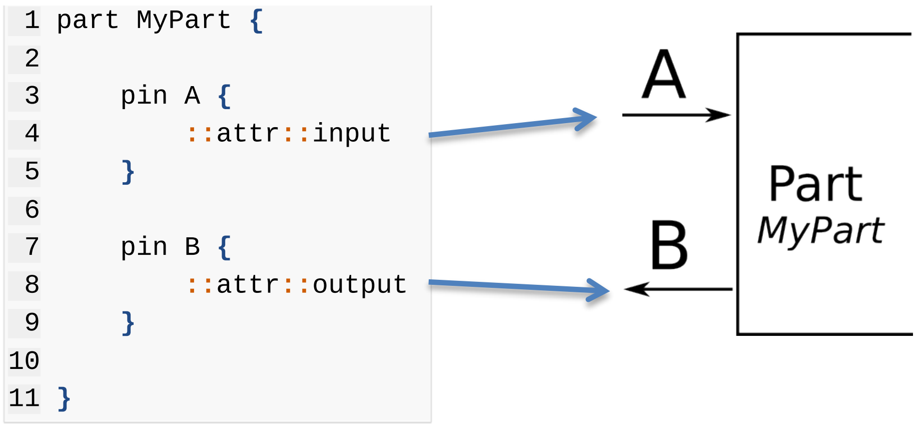
Figure 4-39 Part language class hierarchy

Pin definitions require a name and a location which has not already been set on another pin. This requirement was defined to avoid unnoticed wrong locations in the part file. The Location is defined using a JEDEC standard Ball Grid Array naming convention.

4.3.1.1 Attributes

To support various possible outputs or tool integration, the same strategy was adopted as for the register file generator (4.1). Most pin properties of are supported by attributes.

Repository: thesis, Path: sources/4.3-Package/part-io.tcl



```
part MyPart {

    pin A {
        ::attr::input
    }

    pin B {
        ::attr::output
    }

}
```

Figure 4-40 Input/Output Pin definition using attributes

4.3.1.2 *Abstraction level improvement example: Differential Pairs*

We introduced in 3.3.2.2.1 a methodology to add abstraction levels to an EDSL. The part language presents a nice example for this in the case of differential signal pairs or DiffPair for short. A DiffPair is a tuple of two pins which transmit or receive one signal using two opposite signals. They usually share the same name, with a character token appended to it, like "_N" or "_P", to differentiate between both positive and negative signals. Additionally, it is useful to annotate both pins as being part of a differential signal, along with a reference to the opposite pin name.

Figure 4-41 presents the usage of such a DiffPair object procedure, to create a DATA differential pair, while a CLOCK differential is created per hand.

Repository: thesis, Path: sources/4.3-Package/part-diffpair.tcl

```
odfi::dev::hw::package::part MyPart {

  diffPair DATA B1 B2 {
    ::attr::output
  }

  pin {CLK_P @A2} {
    ::attr::input
    ::attr::differential CLK_N
  }

  pin {CLK_N @A1} {
    ::attr::input
    ::attr::differential CLK_P
  }

}
```

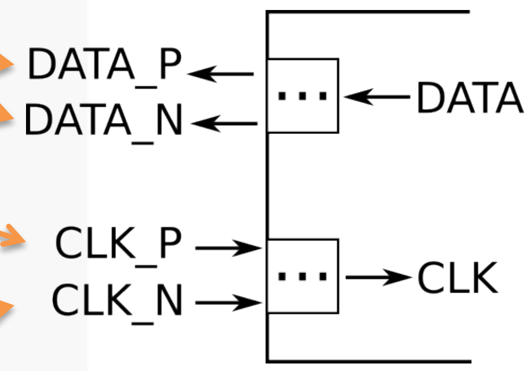


Figure 4-41 DiffPair abstraction level improvement

4.3.1.3 *Output generator rules*

The core programming interface defines a common *BaseOutputGenerator* class which should be inherited by software components which are producing specialised outputs from a part definition. This is not a compulsory requirement, but the *BaseOutputGenerator* interface is a common ground used by the tool to detect available generators.

Various parts may have various additional configuration parameters for output generators. It has been decided to delegate the generators' configurations to additional files, to allow a user to write multiple rule files for the generator, and select the desired one at runtime. A rule file holds on each line a parameter value for some pins. The lines thus follow the syntax:

$$REGEXP <,> NAME < space > VALUE$$

- ✓ REGEXP: Matches the pins to which the parameter must apply
- ✓ NAME: The parameter name
- ✓ VALUE: The parameter value

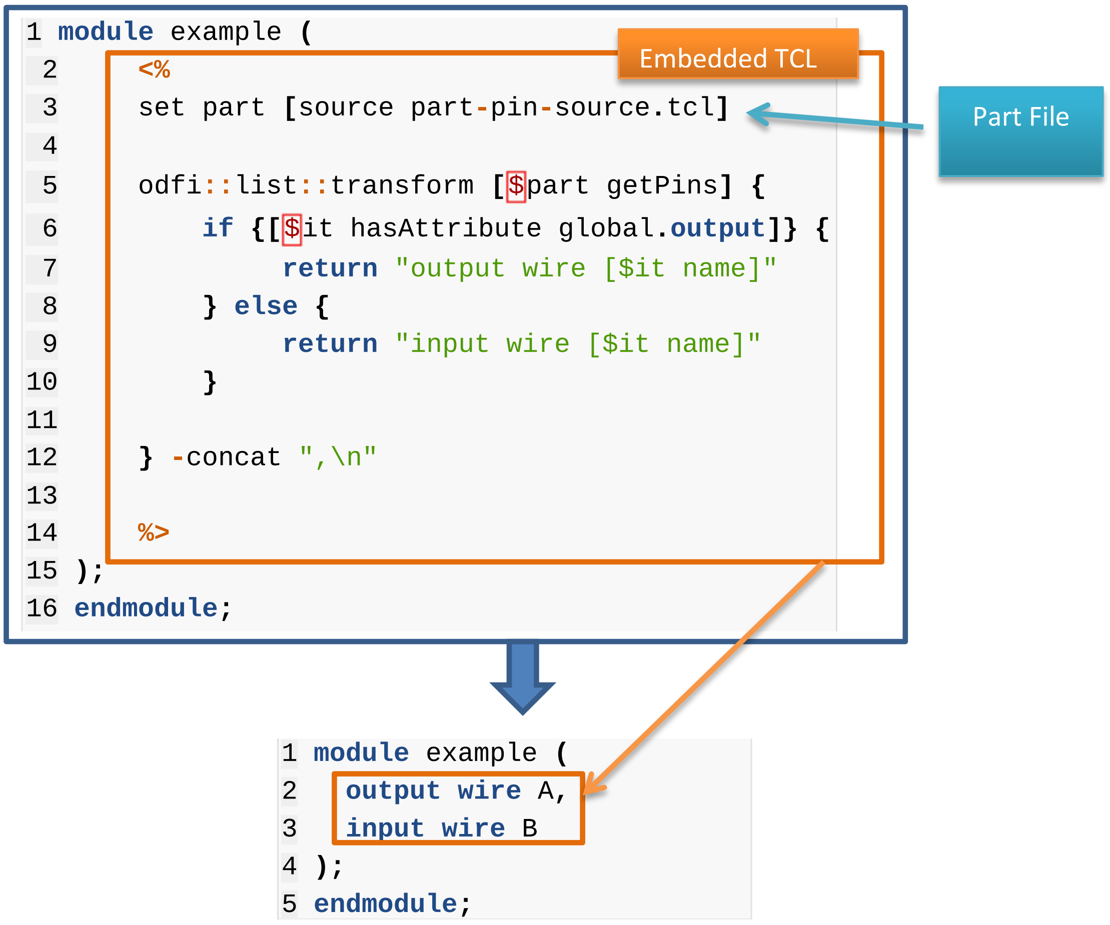
Some concrete examples are provided in 4.3.3.1 and 4.3.3.2.

4.3.2 Hardware description language (HDL) integration scenarios

⚠ Repository: thesis, Path: sources/4.3-Package/rtl-pin-import.v ; part-pin-source.tcl

The pins defined in a part definition are the same as the input and output wires defined on the top level hardware design entry language (HDL, Verilog or VHDL for example). It is thus a good idea to define them in only the HDL or the part file.

However, the part file can be used as an exchange format, so it is better to keep it consistent in one file, instead of delivering it with an HDL file to be parsed. Moreover, it can be relevant to generate the HDL input/output wires on the fly based on the part file data. The following example illustrates a way to better integrate part definition and HDL design language. Figure 4-42 presents a Verilog module source file whose input/output definition is generated on the fly by reading a part file.



```
module example (
    <%
    set part [source part-pin-source.tcl]

    odfi::list::transform [$part getPins] {
        if {[$it hasAttribute global.output]} {
            return "output wire [$it name]"
        } else {
            return "input wire [$it name]"
        }

    } -concat ",\n"

    %>
);
endmodule;
```

```
module example (
    output wire A,
    input wire B
);
endmodule;
```

Figure 4-42 Verilog embedded input output on-the-fly generation

The TCL code embedded into the file is executed and replaced by its resulting output when passing this file to the embedded stream special function provided in the set of libraries published with this work. A special executable tool called "odfi_tcl_embedded" can be used to convert a local file.

4.3.3 Tool integration examples

4.3.3.1 SVG View

Using the SVG scene graph language presented in 4.2.5, it is easy to generate an SVG view for a part. The generator creates an SVG scene graph tree, and structures the view parts as presented in Figure 4-43:

- ✓ A Row on top for the column names
- ✓ A flowGrid for the pins array and the row names
  - o The list of pins is sorted by location, at each row begin a row name text element is added
  - o The Flow grid layout function is called with a number of columns being the width of the pins array plus one.
- ✓ A column to place the pins and row names array under the column names

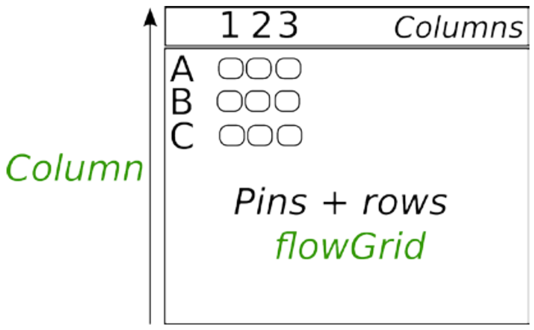


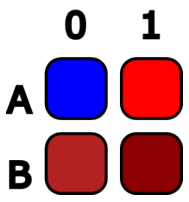
Figure 4-43 SVG generator scene graph layout

The SVG generator also supports some rules as described in 4.3.1.3. Supported parameters are for example:

- ✓ color : a standard SVG color can be set to change the output color of the pin.
- ✓ shape: values like rect, circle or triangle can be set to change the pin graphical representation.

Two examples of SVG outputs are:

- ✓ A very simple 4 pins part example can be tested on the LiveRun website (see Appendix A script is: sources/4.3-Package/part-to-svg.tcl), and produces the result shown on the right.



- ✓ Figure 4-44 presents a more evolved output customised for documentation purpose was created for the Extoll integrated circuit,

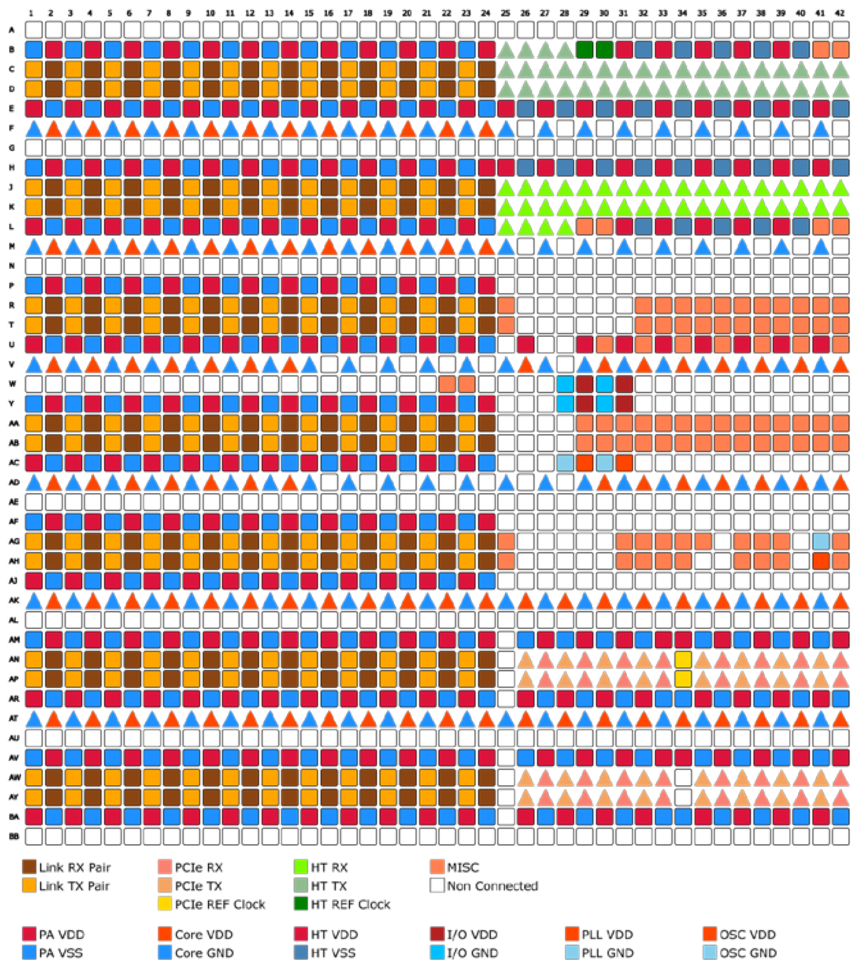


Figure 4-44 SVG view example for the Extoll ASIC (~3000 balls)

4.3.3.2 *Cadence capture integration*

Cadence Capture Design entry is a Printed Circuit Board (PCB) design tool. A PCB is engineered in two phases:

- ✓ First the circuit schematic must be defined, i.e. all the connexions between parts.
- ✓ Afterwards, the schematic goes in layout mode, where the signals are physically traced on the physical view of the circuit board.

The part language library can be loaded inside the Cadence Capture Design software to assist the user when importing data into the software. Indeed, all pins of a part must be created per hand in the database if the part is new to the environment. Using the part language, we wrote an import function which, using the internal programming interface of the tool creates the part definition in the schematic database automatically.

As Illustrated in Figure 4-45, upon loading, a small extra TCL graphical interface pops-up and offers the user to open a file for importing. When imported, the new part appears in the database, and its pins definition can be seen in the part schematic view.

