Multi-Language Software Development

with

Marcel Toele

Summer 2007

Master's Thesis in Computer Science
FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
University of Amsterdam
Supervisor: prof. dr. P. Klint

Abstract

Software projects tend to grow to exist of large quantities of program code. Most of this code will probably be support code for secondary features. It is known that some programming languages are more concise than others (i.e. they require less lines of program code to implement a function point). Most notably, *higher-level "dynamic"* programming languages tend to be more concise than *lower-level "static"* programming languages, making them suitable to be used for such support code. However, *lower-level* programming languages certainly have advantages of their own, most notably in terms of execution speed, and tighter integration with the underlying operating system and hardware. Many attempts exist to make software components, written in different programming languages, interoperate with eachother. We divide these into two groups, *distributed* and *non-distributed* approaches. Our research has focussed on the non-distributed approaches. To interface with an existing software component (a shared library in our case), often a solution that uses such a non-distributed approach requires a seperately compiled shared library, which is the result of extensive programming, or, the solution uses a specification. The programmatic solutions that we have come across, because of their programmatic nature, made it hard to establish the interface, and, again because of their programmatic nature, they bare the risk of becoming very hard to maintain. The solutions that use a specification are easier to use and maintain, but are often too simplistic or incomplete to effectively interface with any shared library, regardless of the way such library is intented to be used. In this thesis, we argue that it is beneficial to interface the *higher-level "dynamic"* programming languages to *lower-level "static"* programming languages, and we propose a way for interfacing these in such a way, that the interface is easy to establish, maintainable, efficient in use, and effective in all required circumstances. One of the key benefits is that we attempt to interface the languages in run-time, without requiring a separately compiled "binding" library, making it a more natural interfacing approach for an interpreted, *higher-level*, *"dynamic"* programming language.

Table of Contents

Chapter 1

Introduction

A trend that has started ever since the 1970s and 1980s, but at the dawn of digital convergence, where almost every electrical appliance is equipped with a more than adequate processing unit, software is getting more important than ever.

As the processing power increases, so do the sizes of the software products that are running on these appliances, with ever growing feature lists to suit the specific preferences of the individual.

In a world where everything competes with everything else, the big challenge for the future of software development is, how to produce these large feature-rich software products effectively and efficiently.

This thesis addresses a specific problem domain that can be placed directly within this context, albeit at a level that is normally not visible to the end-user. In this thesis we are going to address *multi-language software development*, we are going to seek answers to such questions as:

- Why do we need multi-language software development?
- How may large, feature-rich, software products benefit from it?
- How can we make a multi-language interface easy to construct, maintainable and easy to use?

This thesis does not pretend to pose *the* ultimate solution to the above problems, but we think that our solution is a worthy contribution to the software development community.

Focus of This Thesis

This thesis focusses on the discussion of software development using *general purpose*, *imperative*, programming

languages. It may be obvious that *domain specific* languages will yield higher productivity within their *specific* domains, however, they are less applicable for generic use. We will also not focus on functional programming languages, logic programming languages, etc.

How This Thesis Is Organized

This thesis has become quite an extensive work. In order to discuss all the material in a structured manner, we have organized this thesis as follows:

Chapter 1. Introduction

This chapter.

Chapter 2. Why Multi-Language Development?

In chapter 2 we start by introducing some terminology, to make it easier to communicate about the subject of multi-language software development. Then we try to answer the question: Why multi-language software development? We do this by outlining three scenarios in which we think that, under the right circumstances, it is beneficial to use multi-language software development.

Chapter 3. Related Work

To understand where our solution fits in with the rest of the multi-language software development solutions, we shall discuss some related work and provide a categorization for these solutions in chapter 3.

Chapter 4. Goals and Requirements

Then, in chapter 4, we shall establish a list of goals and requirements. As input to this list we will use the three scenarios from chapter 2 as well as information from existing literature.

Continued on next page...