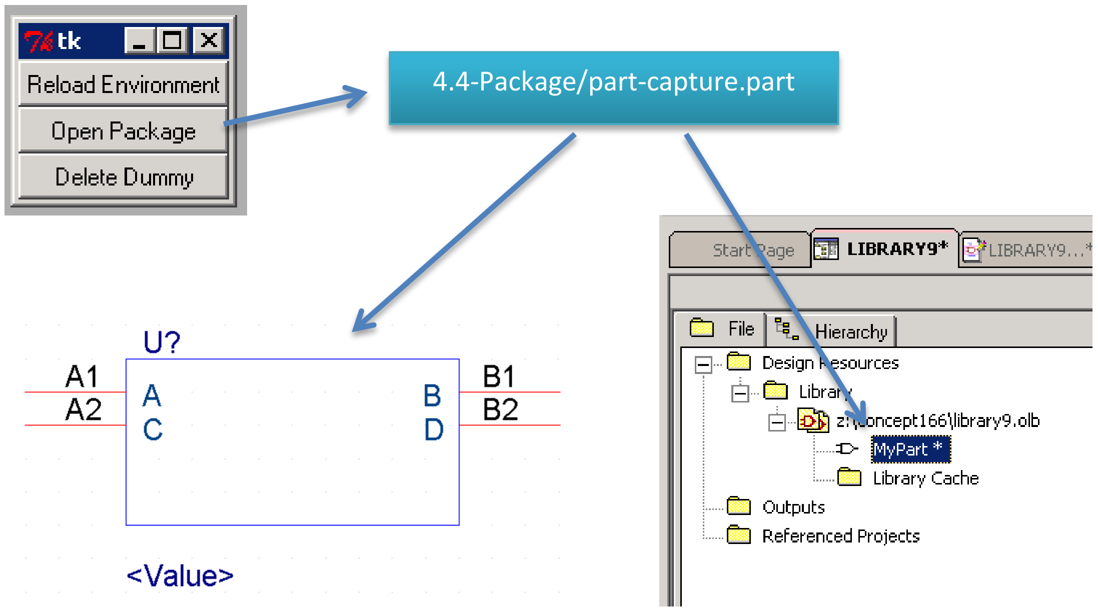


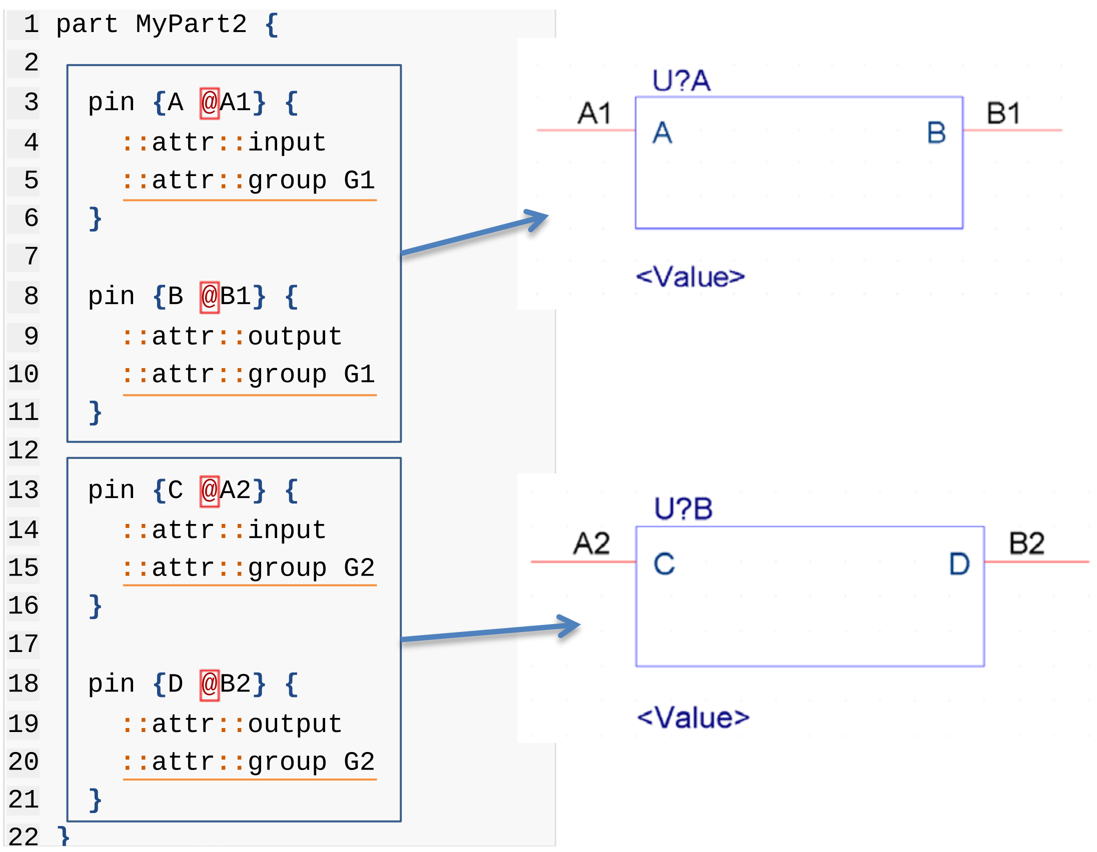
Figure 4-45 Part file import in capture

4.3.3.2.1 Large part support: usage example of the generic group attribute

The part import function is implemented as a generator and supports rules definitions as presented in 4.3.1.3. However, we found an interesting usage for an attribute called “group”, which is defined in the core programming interface. It is generally a good idea to provide grouping information on the pins which can be used as a sorting key for the output generators.

In Capture, a part is called a symbol, and if it is large, it is divided in sub-symbols which each contain a certain number of pins. Each of those sub-symbols must be placed individually on the schematic, and thus should preferably contain pins which are related to each other. The import generator can translate the group attribute to the correct input data for Capture to create matching sub-symbols.

Figure 4-46 shows a simple part definition with group attributes, and the two sub-symbols yielded in the capture symbol library. The same generic group attribute could be for example used by a documentation generator to sort the pins in a table.



```
part MyPart2 {

  pin {A @A1} {
    ::attr::input
    ::attr::group G1
  }

  pin {B @B1} {
    ::attr::output
    ::attr::group G1
  }

  pin {C @A2} {
    ::attr::input
    ::attr::group G2
  }

  pin {D @B2} {
    ::attr::output
    ::attr::group G2
  }
}
```

Figure 4-46 Automatic sub-symbol assignment from generic group attribute

4.3.4 Outlook and integration in actual work

The presented part language has the specificity of being at the boundary of design flows. It can provide useful data along two axes:

- ✓ Transversal as exchange vector between users of a defined part: tools, business and engineering partners etc…
- ✓ Downward to integrate with the part engineering design flow itself.

There would be a great advantage for business entities to providing TCL part files to each existing part. Its simple and clear syntax make it close to a simple text file, and its dynamic aspect enables a tight integration in tool chains: one simply have to source the file, and start coding to extract the useful information relevant to a specific use case.

On the downward axe, a TCL part definition format was already used in combination with the Floorplanning API presented in 4.2 for the implementation of the Extoll ASIC. It was generated from a script and used to automatically position the integrated circuit bumps based on the part pins positions (the bumps are connecting the die to the part package). For a future integrated circuit design, it would be interesting to write a part file yielding the correct pins, and use it directly with the Floorplanning API and any other design tool, instead of generating the same part description in multiple different formats for each target.

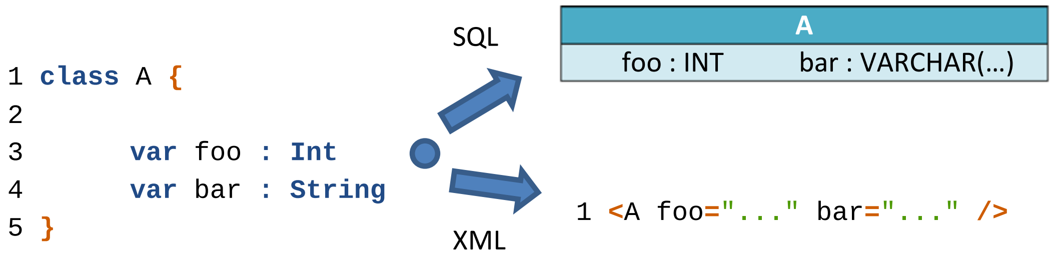
4.4 OOXOO: A dynamic XML data binding interface

To exchange data between actors of a software architecture, may they run concurrently or be part of a process chain, Extensible Markup Language (XML) [55] documents are widely used, both because of their structural aspects (we speak of XML trees) and the freedom of elements definition which is given to the software architects. Such an exchange format is used by the register file generator tool described in 4.1 to enable interfacing between high-level software and the described hardware registers.

This section presents a novel and highly flexible data binding architecture for structured data formats like XML, on top of which, a set of simple components have been developed to efficiently bind a register file interface into the application space (see 4.4.4).

4.4.1 Data binding for XML

Any software developer dealing with static data storage, will usually be confronted with the implementation of a functional layer to cope with the bidirectional translation between application-oriented data structures and the storage. These processes are called marshalling and un-marshalling. The storage backend can take the form of binary files, structured trees like XML, or very commonly a relational database engine like MySQL, as illustrated in Figure 4-47.



```
class A {

      var foo : Int
      var bar : String
}
```

A	
foo : INT	bar : VARCHAR(...)

```
<A foo="..." bar="..." />
```

Figure 4-47 Simple class translated to XML or a Relational Database table

Typically, this translation process involves:

- ✓ Defining the application data structures
- ✓ Defining the storage format
- ✓ Driving the data engine the store or retrieve data
- ✓ Optimizing performance, by caching objects for example
- ✓ Migrating older data to newer versions

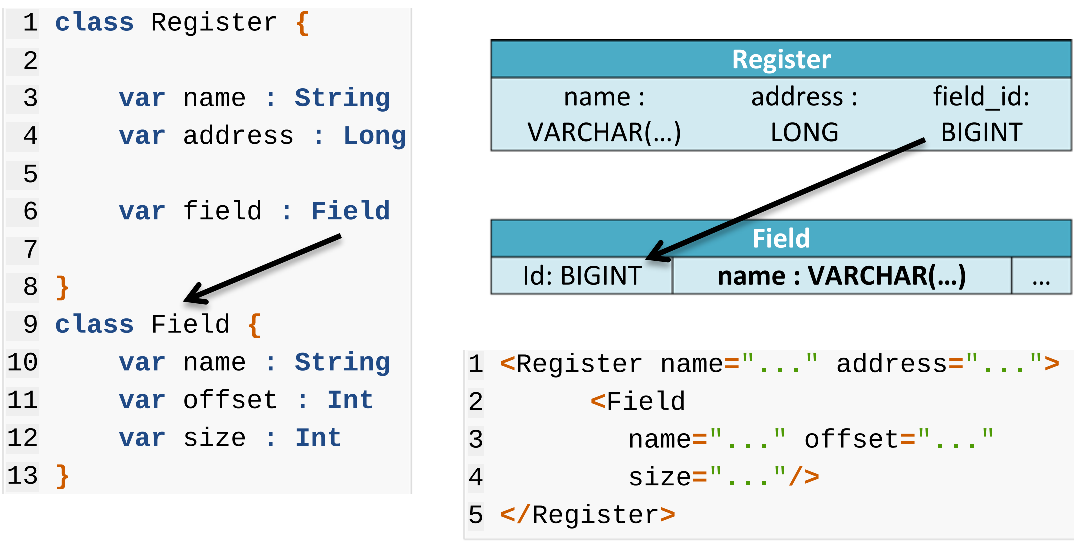
In the context of a typed introspective runtime, like the Java Virtual Machine (JVM) offers, (un)marshalling can be greatly lightened by delegating the data format definition and interfacing with the storage engine to an automatic binding layer, for example called Object/Relational mapping (ORM) in the case of a relational database.

In Figure 4-47 for the SQL storage case, such an ORM layer would automatically:

- ✓ Create the Database table
- ✓ Generate the Queries to retrieve or update data from/to the tables
- ✓ Create and populate objects upon user requests
- ✓ Update the table based on object update or insertion upon user request
- ✓ Optionally Maintain an object cache to optimise data retrieval

In Java, the most popular implementation is the Apache Foundation Hibernate project [56] , but concurrent ones exist from a wide range of actors (Oracle TopLink, Eclipse EclipseLink etc…).

Object-Relation mapping allows recursive imbrication of objects (A can contain B, once or multiple times), although it then maps to flattened tables. Intuitively, we can see that XML documents implicitly offer data structure hierarchies, we just need to be able to generate an XML tree for a given data type, and place it in its container's tree, as illustrated in Figure 4-48.



```
class Register {

    var name : String
    var address : Long

    var field : Field

}
class Field {
    var name : String
    var offset : Int
    var size : Int
}
```

```
<Register name="..." address="...">
     <Field
        name="..." offset="..."
        size="..."/>
</Register>
```

Figure 4-48 Data structure composition example

4.4.1.1 *Automatic data model generation and validation*

An issue with XML documents is the definition of data types present in the tree, as data nodes only are represented by strings (TEXT_NODE in XML semantic). In a relational database however, the user must explicitly set the type of each table column (VARCHAR, TEXT, INT etc…), as was presented previously in Figure 4-48.

To palliate to this issue, an XML data modelling standard called XML Schema [5][6] was created around 2001. It is itself an XML document which describes the structure that a specific type of XML documents must conform to. It defines for example:

- ✓ Available Elements
- ✓ Composition relations between elements
- ✓ Type inheritance
- ✓ Simple types for attributes and non structural elements (int, long, float,string etc…)

Using XML Schema, our previous example with *Register* and *Field* can be rewritten as presented in Example 4-1 (some data structures have been omitted for concision):

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="…"
    targetNamespace="…">

    <element name="Register">
       <complexType>
          <sequence>
           <element ref="tns:Field"></element>
          </sequence>
          <attribute name="name" type="string"></attribute>
       </complexType>
    </element>

    <element name="Field">
        <complexType>
            <attribute name="size" type="int"></attribute>
        </complexType
    </element>

</schema>
```

Example 4-1 Register and Field XML Schema example

This document would be used when parsing XML to validate it against the expected format, and perform (un)marshalling, as depicted in Figure 4-49. Moreover, it allows generating the required data structures for the developer by compiling the Schema to code.

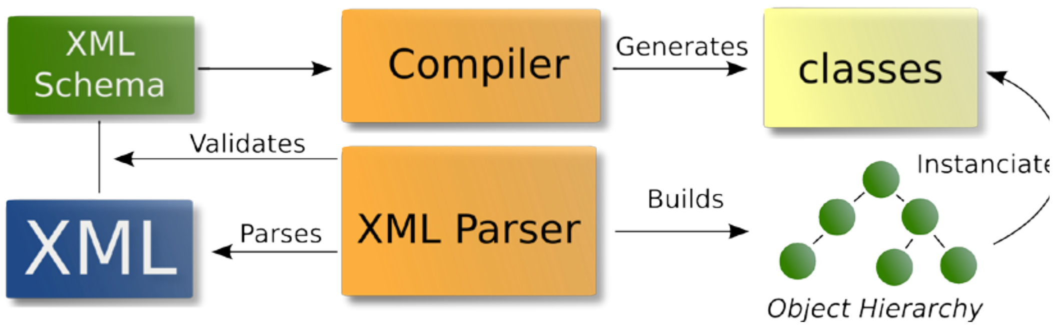


Figure 4-49 XML marshalling/unmarshalling and validation scheme

4.4.1.2 *Flat binding*

The historical reference implementation for XML data binding in Java is the Java Architecture for XML binding (JAXB) [59] . In its first version released in 2003, it required the user to follow a precise workflow:

1. Write an XML Schema
2. Generate code using an XML Schema compiler (In that case a lot of interfaces and classes)
3. Parse/Generate XML in application using generated data structures

Back then, the generated code was quite complex, and there was no other choice than starting with writing XML Schema. This approach was however not very efficient for the user because of XML schema verbosity, and considering that most applications start with a very limited number of data structures, it would have been more convenient to just be able to write simple classes, and have them mapped back and forth from/to XML until the feature set is stable enough. Moreover, during development, XML validation is not critical because the application runs in a safe environment, where actors keep data consistent.

This is how the first version of OOXOO was designed in 2005 (as a student project) to palliate those issues. The main Idea for the data binding was to simply use the newly introduced Java Annotations (starting with Java 1.5), which allows embedding class metadata in the byte code. These metadata were used to mark the class fields that should be considered in XML. This approach has been used by JAXB from its version 2.0, released as a standard in 2006 [60] , as presented in Figure 4-50.

```
@XmlRootElement
class Car {

    @XmlElement
    var model : String

    @XmlAttribute
    var color : Int

    // Not relevant for XML marshalling
    var foo : String
}
```

```
<Car color="...">
      <model>...</model>
</Car>
```

Figure 4-50 Java Annotations for XML binding

Some other implementers, like XMLBeans [61] or JibX [62], however kept relying on the XML Schema compiler workflow, sometimes even adding extra XML to object mapping documents in the case of JibX.

Benchmarking of the various binding implementations is not a very well covered field, because of the lack of a precise framework for the implementations. Even if JAXB is a Java Specification, it is not very well followed (OOXOO itself does not respect any standard), and as usually (un)marshalled data sets tend to be quite small, they don't really impact application performances that much in their usage, making it a poor criterion in the design space.

4.4.1.3 Application Binding

We have seen how XML binding can help map data to objects in the application space. We can raise a concern about how those objects are to be used by which components (graphical user interface, networked data exchange etc…). Two main questions appear then:

1. Does the data structure diverge from its target usage?
    - Example: The Flat XML is used to build/restore a Map
2. How do upper application layers interact with the data objects?
    - Example: We want to edit the data in a Graphical User Interface (GUI)

This concept is called application binding, or functional binding. The first point is usually solved by the features of the chosen binding library, typically by providing special type handlers to modify the marshalling behaviour, but requires specific configuration.

The second point can be considered out of scope, the software architects being responsible for correct read-modify-update of data in the application context. However, boiler plate code which just handles data copy and validation between types that are incompatible with each other, conversions (etc…) is difficult to avoid, and complex scenarios difficult to test.

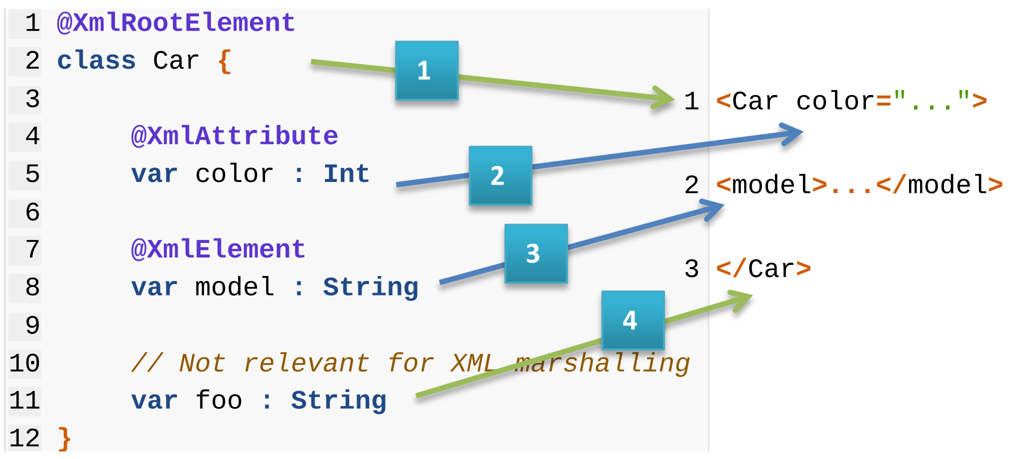
Can we then handle this so called Application binding in a generic way that could provide an answer to both previous statements? The design of OOXOO v2 (Java implementation) and OOXOO v3 (Scala implementation) proposes a solution by using a dynamic binding approach, instead of a simple flat binding.

4.4.2 Dynamic hierarchy

When using flat binding, as previously illustrated in Figure 4-49 and Figure 4-50, the data binding logic is located in a special function which performs marshalling and unmarshalling. The two issues raised in 4.4.1.3 however, speak for merging the pure data and the binding process together to improve application integration.

The simple idea implemented in the OOXOO library, is to get every data field to be able to generate its own marshalled memory representation, which then can be written out as XML. This way, every data field embeds the logic to generate itself, depending on its type. This general concept is presented in Figure 4-51, where marshalling would:

1. Open the *Car* element.
2. Find the field *color* as an attribute; ask field to output its value as attribute.
3. Find the field *model* as an element; ask field to output its value as an element.
4. No more data, close the *Car* element.



```
@XmlRootElement
class Car {

    @XmlAttribute
    var color : Int

    @XmlElement
    var model : String

    // Not relevant for XML marshalling
    var foo : String
}
```

```
<Car color="...">
<model>...</model>
</Car>
```

Figure 4-51 Structured Binding base idea

What can be noted though is clearly that:

1. The presented data types are not rich enough to marshal themselves
2. There are two types of data:
    - Structural : This is the Car class, that generates a <Car> XML element
    - Simple : These are the data fields that map simple types to XML elements or attributes (model and color)

If we can correctly solve the first issue, the solution to the second one should be naturally derivable.

We are going to present in the next paragraph the *Buffer* and *DataUnit* (DU) classes, which are the basic building blocks of OOXOO. Their responsibility is to provide the two interfaces required to solve the previous matters:

- ✓ *DataUnit* : A Marshalled data representation
  - ➔ We can use this information to generate XML and or integrate with other application layer
- ✓ *Buffer* : A chain of objects, that drive the marshalling process by exchanging *DataUnits*
  - ➔ The Buffer chain can be used as integration point for customisation

4.4.2.1 Buffers and Data units

As just mentioned, the basic architecture of our library relies on *Buffers* exchanging *DataUnits* (DU), as can be seen in Figure 4-52 . The Buffers can be extended by classes, provide specific functionalities, and be composed together throughout classes. *DataUnits* represent marshalled data, with optional structure information, so that buffers can for example build or read XML.

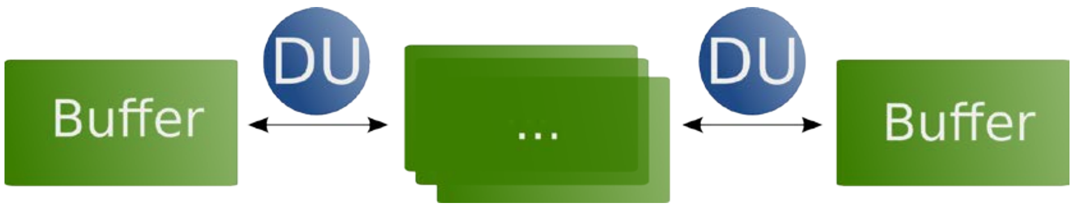


Figure 4-52 Buffer chain and DataUnit overview

4.4.2.1.1 Buffers

In the problem statement, we mentioned that OOXOO was designed to improve integration with the application, whilst providing data input/output through marshalling. To split those two application cases, the Buffers exchange with their neighbours using two virtual channels: Stream and Push/Pull.

Stream interface

The Stream interface is used for I/O operations like XML parsing or exporting. It only generates Data Units on the right side.

A simple Data unit exchange example is shown in Figure 4-54 (syntax elements removed for picture concision)

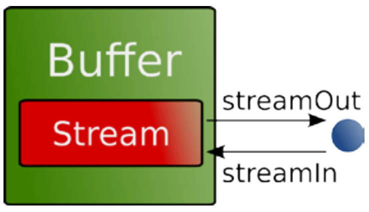
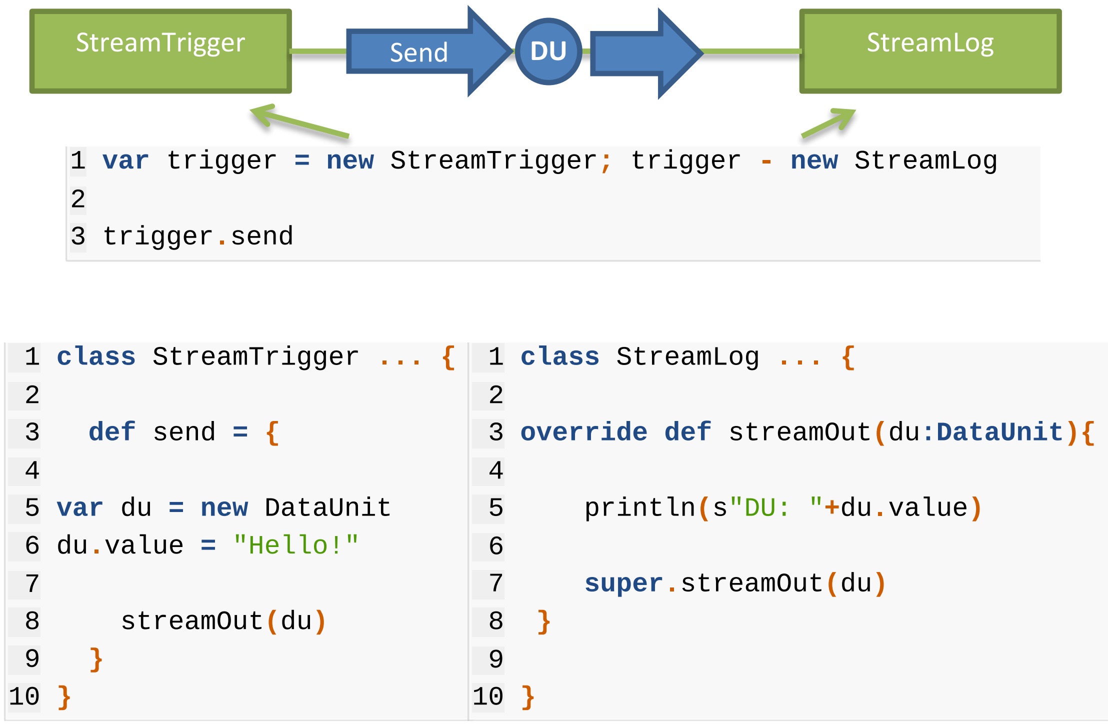


Figure 4-53 Buffer Stream interface

Repository: thesis
Path: example-project/src/main/scala/ooxoo/simple/OOXOOSimpleExamples.scala



```
var trigger = new StreamTrigger; trigger - new StreamLog

trigger.send
```

```
class StreamTrigger ... {

  def send = {

var du = new DataUnit
du.value = "Hello!"

    streamOut(du)
  }
}
```

```
class StreamLog ... {

override def streamOut(du:DataUnit){

    println(s"DU: "+du.value)

    super.streamOut(du)
  }

}
```

Figure 4-54 Data Unit exchange using stream interface

Push/Pull interface

The Push/Pull interface follows the same principle as the stream interface, but can be used on the buffer chain in both left and right directions. Figure 4-56 presents a simple example setup to show push and pull calls on Buffer instances on both direction, and Figure 4-57 a simple application example to resolve a numeric value multiplied by a buffer (The full sources are not reproduced for concision).

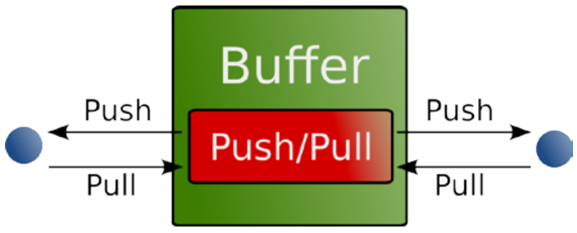
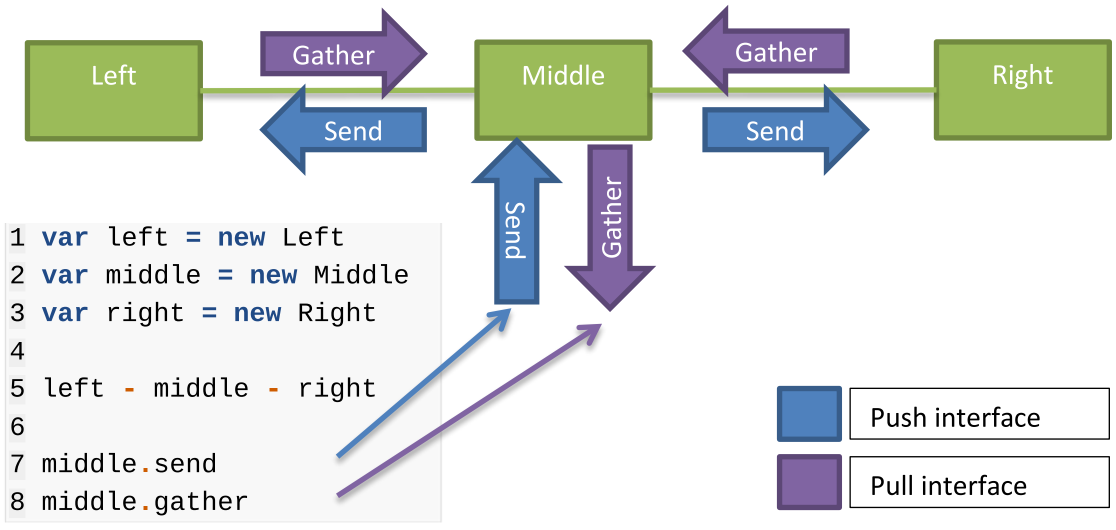


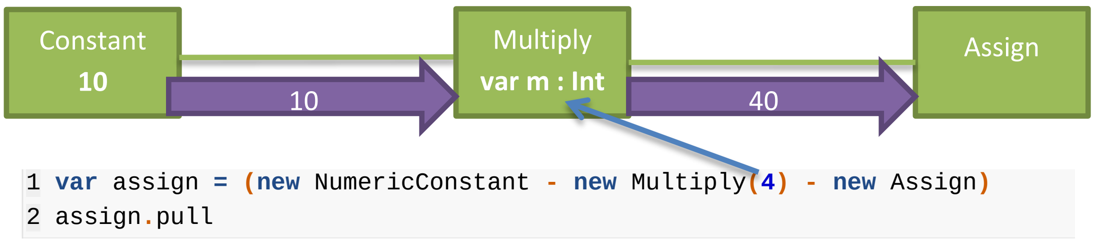
Figure 4-55 Buffer Push/Pull interface

Repository: thesis
Path: example-project/src/main/scala/ooxoo/simple/OOXOOSimpleExamples.scala



```scala
var left = new Left
var middle = new Middle
var right = new Right

left - middle - right

middle.send
middle.gather
```

Figure 4-56 Simple Push and Pull interface example



```scala
var assign = (new NumericConstant - new Multiply(4) - new Assign)
assign.pull
```

Figure 4-57 Pull interface applied to numeric expression resolution

4.4.2.1.2 DataUnits

After defining how Buffers exchange DataUnits, we need to define how these hold content. Depending on the Buffer that generated it, the content of a DataUnit vary, but to be able to mirror an XML structure, it also has to be able to hold following information:

- ✓ Element name and Namespace: If the DU is mirroring an XML element
- ✓ Attribute name and Namespace: If the DU is mirroring an attribute of current Element
- ✓ Value: If the DU is holding any kind of value, it is serialised as a string
- ✓ Hierarchical: DU represents structured data, because we need to know at some point when opening or closing a hierarchy (an element in an element for example)

The following table summarises the DataUnit values depending on the possible cases.

Case	Element	Attribute	Value	Hierarchical
Element open	☺			☺
Element close				☺
Element open + value	☺		☺	☺
Attribute		☺	☺	
Value			☺	

4.4.2.2 Element Structural Buffer

Finally, if we now can generate data units for all XML cases, we need a way to map the object structure to a valid DataUnits stream. We presented earlier two types of data in an object hierarchy: Structural and Simple.

4.4.2.2.1 Simple Data handling

The simple data are the one which only hold a value like a String, Integer, Long etc…. We can create Buffers which map those simple data types to DataUnits, but we wouldn't be able to know at runtime if the field is supposed to map to an attribute or an XML element.

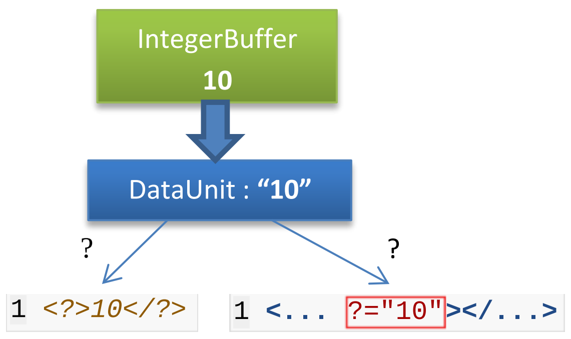
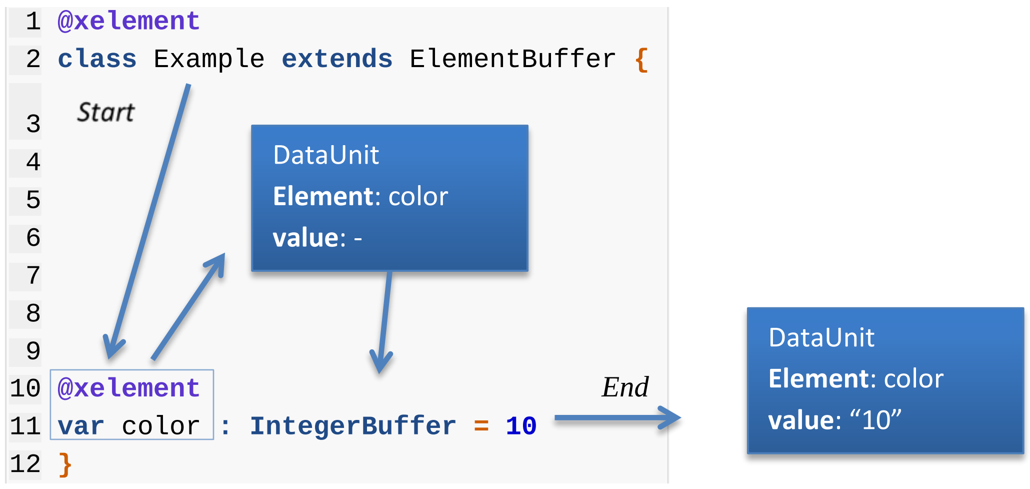


Figure 4-58 Simple DataBuffer can't map to structure

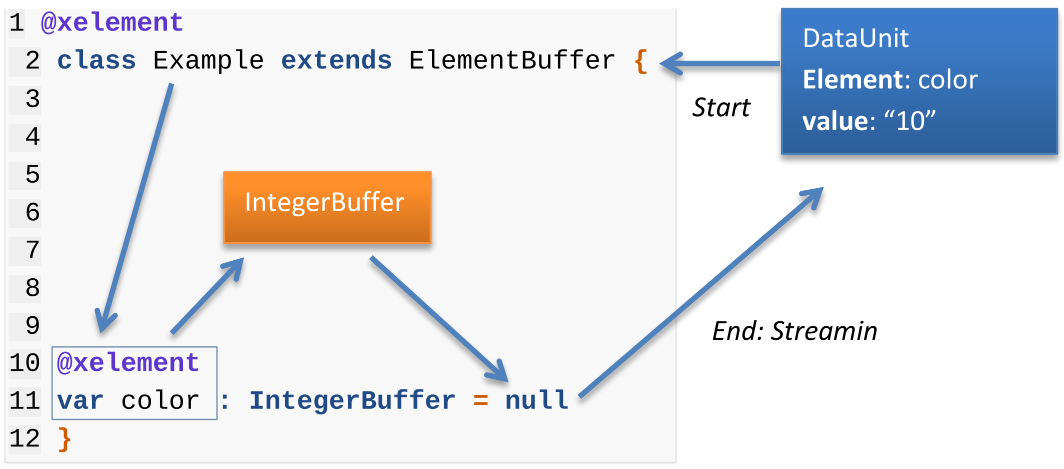
The structural buffers however are the containers of simple data. Based on annotations provided as metadata in a class definition, it can setup the structure data of the data unit, and pass it to the simple data buffer which only needs to merge-in its internal value (Figure 4-59).



```
@xelement
class Example extends ElementBuffer {








@xelement
var color : IntegerBuffer = 10
}
```

Figure 4-59 DataUnit production for a simple data mapped to an element

The same process happens when receiving DataUnits in a Structural buffer as well, but the other way around: First create the simple data buffer and then stream to it the DataUnit from which it will extract data for its interval value (Figure 4-60).

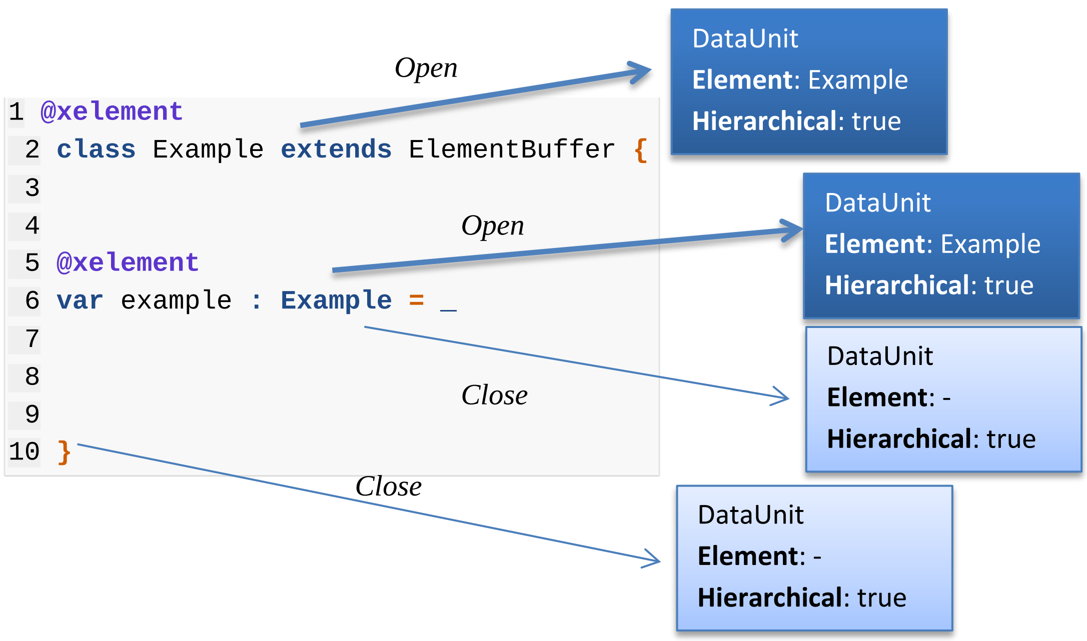


```
@xelement
class Example extends ElementBuffer {







@xelement
var color : IntegerBuffer = null
}
```

Figure 4-60 DataUnit receive for a simple data mapped to an element

4.4.2.2.2 Hierarchy handling

To repeat this input or output process on an object hierarchy, the structural buffer simply traverses all its class fields, previously listed using the Java reflection API. The last case is met when a structural buffer meets another structural buffer in its content. It then simply delegates the ***streamIn*** or ***streamOut*** process to this buffer, triggering sub-tree production. Figure 4-61 shows the DataUnit Production for structure mirroring.



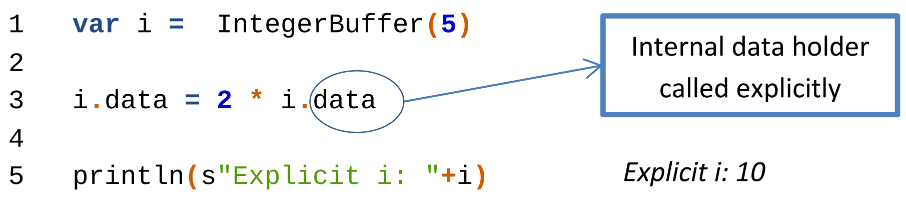
```
@xelement
class Example extends ElementBuffer {


  @xelement
  var example : Example = _



}
```

Figure 4-61 Open-Close across hierarchies

4.4.2.3 The simple data types issue

Repository: thesis
Path: example-project/src/main/scala/ooxoo/simple/SimpleDataTypes.scala

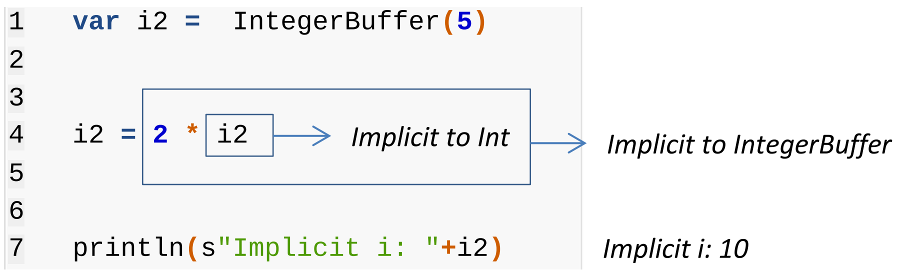
An important issue with the presented architecture is that it basically does not allow simple data types like String, Int, Long (…). It is problematic for simple data structures where the user does not really need the Buffer feature, but also inconvenient to work with when manipulating the data, as the user always have to access information in an explicit way:



```
var i =  IntegerBuffer(5)

i.data = 2 * i.data

println(s"Explicit i: "+i)
```

```
Explicit i: 10
```

To palliate to this syntax issue, it is possible to specify implicit conversion from a simple data buffer to its underlying data type back and forth. This mechanism has been detailed in 2.2.1.



```
var i2 =  IntegerBuffer(5)


i2 = 2 * i2



println(s"Implicit i: "+i2)
```

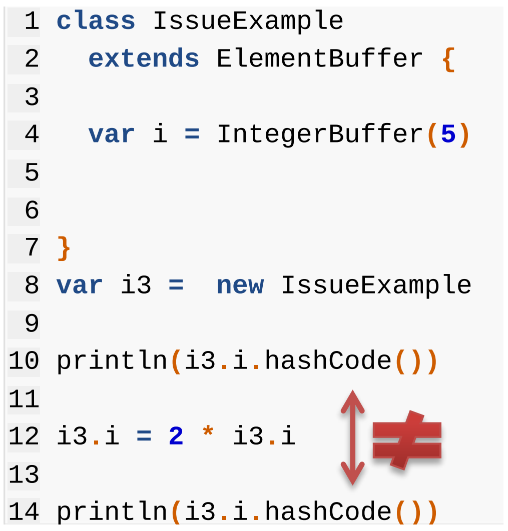
```
Implicit i: 10
```

Another possibility would be naturally to add a type map between non-buffer types and buffer types for the structural buffers to make conversions, but it has not been implemented yet because the implicit mechanism was enough to live with.

4.4.2.3.1 Implicit conversion trap

Implicit conversions can lead to some cumbersome issues if not properly considered. It is especially true in our buffer architecture and here is why:

- ✓ When performing implicit conversions, new object instances are created to represent the created value.
- ✓ When assigning a buffer variable through implicit conversions, we get a new instance of the buffer.
- ✓ If a buffer chain had been setup previously, it remains on the pre-update value, which is probably going to be garbage collected.
- ✓ The new instance has no buffer chain connected to it

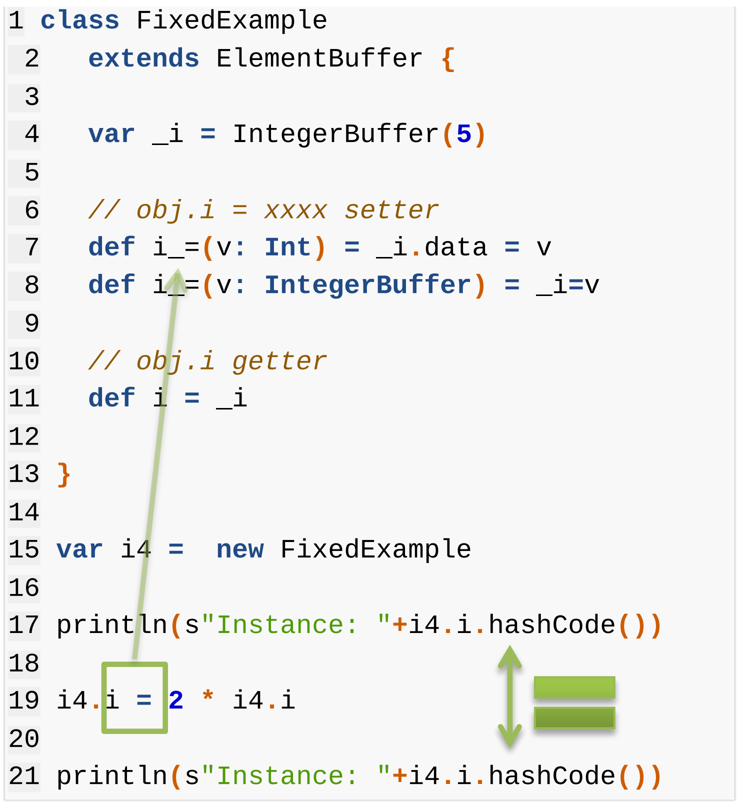


```
class IssueExample
  extends ElementBuffer {

  var i = IntegerBuffer(5)


}
var i3 =  new IssueExample

println(i3.i.hashCode())

i3.i = 2 * i3.i

println(i3.i.hashCode())
```

Figure 4-62 Class-field instance override instead of value update

In most use cases, this won't be an issue, but when complex buffer chains are setup, it might happen, and the "error" might become difficult to find out, as the problem lies in a wrong language usage.

A good practice to avoid that kind of issues consists in using a getter-setter mechanism in classes, to ensure simple data type assignments to Buffers only merges the underlying data holder, rather than replace the whole Buffer. This cost a data structure setup overhead, but if the classes are generated, the code generator can take care of this. Fortunately, the flexibility of Scala makes it possible to hide the getter-setter methods behind a syntax strictly identical to the one of standard variable assignment. Figure 4-63 provides an illustration for a class variable *i* of type *IntegerBuffer*, which can be set using a standard = operator, while the implementation correctly differentiates between resetting the whole buffer or just the internal value.



```
class FixedExample
   extends ElementBuffer {

   var _i = IntegerBuffer(5)

   // obj.i = xxxx setter
   def i_=(v: Int) = _i.data = v
   def i_=(v: IntegerBuffer) = _i=v

   // obj.i getter
   def i = _i

}

var i4 =  new FixedExample

println(s"Instance: "+i4.i.hashCode())

i4.i = 2 * i4.i

println(s"Instance: "+i4.i.hashCode())
```

Figure 4-63 Scala = operator used as getter-setter

4.4.2.4 Collections

The last vital core feature required is a way to handle collections of Buffers. It is very common to repeat the same data structures, and thus create a collection typed field (like a List) in the structure of a class. However, we only want to rely on the generic Buffers architecture to handle hierarchy traversing, and avoid special handling as much.

A special List type called *XList*, which extends a Mutable List and imports the Buffer trait, has been created for this purpose. A structural buffer will then only see the List as a buffer, and pass-on DataUnits when required. The *XList* simply redefines the ***streamIn*** and ***streamOut*** behaviour of a normal Buffer and repeats it for the list content, or stores incoming elements in the list.

4.4.2.4.1 DataUnit production

Producing DataUnits is an easy task. The process is the same as the one presented in Figure 4-59, only the *XList* produces as many DataUnits as it contains elements. Figure 4-64 illustrates this for an Element DataUnit passed by the enclosing structural buffer to a list holding *IntegerBuffer*.

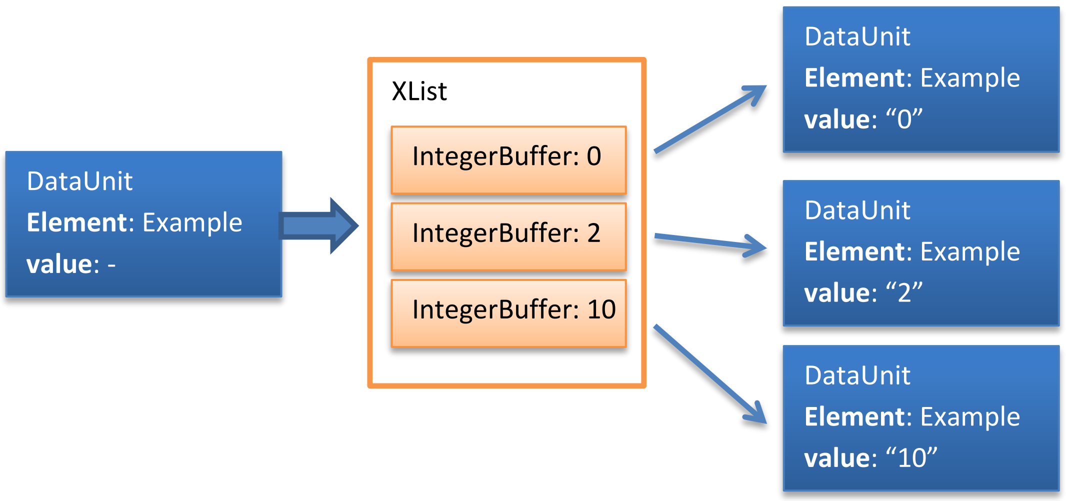
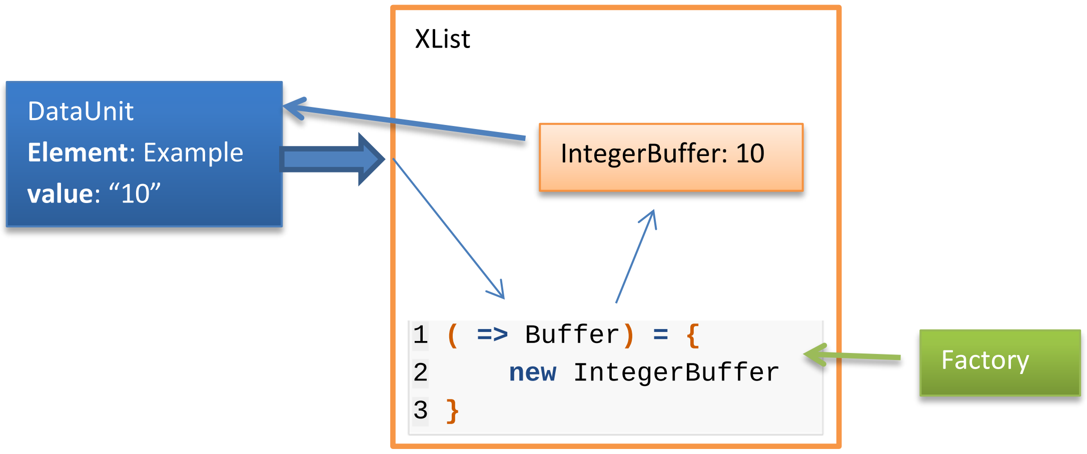


Figure 4-64 XList DataUnit repetition for a collection

4.4.2.4.2 DataUnit consumption

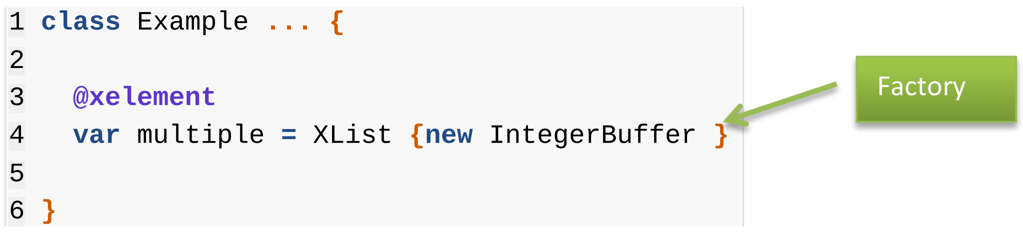
When receiving element, it is once again exactly the same procedure as in Figure 4-60, with the small difference that the structural buffer instantiates the *XList*, and not its content, which must be created by the latter. The implementation of the *XList* is totally generic, and thus cannot guess the type of data which it stores to perform instantiation. It is required to provide an anonymous function to the XList constructor, which will be used as a factory when receiving elements. Figure 4-64 illustrates the conversion path from a received *DataUnit* to an *IntegerBuffer* by using a factory lambda-function to create the base Buffer instance.



```
( => Buffer) = {
    new IntegerBuffer
}
```

Figure 4-65 XList DataUnit receive with factory-based content instantiation

The following Figure 4-66 shows an XList declaration example inside a class. On line 4, the XList is created, with a constructor argument being the lambda function used as factory when receiving DataUnits.



```
1 class Example ... {
2
3   @xelement
4   var multiple = XList {new IntegerBuffer }
5
6 }
```

Figure 4-66 XList instantiation example

4.4.3 Marshalling and un-marshalling: the I/O layer

So far we have setup ground for marshalling and un-marshalling by presenting the Buffer and DataUnit architecture. It is easy to see that marshalling or un-marshalling data could be performed by creating a special buffer that will receive DataUnits (for marshalling), or produce some (for un-marshalling).

When considering a data format like XML, we can say that this buffer will be a serialising or deserialising buffer (called SerDes buffer for short) , because it will be converting an object hierarchy to and from a character stream.

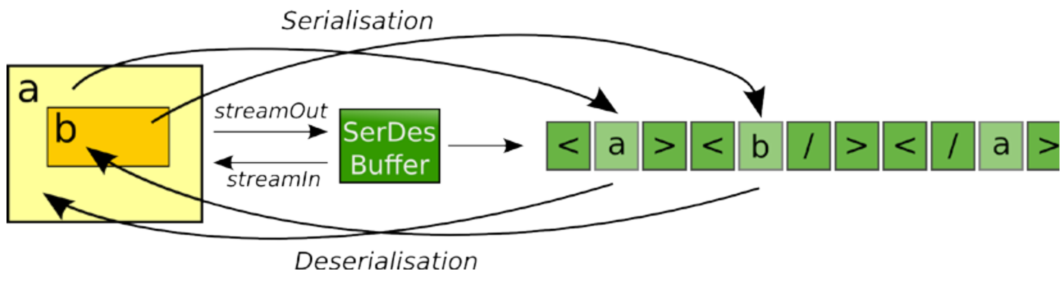


Figure 4-67 Serialisation and de-serialisation of an object hierarchy

In Figure 4-67, where “a” and “b” would be two structural buffers, we can note that the SerDes buffer has a many-to-one relation from the data structure to the character stream on the serialisation side, and a one-to-many relation on the deserialization side. We can thus derive following characteristics:

- ✓ Only one instance of the buffer is required to serialise or de-serialize data (The "one" side of the relation).
- ✓ This same instance must be present on all the data buffer chains to be serialised or de-serialised (The “many” side of the relation)
- ✓ Additionally, the SerDes process is typically punctual and takes place after a synchronisation point between all threads which are using the data, until the next I/O phase (Figure 4-68). The persistent presence of the SerDes buffer on the buffer chains is thus not desirable as it is not related to the standard application workflow of data processing. The SerDes buffer should then be transient and disappear after hierarchy traversing.

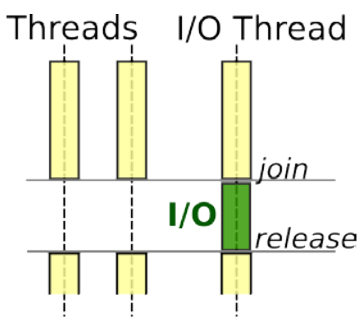


Figure 4-68 I/O synchronisation phase

To support those characteristics, a subset of the standard buffer has been defined, called I/O buffer. I/O buffers must follow the object hierarchy traversing during ***streamOut*** or ***streamIn*** operations, but they also have to be transient. This is easily achieved by implementing following behaviour:

- ✓ For all Buffers: After streamIn or streamOut, remove the I/O Buffer
- ✓ For Structural Buffers: When entering a sub-hierarchy, connect the I/O buffer to the target buffer.

However, the I/O buffer must also return to the parent hierarchy when a sub-hierarchy has been fully processed. Standard tree-traversing always feature a stack to be able to track the parent-child hierarchies, stack which is missing at the moment. The intuition would bring us to making the structural buffers follow the hierarchy. It would work if both streamOut and streamIn where driven by the structural buffer and thus blocking.

Figure 4-69 and Figure 4-70 illustrates the behaviour of the streamOut and streamIn processes when hitting hierarchies (simple data are a corner case because they are single atomic operations). The streamOut side is blocking on sub-hierarchy processing, but not the streamIn side as it is driven by the I/O buffer. Only the I/O buffer will trigger the end of sub-hierarchy, this is why the parent hierarchy cannot retrieve the I/O after the sub-hierarchy processing, whose return does not imply the sub-hierarchy has been completed.

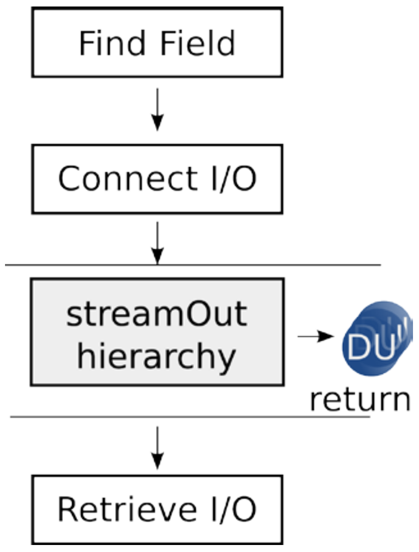


Figure 4-69 Streamout driven by hierarchy

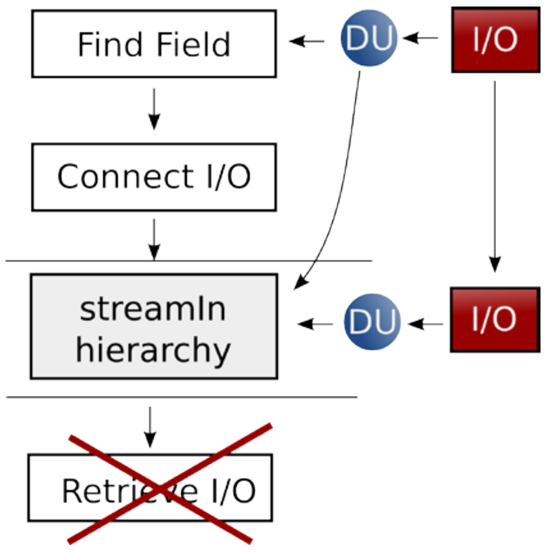


Figure 4-70 Hierarchy driven by streamIn

Note: This characteristic of the architecture is difficult to understand, so was it also at implementation time, but reveals itself quite elegant.

To solve the hierarchy tracking, the solution is simpler than it appears. Instead of handling it on both sides independently, we managed to implement the behaviour at the level common to both behaviours: I/O buffer connection (enter hierarchy) and removal (leave hierarchy). The special I/O Buffer class behaves in both cases as following:

- ✓ On connect:
  - o Stack the buffer to which to I/O buffer is connected to.
  - o Connect normally to the new buffer chain.
- ✓ On remove:
  - o Connect back to stack top.

Finally, the implementation does not only move around only one I/O Buffer, but an I/O chain, which is the set of buffers connected after an I/O buffer.

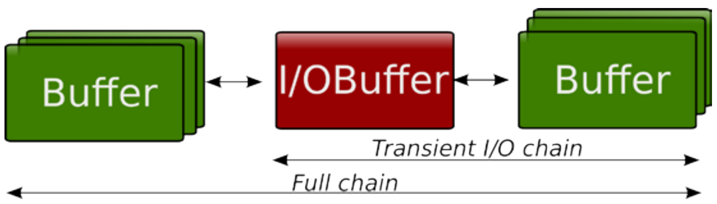
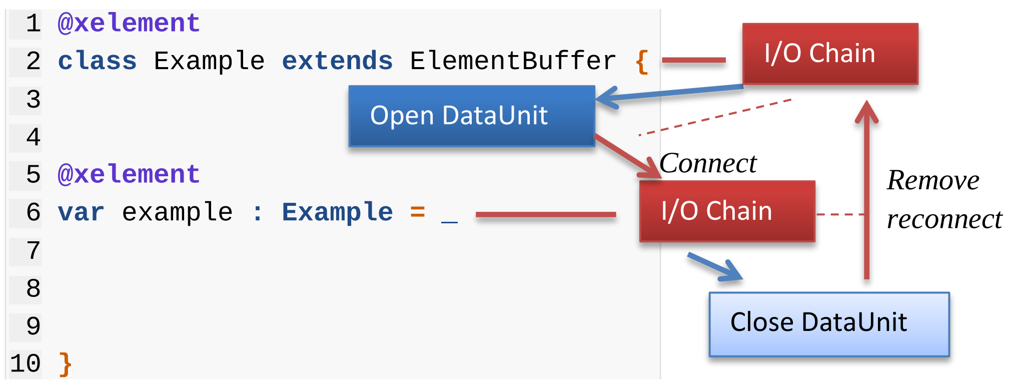


Figure 4-71 I/O Chain

Figure 4-72 illustrates both connect and remove process on the I/O chain for the streamIn case where both operations are driven by receiving data units:

1. First a hierarchy open DataUnit is received, and matched against the example class-field
2. The I/O chain is connected to the class-field Example Buffer, streamIn continues on this sub-hierarchy
3. When the hierarchy close DataUnit is received, the I/O chain is removed from the class-field and jumps back to the container Buffer automatically



```
@xelement
class Example extends ElementBuffer {


@xelement
var example : Example = _



}
```

Figure 4-72 I/O chain hierarchical connect/remove

The streamOut process is identical, but the DataUnits are produced by the classes instead of being received.

4.4.3.1 Handling non hierarchical buffer levels: the collection case

If we analyse the I/O process for a class containing an XList collection, it appears that the I/O chain would get stuck at this hierarchy level. Figure 4-73 details the I/O chain location when processing an XList content, and shows that it does not return to the structural buffer level. During streamOut, no problem would be seen, as the XList must remove the I/O chain after producing its content, but streamIn would see all further DataUnits consumed by the list.

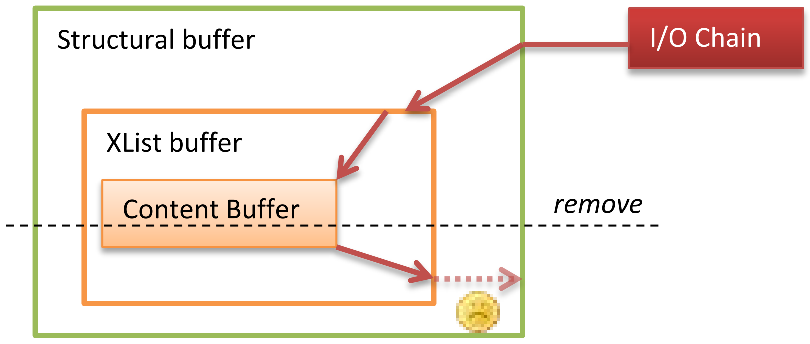


Figure 4-73 I/O gets stuck in XList buffer level during streamIn

This issue can be characterised in a generic way, by noticing that the I/O process needs to consider only the real relevant data buffers. As the collection buffer is just a container which is not mirrored in the data representation, it can be seen as a "virtual" hierarchy. The solution has thus been implemented in a very simple way by defining an interface, used as type marker, called *IOTransparentBuffer*.

When reconnecting to the stack top during removal, an *IOBuffer* simply ignores any stack-top which is type *IOTransparentBuffer*, and jumps one level higher. All buffers which are special container of data must extend the *IOTransparentBuffer* interface.

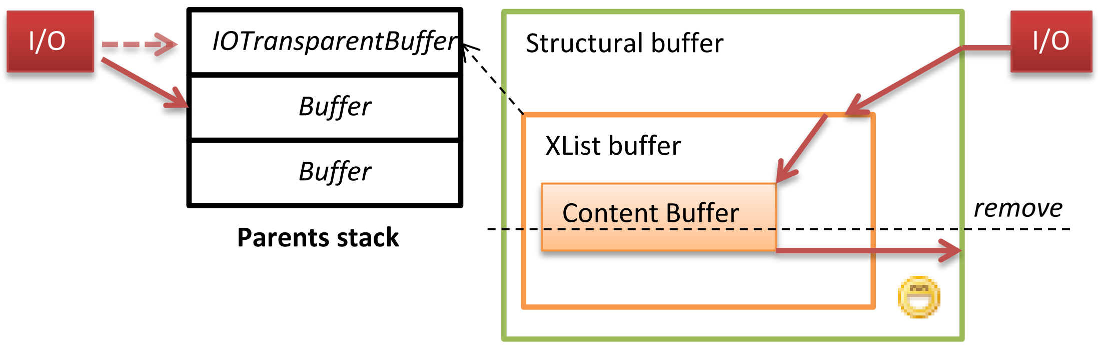


Figure 4-74 Transparent buffer type marker for "virtual" hierarchies

4.4.3.2 XML I/O

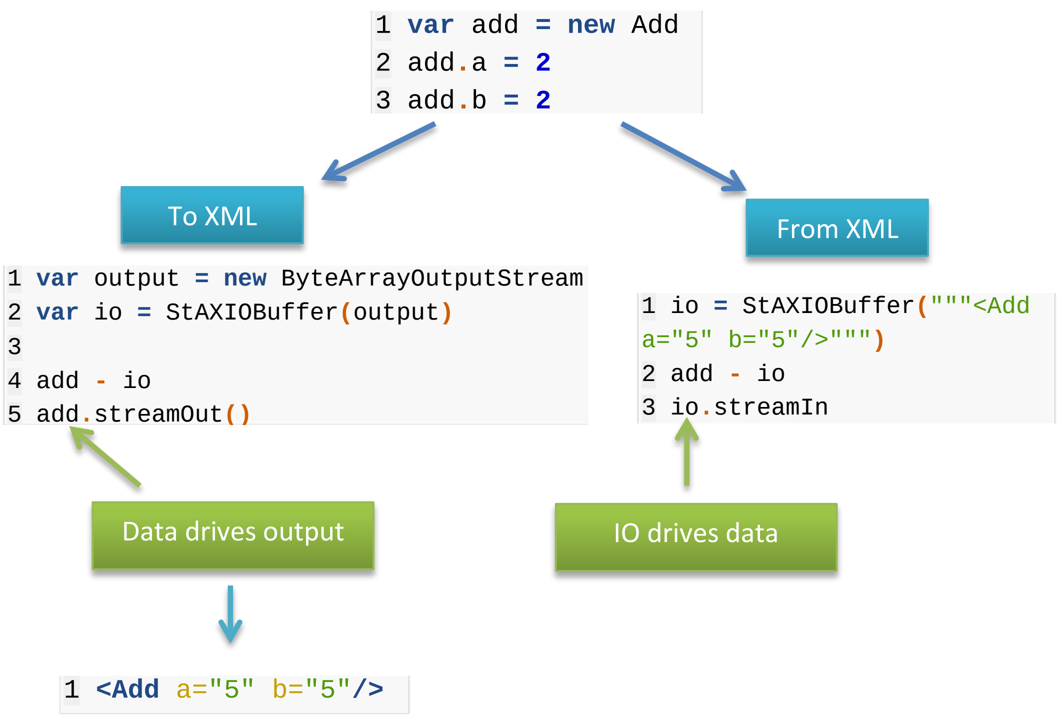
A common parsing strategy for XML documents is called event-based parsing. The XML parser produces events to notify the caller when structural character sequences are met, like element open/close, attribute, text content etc... We already presented in 4.4.2.1.2 the mapping between those event types and the DataUnit configuration.

The default XML IOBuffer implementation which is provided in OOXOO's standard library uses the standard Java Stax parser, and simply converts parsing events to DataUnits, or DataUnits to write events for the serialising side.

Figure 4-75 shows both XML output and parsing usage for a simple Add element. The *IOBuffer* called *StAXIOBuffer* is used to write out or parse the XML. It is appended to the *Add ElementBuffer* chain before *streamOut* or *streamIn*, and automatically disappears afterwards.

Repository: thesis
Path: example-project/src/main/scala/ooxoo/simple/XMLIO.scala

Repository: ooxoo-core
Class: com.idyria.osi.ooxoo.core.buffers.structural.io.sax. StAXIOBuffer



```
var add = new Add
add.a = 2
add.b = 2
```

```
var output = new ByteArrayOutputStream
var io = StAXIOBuffer(output)

add - io
add.streamOut()
```

```
io = StAXIOBuffer("""<Add a="5" b="5"/>""")
add - io
io.streamIn
```

```
<Add a="5" b="5"/>
```

Figure 4-75 Data to XML and reverse

4.4.3.3 JSON I/O

Repository: ooxoo-core
Class: com.idyria.osi.ooxoo.lib.json.JsonIO

The JSON format, standardised in [63] by the IETF, is a structured data format representation for the Javascript language, which provides data binding. It is widely used in web application context and its data <-> structure concept makes it a natural candidate for a dedicated I/O layer implementation in OOXOO.

The SerDes process won't be detailed here, but Figure 4-76 presents an interesting use case of this I/O layer combined with the XML one, to allow remote procedure calls from different application sources using divergent data representation format. It is indeed possible to use the same Data definition classes, and use either XML or JSON as serialised representation format.

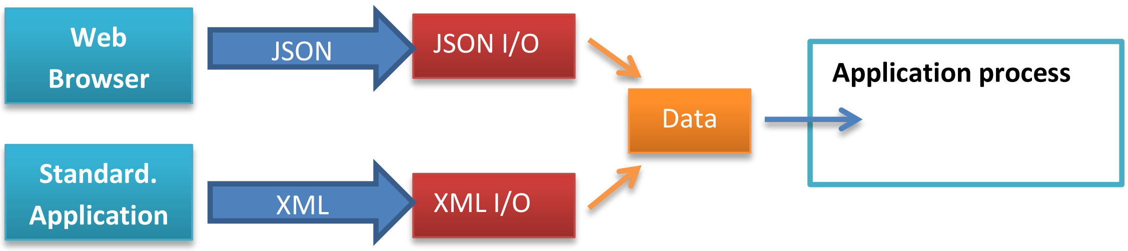


Figure 4-76 XML or JSON formats used with the same Data definition

4.4.4 Register file application interface

In section 4.1 we presented the concept of register files and associated generator software. Among the possible outputs of a register file generator, an XML format mirroring the register file hierarchy and holding various data produced by the tool-chain has be described. We are going to describe here how the OOXOO library was extended to efficiently offer the developer a way to access a device register file.

4.4.4.1 Register file software interaction characteristics

4.4.4.1.1 Read-Modify-Write support

In a register file specification, the addressable memory locations are Registers and RamBlocks entries. Depending on the target architecture, these locations have a certain byte granularity. On a standard 64bit x86 architecture, the minimal host-addressable unit size is 64bit, or 8 bytes (quad-word). However, the digital logic has no specific granularity and very often only relies signals which are a few bits wide.

Registers and RamBlocks can define fields, which are named bit subsets of the host minimal read/write size. To modify a memory location field, the host must thus perform a read-modify-write cycle. Considering that most hardware functions' control and status registers are defined using fields, the developer expects a programming interface that exposes the fields as atomic elements, while hiding the register level, as illustrated in Figure 4-77.

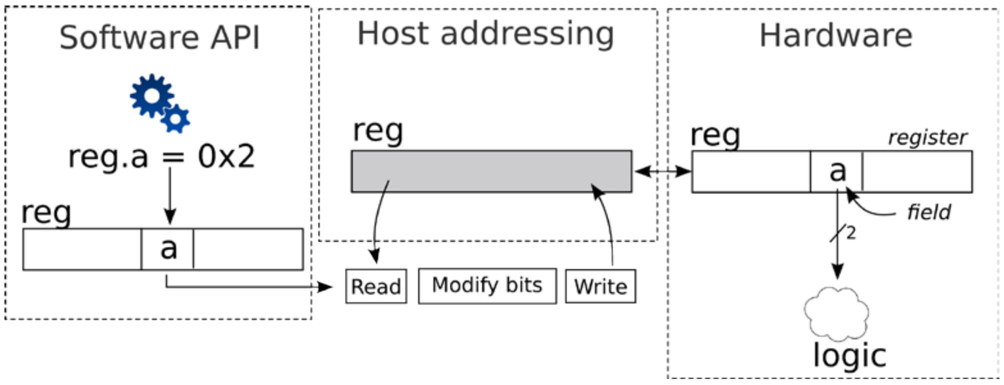


Figure 4-77 Read-Modify-Write for register field

When performing multiple field modifications, especially on the same register, it is not very optimal to trigger a read-modify-write to the device for each change. An interesting option would be to be able to group the modifications and I/O operations. Figure 4-78 shows the read-modify-write flow for both non-optimised, on the left, and optimised I/O on the right. It can be seen that only one read is needed to modify the two fields of the "reg" register, and only two writes are performed at the end to commit the modification on the two registers.

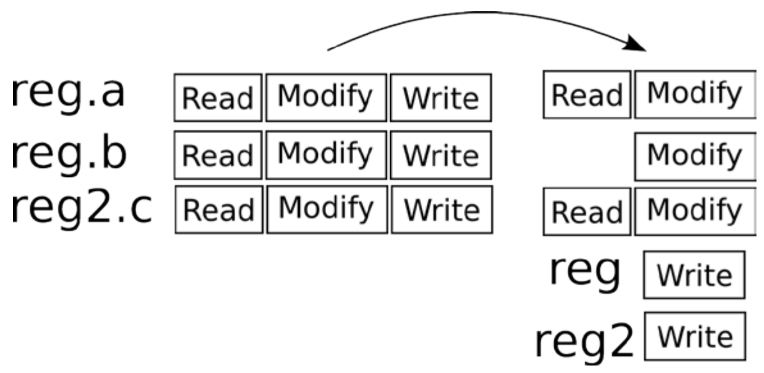


Figure 4-78 Synchronous or Asynchronous Read-Modify-Write

4.4.4.1.2 Scalable Multiple register file access

In some specific application context, it shall be possible for the software to access multiple identical register files located on distinct hardware. To serve this purpose, one instance of a register file object could be created, and configured to reach the desired hardware instance. This solution is however not really scalable, for when the number of supported targets, and the base size of the register file specification grow, the amount of main memory required to hold the descriptors would consequently also grow beyond the acceptable.

The best option would be to keep only one parsed register file description, and decide which target to address at runtime, as illustrated in Figure 4-79.

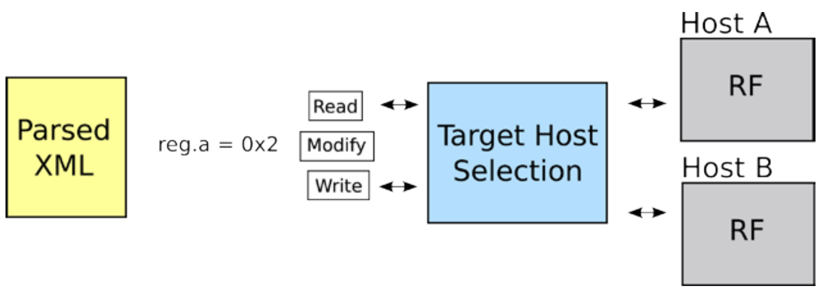


Figure 4-79 One descriptor to many register files

4.4.4.2 The Register file OOXOO interface

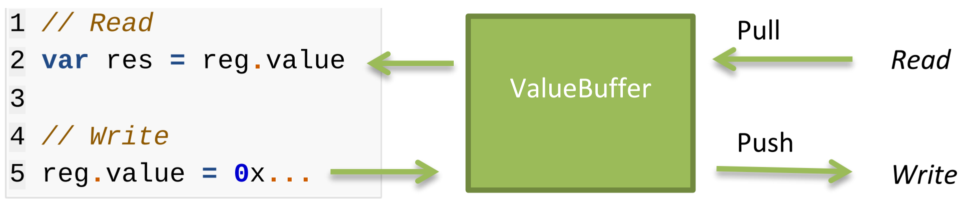
Repository: odfi-rfg, Library: scala-interface.

The first step to implement the register file interface is to mirror in Scala classes the XML structure described in 4.1.4.3. The current implementation is compatible with the older RFS XML format, but the concept has been written with the *RFG* rework in mind, and is valid no matter which version is considered. The following work represents a standard software design, with no specific innovation.

4.4.4.2.1 The value buffer

Once the XML structure has been defined, we must add the placeholders to modify the values of the registers, RamBlock entries and fields. On a standard 64bit x86 host running the Java virtual machine, the maximum width available to represent values is of 64bit, which we can represent using a Long data type. Therefore, a standard LongBuffer typed value field is added to the relevant classes.

Using the value Buffer, in combination with the getter/setter field definition model defined in 4.4.2.3.1, we can now implement an appropriate behaviour when a value is fetched or set through the programming interface. Using the Buffer API, we can perform push and pull operations on the value buffer chain, which could further down be translated to real Read and Write requests (Figure 4-80).



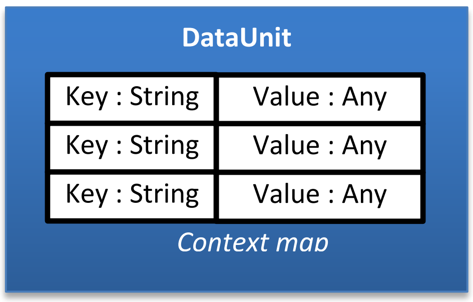
```
// Read
var res = reg.value

// Write
reg.value = 0x...
```

Figure 4-80 Getter/Setter to Push/Pull mapping

In this configuration, the *ValueBuffer* lacks tough the addressing information to issue Push/Pull DataUnits which would be rich enough to be processed. The *ValueBuffer* was therefore improved with a reference to its containing Register or *RamBlock* descriptor, from which it can fetch the address information.

DataUnit Format

The push and pull DataUnits could be issued by the ValueBuffer with the memory address set as a string in the value field (of the DataUnit). However, it has been chosen to add a context map to the DataUnit specification, in which the application can store arbitrary values.



Using a context map allows saving unnecessary string conversion specification and implementation when Buffers belonging to the same application communication with each other. The programming can be made cleaner this way.

Read/Write decoupling

We could argue at this point, that the ValueBuffer could simply directly issue read and writes to a Device interface, instead of first issuing Push and Pulls. This argument makes sense, but the main idea behind the *OOXOO* design is to keep the architecture components decoupled, to allow plugin-in other components, potentially alien to the initial design to support complex scenarios. Quite often, those scenarios are not even know to the specification at implementation time, and will be discovered in a later design stage.

4.4.4.2.2 The Device Buffer

The Device buffer handles issuing the Push and Pull requests from the value buffer to the underlying device. Typically, device connections are handled by a single instance of an interface object in the application space, called a Singleton. The reason is simply that devices are normally single discrete actors in a system, on which I/O operations are performed by a single thread during a synchronised application phase (remember Figure 4-67), and thus only need one interface.

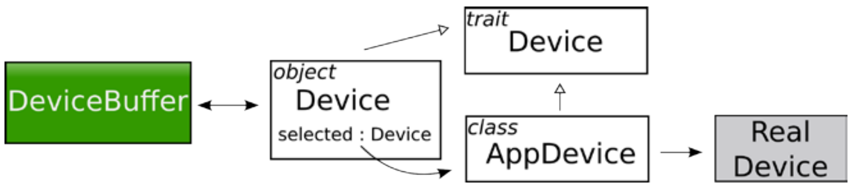
The pendant device interfacing used by the Device Buffer is an Interface + Singleton (*object* in Scala semantic) pair:

- ✓ The Device interface (a Trait in Scala) must be implemented by an application-provided class, and defines the read/write methods targeted at the desired memory location.

```scala
trait Device {

  def open
  def close

  def readRegister( address : Long) : Option[Long]
  def writeRegister(address : Long, value : Long)

}
```

  The read method returns a Scala *Option* object, which can match to the value *None* in case the read should have failed.

- ✓ The Device singleton implements the Device Interface, but forwards the read/write calls to the application-selected interface.



4.4.4.2.3 The Field value

The field objects are finally added to their container Register or RamBlock with a reference to the latter. The “value” getter/setter interface of a field is simply a wrapper that fetches the value of the container, modifies the necessary bits, and set the value back.

4.4.4.3 The generic transaction extension

Two features defined in 4.4.4.1 are still missing:

1. Write grouping for a sequence of Read-Modify-Write
2. Multiple host’s register files access.

RegisterFile interfacing must be considered an I/O process. All read-modify-writes should thus be run sequentially by one Thread. Multi-threading is possible if accessing multiple register file instances on distinct hardware.

Single Threaded, or Thread-Specific bulk update of data looks very much like a Database Transaction mechanism. Basically, a transaction is a phase during a sequential thread execution flow, marked by a start and an end, during which the changes in the state of some persistent data is logged, and submitted to the storage at the end, or discarded. If an error happens during data updating, or if the transaction is discarded for some reason, a rollback operation takes place to restore the pre-Transaction state. This dataflow is presented in Figure 4-81.

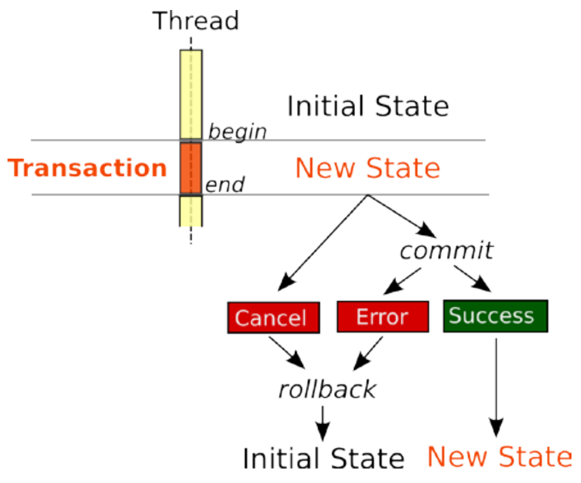


Figure 4-81 Transaction begin - commit - rollback flow

The two missing features can be implemented using a transaction mechanism. The bulk update will be handled by a simple transaction behaviour, and the host selection can happen when starting a transaction.

4.4.4.3.1 The transaction buffer

So far we have defined read/write mapping to a device by specifying two buffers chained together: a *ValueBuffer* and a *DeviceBuffer*. To implement a Transaction mechanism, it is sufficient to catch the Push/Pull DataUnits issued by the ValueBuffer, and release, discard or issue some back to restore state. A *TransactionBuffer* as been developed to support the transaction mechanism, which was inserted between the *ValueBuffer* and *DeviceBuffer*, as shown in Figure 4-82.

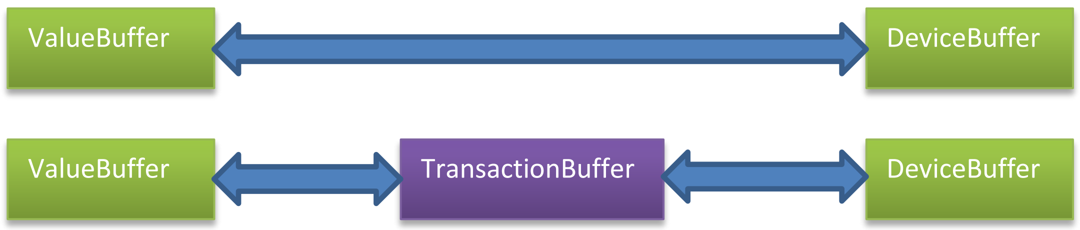


Figure 4-82 Transactional value chain

For example, during read-modify-write cycles, under an active transaction, the behaviour of the push and pull channels will be:

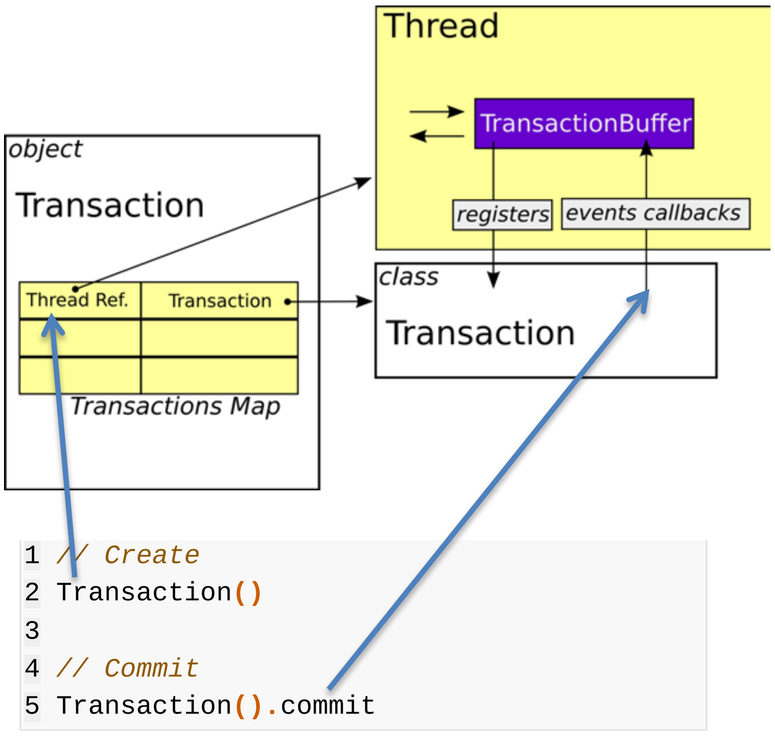
- ✓ Pull:
  - o Initial state: No cached value
  - o If a cached value is available, return it
  - o If no cached value is available, forward
- ✓ Push:
  - o Store the DataUnit
- ✓ Transaction Commit
  - o Forward the Push DataUnit
  - o Discard the cached Pull DataUnit
- ✓ Transaction rollback or cancel
  - o Pull on the right if no cached Pull DataUnit is available
  - o Push to the left the last cached Pull DataUnit or the one just pulled to restore values
  - o Discard the Stored Push

Some additional Transaction states are available for special behaviours like:

- ✓ Stopped/Inactive: Push and Pull are always forwarded
- ✓ Blocking: Push is stored, and Pull return no value (blocked)

4.4.4.3.2 Transaction State management

The transaction state is managed per thread. It is easily implemented using an object (Singleton) called Transaction, on which the user can manage the state, with very simple API calls. The transaction buffers, when triggered on their Push or Pull interface, check if a transaction was setup for the current thread, and if so, registers event listeners on the current Transaction object to be called on state change.



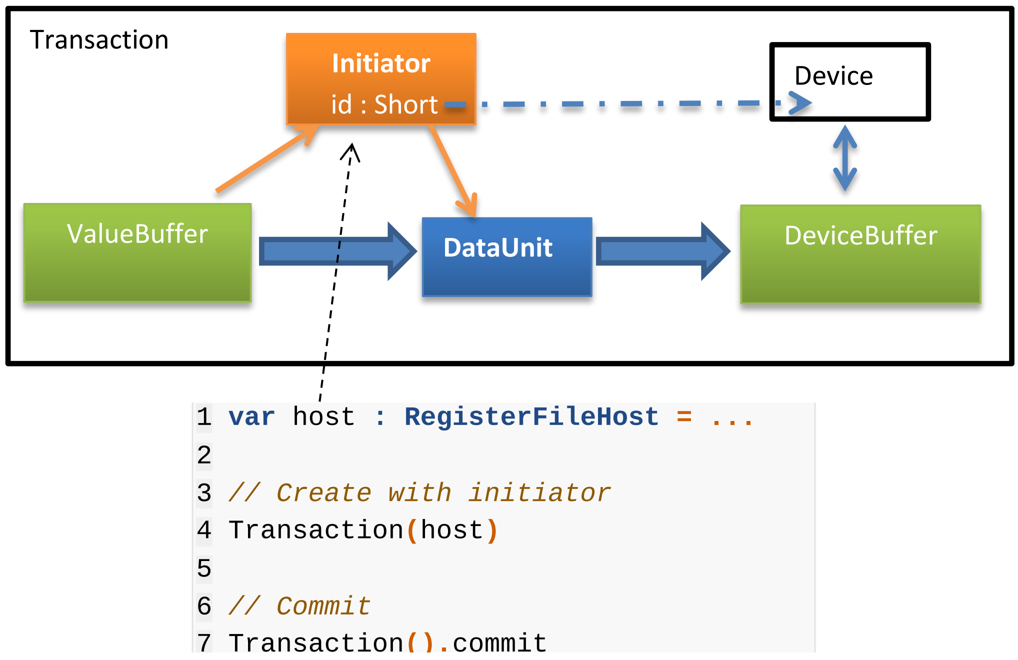
```
// Create
Transaction()

// Commit
Transaction().commit
```

Figure 4-83 Transaction state relation to transaction buffer

4.4.4.3.3 Target host selection: Transaction initiator

The current Transaction stage management API already features nearly everything needed to support multiple register file targets. The transaction class representing the currently setup transaction was improved to be able to hold a reference to an object of any type called an initiator.

The initiator reference can be passed to the Transaction singleton when setting up a transaction. As presented in Figure 4-84, the *ValueBuffer* enriches the Push/Pull DataUnits with the transaction initiator reference, if it is of a certain type called *RegisterFileHost*. A register file host holds an extra piece of information called ID (of type Short), which is passed the Device layer's read and write functions to choose a target host if possible.



```
var host : RegisterFileHost = ...

// Create with initiator
Transaction(host)

// Commit
Transaction().commit
```

Figure 4-84 DataUnit enrichment with initiator

4.4.4.4 *Final ValueBuffer configuration*

To sum-up the register file interface coupled with the transaction mechanism for the value representation, Figure 4-85 shows the class hierarchy and Buffer chain which is created per default for all value buffers. The whole setup is grouped under a class called *RegisterTransactionBuffer* in the source code.

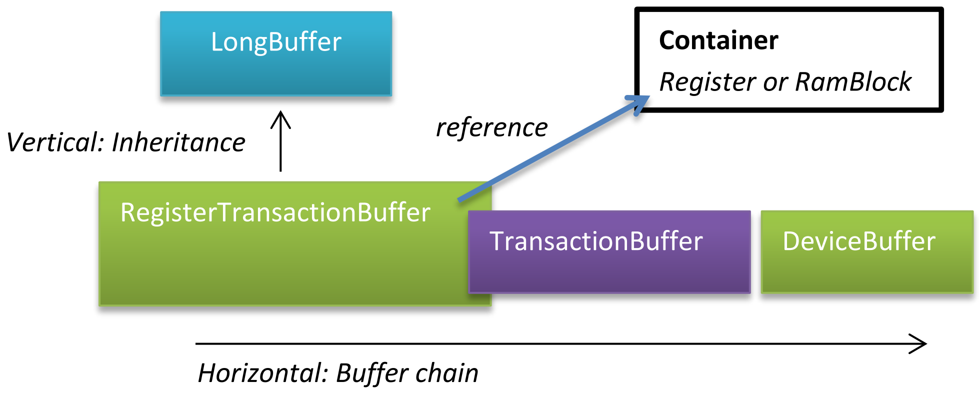


Figure 4-85 Value buffer final inheritance, buffer setup and container reference

4.4.5 Conclusion

In this chapter we presented a flexible architecture for structured data binding, designed to reach heterogeneous setup of application components. Although the historical design was meant for XML-based applications, it appears that the generic nature of the data structures and their interconnection methodology, make it suitable and easy to adapt for various kind of usages.

A concrete application example has been presented, which mixes XML binding with a hardware register binding, using the generic semantic and interconnection methodology presented in the global architecture definition. This helped minimizing the implementation effort.

During the register file interface implementation, we prove that the Buffer chain idea, correctly used to decouple application layers, could allow integrating external behaviours to solve internal issues discovered late in the design phase. This was the case when we used the generic Transaction mechanism to add vital features to the interface behaviour, without modifying it consequently.

After nearly 10 years of improvements and redefinition of the OOXOO library’s behaviour, we could show that generic architecture definitions, extracted from a concrete problem statement (XML-data binding) can lead to very reusable software components, and should never be neglected when designing new applications…even if it comes at the cost of a top-hill analysis overhead.

It is also interesting to note that the presented architecture was first designed when the Register file interface software presented in 4.1 did not exist, and where not even planned.

5 Conclusion and Outlooks

To trace a path toward raising the abstraction level in hardware-software co-designs, this work was divided in two main themes:

- ✓ Language design
- ✓ Applications to integrated circuit design flows.

A set of programming methodology building blocks was first presented. Inspired from the convergence of traditional imperative and functional programming, they set ground for embedded domain specific language (EDSL) design. While standard domain specific languages allow creating software interfaces which are closely modelling specific application fields, embedding them in host language allows lightening their setup and maintenance costs, and makes them very interoperable within the hosting technology bounds.

As a consequence, the host language choice is critical, and should ideally be adapted or adaptable to both the existing software environment and to the criterions of EDSL design presented in 2.4.

Electronic Design Automation tools widely support the TCL language, which is well known for being minimalistic (everything is a list in TCL), but less for being very open to deep customisation. To be able to use domain specific languages for digital design flows, we thus tried to find a way to bring a functional programming flavour to TCL scripts .We successfully reached this goal by presenting a library for closure-based programming in 3.2, and implementing an EDSL design methodology in 3.3.

The results of EDSL design in TCL are shown to be promising. The code can be structured in a very clear way, and be spared from too many language syntax-specific keywords and characters. Moreover, we prove that an EDSL can emerge directly from the data structures definitions, which enables creating a new language within a few minutes, and without any external tool-chain. One simply opens a new script and load two libraries.

We then introduced some applications which make a concrete and extensive usage of TCL based EDSL design. The selected work spans along the digital hardware design flow: from design input (register file 4.1) to physical implementation (floorplanning 4.2) and high level integration (part language 4.3, OOXOO 4.4). Applications sharing the same background (TCL scripts) were shown to be very interoperable, while less interoperable components (register file script and user-space software binding) need to exchange data using serialised character data (XML files).

Altogether, we can say that we succeeded in proposing a way to really bridge the gap between abstraction levels and implementation specificities while keeping design flows consistent across those abstraction levels.

To go further in this direction, we will conclude by exploring a possible future application to hardware description language.

5.1 Abstraction in hardware description languages

The presented applications mainly focus on the technology implementation (manufacturing) and integration steps of design flows. Digital circuits are however specified using special hardware description languages (HDL), the most widespread one being Verilog and VHDL.

Those languages are domain specific, and aim at providing an abstraction level suitable to the description of digital synchronous or asynchronous circuits. In regard to the presented language design methodologies, it appears possible to create an EDSL which would provide a programming interface to create an in-memory representation of a digital circuit. This approach presents the advantage of offering the full flexibility which has been shown for this work's applications, like custom abstraction level definition, tree transformation, optimisation, while preserving the possibility of full circuit description.

The "programming language" nature of HDL is a challenge for the creation of an EDSL. Indeed, defining a programming language inside another programming language leads to name clashes for control structures and operators. Additionally, the language acceptance is an important issue. If the designers must learn how to use a library to manually instantiate and connect together circuit elements, the benefits of the abstraction level could be lost to the usage complexity. For example, as presented in Figure 5-1, the Verilog language uses the C-like "if" syntax to describe value multiplexing.

```
wire a;
wire b;
wire res;
if (a & b) begin

    res = 3;

end
else begin

    res = 2;
end
```

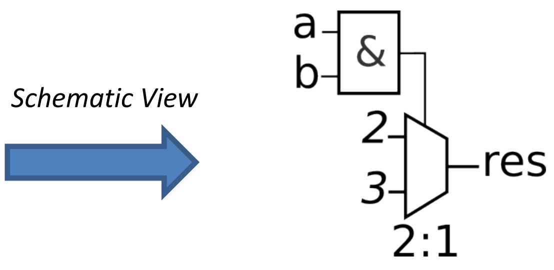
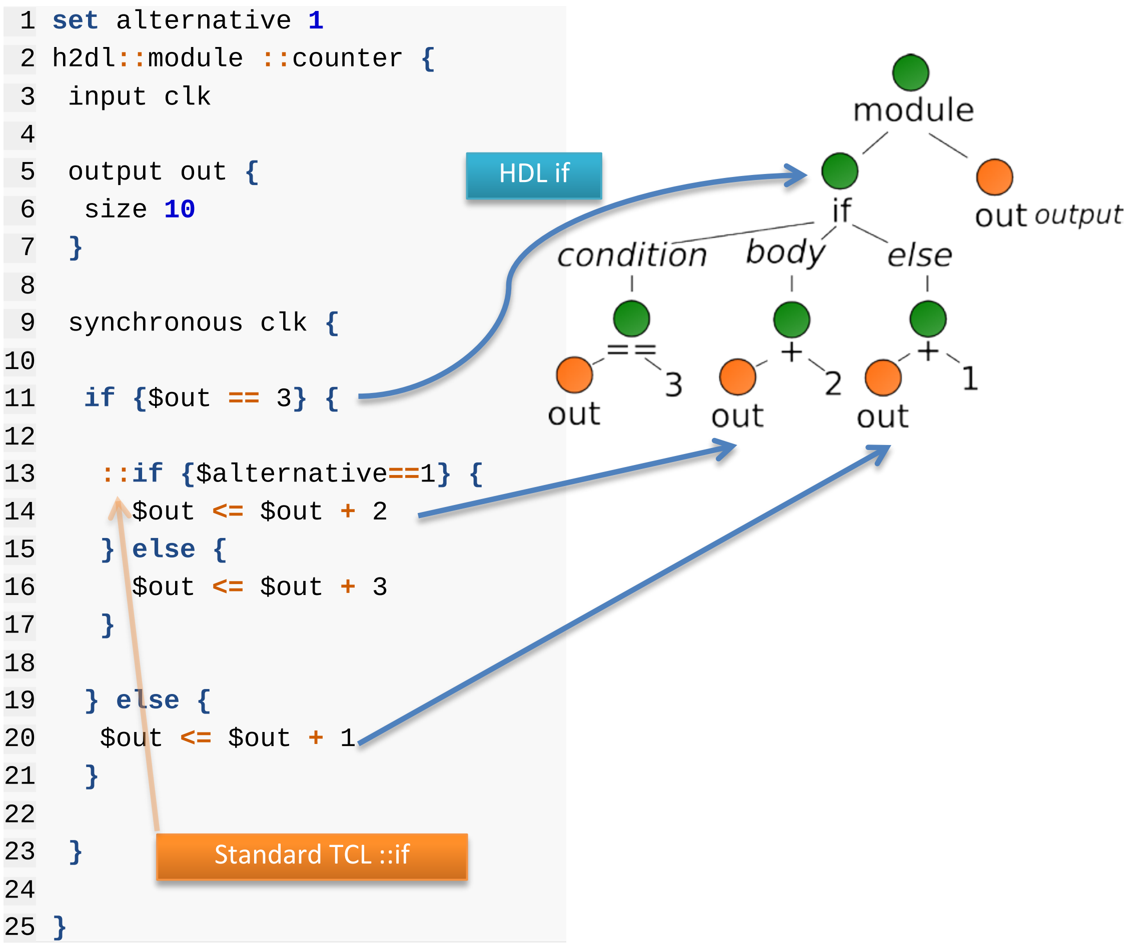


Figure 5-1 Verilog if to multiplexer mapping

Porting this syntax in an EDSL could become challenging, because a control structure like "if" is typically a core operator in the language parser, and can't be overloaded. Some existing projects try to implement an EDSL applied to hardware design. The two most notable one are MyHDL [64] written in Python and Chisel [65] integrated in Scala. They both work around this challenge in two different ways:

- ✓ Chisel adds its own control structures. The “if” keyword is implemented using the “when” function definition. The main drawback of chisel is its highly object oriented nature in Scala, which forces the usage of classical Scala programming syntax and code overhead, which is not desirable.
- ✓ MyHDL uses Python decorators. A decorator is a function which can process the Abstract Syntax Tree of another function definition. This way MyHDL can reuse the Python syntax to generate an in-memory circuit description, but forces the user to write HDL code in special annotated functions, and limits standard Python usage at this level.

How would then behave our TCL EDSL strategy in this context? Surprisingly, it appears possible to come out with an elegant solution. If example of an “if” control structure override was already presented in 3.1 Example 3-4, we can try to define a few elements of language as we did during this work. We tried to stay close to Verilog, and managed to produce a result presented in Figure 5-2.



```
set alternative 1
h2dl::module ::counter {
 input clk

 output out {
  size 10
 }

 synchronous clk {

  if {$out == 3} {

   ::if {$alternative==1} {
    $out <= $out + 2
   } else {
    $out <= $out + 3
   }

  } else {
   $out <= $out + 1
  }

 }

}
```

Figure 5-2 Example of an HDL EDSL in TCL

On the left, a now familiar TCL EDSL syntax with its associated generated in-memory abstract syntax tree on the right. We can see that it is possible to alternate host language control structures (line 13) and HDL control structures (line 11), in order to define the elaboration-time behaviour, and the pure digital logic behaviour. This syntax should be acceptable to learn for hardware designer, as it stays close to a classical Verilog Syntax. Moreover, it would be very easy to create different HDL control structures, for example some closer to VHDL.

Finally, another possible example presented in Figure 5-3 could be a tight integration of the register file language inside this HDL DSL. In such a case, each HDL hierarchy could define its own register file locally, and the final top-level hierarchy would gather the register files in the hierarchy, and adapt the circuit depending on the final requirements: The RFG registers can be either be moved across levels, or new connection wires created.

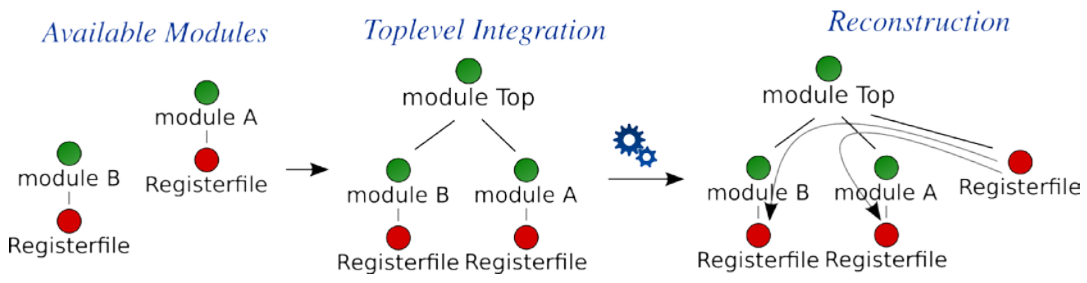


Figure 5-3 Register file generator and TCL HDL language integration examples

5.2 Design flow libraries open sourcing

In accordance to the software reusability concepts set along this work, all the sources are available under a General Public License (GPL), and opened to usage and improvement. Details are provided in Appendix A.

Bibliography

[1] "Moore's Law." [Online]. Available: http://en.wikipedia.org/wiki/Moore's_law.

[2] Satya Prakash Singh and Jayant V. Deshpande, "Break-Even Point," *JSTOR: Economic and Political Weekly, Vol. 17, No. 48 (Nov. 27, 1982), pp. M123+M125+M127-M128*, 1982. [Online]. Available: http://www.jstor.org/discover/10.2307/4371597?uid=3738016&uid=2&uid=4&sid=21103704789777. [Accessed: 20-Mar-2014].

[3] "Break-Even Analysis." [Online]. Available: http://simplestudies.com/accounting-cost-volume-profit-analysis.html/page/8.

[4] "Binary Code." [Online]. Available: http://en.wikipedia.org/wiki/Binary_code.

[5] A. Aho, *Compilers : principles, techniques, & tools*. Boston: Pearson/Addison Wesley, 2007.

[6] B. W. Kernighan, *The C Programming Language*. Prentice-Hall, 1978, p. 228.

[7] "GCC GNU Compiler." [Online]. Available: http://gcc.gnu.org/.

[8] "x86-64." [Online]. Available: http://en.wikipedia.org/wiki/X86-64.

[9] B. J. Cox and A. J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Publishing Company, 1991, p. 270.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software (Google eBook)*. Pearson Education, 1994.

[11] A. Church, "An Unsolvable Problem of Elementary Number Theory," *Am. J. Math.*, vol. 58, no. 2, p. 345, Apr. 1936.

[12] A. M. Turing, "Computability and λ-definability," *J. Symb. Log.*, vol. 2, no. 04, pp. 153–163, Mar. 1937.

[13] W. Kluege, *The organization of reduction, data flow, and control flow systems*. MIT Press, 1992.

[14] M. Schönfinkel, "Über die Bausteine der mathematischen Logik," *Math. Ann.*, vol. 92, no. 3–4, pp. 305–316, Sep. 1924.

[15] H. B. Curry, "An Analysis of Logical Substitution," *Am. J. Math.*, vol. 51, no. 3, p. 363, Jul. 1929.

[16] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978.

[17] J. McCarthy, M. L. Minsky, and N. Rochester, "The LISP Programming System," *Mass. Inst. Tech. Res. Lab. Electron. Q. Prog. Rep.*, no. 53, pp. 124–152, 1959.

[18] G. L. Steele, *Common LISP: the language*. Digital press, 1990.

[19] G. L. Steele Jr and G. J. Sussman, "The revised report on SCHEME: A dialect of LISP," 1978.

[20] R. Hickey, "The Clojure programming language," in *Proceedings of the 2008 symposium on Dynamic languages - DLS '08*, 2008, pp. 1–1.

[21] J. K. Ousterhout, "Tcl: An Embeddable Command Language." .

[22] Y. U. D. of Computer Science, P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, and others, *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language. Version 1.1*. 1991.

[23] J. Armstrong and S. Virding, "Erlang-an experimental telephony programming language," *XIII Int. Switch. Symp.*, 1990.

[24] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An Overview of the Scala Programming Language," no. Section 5, 2004.

[25] J. Armstrong, R. Virding, C. Wikström, and M. Williams, "Concurrent Programming in ERLANG." 1993.

[26] R. D. Greenblatt, T. F. Knight, J. T. Holloway, and D. A. Moon, "A LISP machine," in *Proceedings of the fifth workshop on Computer architecture for non-numeric processing - CAW '80*, 1980, vol. 15, no. 2, pp. 137–138.

[27] A. C. Kay, "The reactive engine," 1969.

[28] B. K. J. Beazley David, "Python Cookbook, 3rd Edition - O'Reilly Media." [Online]. Available: http://shop.oreilly.com/product/0636920027072.do?green=433478B6-29BD-57D3-040C-F811D6E1EDE0&intcmp=af-mybuy-0636920027072.IP. [Accessed: 28-Mar-2014].

[29] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in action*. 2007.

[30] "Rust." [Online]. Available: http://www.rust-lang.org/.

[31] "EPFL LAMP." [Online]. Available: http://lamp.epfl.ch/.

[32] J. Moses, "The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem," *ACM SIGSAM Bull.*, no. 15, pp. 13–27, Jul. 1970.

[33] P. J. Landin, "The Mechanical Evaluation of Expressions," *Comput. J.*, vol. 6, no. 4, pp. 308–320, Jan. 1964.

[34] K. Erk and L. Priese, “Theoretische Informatik,” *Eine umfassende Einf{ü}hrung, Berlin-Heidelberg-New York*, 2000.

[35] D. Grune and C. J. . Jacobs, *Parsing Techniques: A Practical Guide*, Second Edi. Springer, 2007.

[36] J. Backus, “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference,” *… Int. Comference Inf. …*, 1959.

[37] T. Parr and R. Quong, “ANTLR: A predicated LL (k) parser generator,” *Softw. Pract. Exp.*, vol. 25, no. June 1994, pp. 789–810, 1995.

[38] T. Parr and K. Fisher, “LL (*): the foundation of the ANTLR parser generator,” *ACM SIGPLAN Not.*, pp. 425–436, 2012.

[39] M. Odersky, *Programming in Scala*, Second Edi. Artima Press, 2011.

[40] P. Hudak and N. Haven, “Modular Domain Specific Languages and Tools.”

[41] C. Hofer, K. Ostermann, T. Rendel, and A. Moors, “Polymorphic embedding of dsls,” *Proc. 7th Int. Conf. Gener. Program. Compon. Eng. - GPCE ’08*, p. 137, 2008.

[42] “TCL/Tk Developer Web Site.” [Online]. Available: https://www.tcl.tk/.

[43] M. S. Braverman, “CASTE: A class system for Tcl,” in *Proceedings of the 1st Tcl/Tk Workshop, Univ. of Calif. at Berkeley,(June, 1993)*, 1993.

[44] G. Neumann and S. Sobernig, “An Overview of the Next Scripting Toolkit Next Scripting Framework (NSF),” *Tcl/Tk 2011 Conf.*, no. October, 2011.

[45] G. Neumann and U. Zdun, “XOTCL, an object-oriented scripting language,” 1998.

[46] M. Nüssle, “ACCELERATION OF THE HARDWARE - SOFTWARE INTERFACE OF A COMMUNICATION DEVICE FOR PARALLEL SYSTEMS,” 2008.

[47] N. M. Burkhardt, “A Hardware Verification Methodology for an Interconnection Network with fast Process Synchronization,” 2012.

[48] B. U. Geib, “Hardware Support for Efficient Packet Processing,” 2012.

[49] U. Brüning, “HTAX : A Novel Framework for Flexible and High Performance Networks-on-Chip Heiner Litz Holger Fröning.”

[50] C. Leber, “Efficient Hardware for Low Latency Applications,” 2012.

[51] “The Extensible Stylesheet Language Family (XSL).” [Online]. Available: http://www.w3.org/Style/XSL/. [Accessed: 01-May-2014].

[52] O. Chafik, “BridJ.” [Online]. Available: https://code.google.com/p/bridj/. [Accessed: 14-Jul-2014].

[53] K. H. Jürgen Döllner, “A Generalized Scene Graph.”

[54] M. Schsnfinkel and S. Beschr, “l ber die Bausteine der mathematischen Logik.,” 1920.

[55] “Extensible Markup Language (XML) 1.0 (Fifth Edition).” [Online]. Available: http://www.w3.org/TR/REC-xml/.

[56] E. J. O’Neil, “Object/relational mapping 2008: hibernate and the entity data model (edm),” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD ’08*, 2008, p. 1351.

[57] “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.” [Online]. Available: http://www.w3.org/TR/xmlschema11-1/.

[58] “W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes.” [Online]. Available: http://www.w3.org/TR/xmlschema11-2/.

[59] J. Fialli and S. Vajjhala, “The Java architecture for XML binding (JAXB),” *JSR, JCP, January*, 2003.

[60] J. F. Kohsuke Kawaguchi, Sekhar Vajjhala, “The Java^{TM} Architecture for XML Binding (JAXB) 2.2 Final,” 2009.

[61] “XMLBeans.” [Online]. Available: http://xmlbeans.apache.org/.

[62] “JibX.” [Online]. Available: http://jibx.sourceforge.net/.

[63] D. C. <douglas@crockford.com>, “The application/json Media Type for JavaScript Object Notation (JSON).”

[64] J. I. Villar, J. Juan, M. J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, “Python as a hardware description language: A case study,” in *2011 VII Southern Conference on Programmable Logic (SPL)*, 2011, pp. 117–122.

[65] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanovi, “Chisel : Constructing Hardware in a Scala Embedded Language.”

[66] “Letter from Burnaburiash to Amenhotep IV.” [Online]. Available: http://www.britishmuseum.org/explore/highlights/highlight_objects/me/l/clay_tablet_letter,_egypt.aspx.

Appendix A. Software setup

LiveRun online access

The TCL environment is available for experimenting through a Scala-based web application available online:

http://www.idyria.com/~rleys/LiveRun/index.view

The scripts presented in this thesis are listed on the web page to be loaded and executed.

The TCL runner embedded in the web application features a full stream redirection, and will offer any generated files to be viewed online.

Thesis Sources

- ✓ The sources of this work are available on the repository located at: https://bitbucket.org/richnou/phd
- ✓ The branch to checkout is named: final

TCL Setup and bootstrapping

To run all the examples without any issues across all TCL versions, the best option is to use a standard TCL 8.6 distribution. The thesis repository provides a Makefile which downloads and installs locally TCL 8.6 along with the Next Scripting Framework.

To proceed, make sure you have a GCC compiler available (Linux, or windows msys should do), and follow these steps:

```bash
#!/bin/bash

# Clone Thesis repository
git clone https://bitbucket.org/richnou/phd rleys-phd -b final

# Change directory
cd rleys-phd

# Consult README for system requirements
less README

# Build
cd external
make all

# Source environment
source odfi-manager/setup.linux.bash
```

Once done, before running any TCL script, source the setup script which will update your command line environment to use the newly build TCL interpreter.

```
# Source environment
source external/odfi-manager/setup.linux.bash
```

Repositories

The sources of all the examples provided in this thesis, as well as the full sources of all projects are available in a set of GIT repositories. Some text boxes are included at relevant locations throughout this work to direct the reader to the correct repository and file path. Here is a list of all the repositories along with their locations

Thesis Repositories

Repository	Project / Content	Locations
thesis	Thesis sources	https://bitbucket.org/richnou/phd
odfi-dev-tcl	TCL common library	https://github.com/unihd-cag/odfi-dev-tcl
odfi-rfg	Register File generator	https://github.com/unihd-cag/odfi-rfg
odfi-dev-tcl-scenegraph	Scenegraph logic for SVG and flooorplaning API	https://github.com/unihd-cag/odfi-dev-tcl-scenegraph
odfi-dev-hw	Part design language and Hardware design utilities	https://github.com/unihd-cag/odfi-dev-hw
ooxoo-core	OOXOO XML binding library	https://github.com/richnou/ooxoo-core
odfi-tcl-integration	TCL 8.6 distribution and scala interface	https://github.com/richnou/odfi-tcl-integration
odfi-modules-manager	Manager tool to install ODFI libraries	http://github.com/richnou/odfi-manager.git

LiveRun Website Repositories

Repository	Project / Content	Locations
wsb-core	Messaging processor and router	https://github.com/richnou/wsb-core
wsb-webapp	Webapplication Framework Logic for wsb-core	https://github.com/richnou/wsb-webapp
vui-core	Virtual UI language for HTML Scala EDSL	https://github.com/richnou/vui-core
ooxoo-core	OOXOO XML binding library (With JSON IO Layer)	https://github.com/richnou/ooxoo-core
odfi-tcl-integration	TCL native interface library for Scala projects	https://github.com/richnou/odfi-tcl-integration