Chapter 5. Language Selection

After we have established our list of goals and requirements, we do a language selection in chapter 5. First we pose selection criteria. These are influenced by the three scenarios from chapter 2 and by the goals and requirements from chapter 4. Then we do the actual language selection, explain our choice, and discuss several considered alternatives that were discarded, and for what reasons we discarded them. The chapter continues with a brief discussion of existing work. We end the chapter by formulating our ***main thesis***, which is based on the results from the language selection and the goals and requirements that were posed in chapter 4.

Chapter 6. DLX

In chapter 6, we pose our solution. We do this in eight sections. First we pose the *field of application* for our solution, after which we explain our *design decisions* with respects to the goals and requirements set out in earlier chapters. Then we describe *the architecture* that lies at the heart of our solution. This is followed by a rather elaborate discussion on *type mapping* between the two languages, explaining how we do the mapping, and why it has proven to be such a challenge at times. Then we discuss the *memory layout reconstruction algorithm* that makes our solution work at run-time, which is followed by a section on *error handling*. We conclude the chapter with sections on *penalties* and *advanced topics*, such as automating the multi-language interface construction (Auto/DLX), memory management and threading.

Chapter 7. Experiments

To assess the results of our work when it is used, we conduct a series of experiments in chapter 7. The experiments range from laboratory experiments for correctness, to case studies, that are conducted out in the field with real world examples.

Chapter 8. Conclusion and Future Work

We conclude this thesis in chapter 8, where we discuss our conclusion and present starting points for future work.

Chapter 2

Why Multi-language Development?

In this chapter we will outline three scenarios in which, we think, it would be beneficial to use some form of multi-language software development.

In each of the three scenarios a potential benefit will be highlighted. The three benefits that we will focus on in the scenarios are:

1. A potential increase in software development productivity.

2. A potential benefit from multi-disciplinary software development.

3. An increase in effectiveness of any single programming language.

2.1 Terminology

But before we start, what actually is multi-language software development? Let us first introduce some terminology.

2.1.1 Multi-language Interoperability

The problem of cooperation among software modules of different (programming) languages is called the *multi-language interoperability* problem (Barrett et al. [20]). Interoperability problems can also occur in systems of a single programming language, where differences in dialects or hardware platforms cause interoperability issues. Such problems are beyond the scope of this thesis.

2.1.2 Multilingual versus Polylingual Interoperability

In [20] Barret, Kaplan and Wileden define two distinct forms of multi-language interoperability. *Multilingual* interoperability, "in which a component that has been written in one programming language needs to access components (subprograms, data objects, etc.) written in a single other programming language", versus *Polylingual* interoperability, "in which a component written in one programming language needs to *uniformly* interact with a set of components written in two or more different languages (which may or may not include the languages that was used for the first component)".

We underline the word *uniformly* here, because the key distinction is made in the fact that two or more components which are similar in functionality (for example two personnel records, implemented in two different languages), need to be accessed in a uniform way from a single (other) language.

2.1.3 Separation of Concerns

For the definition of the *separation of concerns*, we think [16] gives a concise and workable definition:

> *"In computer science, separation of concerns (SoC) is the process of breaking a program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program. Typically, concerns are synonymous with features or behaviors."*
>
> *"Progress towards SoC is traditionally achieved through modularity and encapsulation, with the help of information hiding."*

2.1.4 Information Hiding

In [12], we have found an agreeable definition for *information hiding*:

> *"[Information Hiding is a principle for] ... the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Protecting a design decision involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change)."*

In addition to this, we think that information hiding, in the particular case of multi-language software development, can also be applied to hide the fact that a different programming language is used for a particular software component or module.

2.1.5 Encapsulation

In computer science, *encapsulation* is a language feature for bundling related data objects and or functions within a single language construct. For instance, *records*[1] in some programming languages allow the

[1] In C these are called structs.

bundling of related data objects (attributes) under the umbrella of a new type.

In object oriented programming languages, classes and objects provide this umbrella to encapsulate both attributes, and the operations on these attributes (usually called functions, member functions, or methods). In procedural programming languages *procedures* allow to encapsulate individual instructions.

2.1.6 Holy Grail

In some religions, the *holy grail* refers to the chalice that Jesus Christ supposedly used during his last supper, granting eternal life to whomever drinks holy water from it, or so the story goes... no proof of the existance nor of its alleged super power was ever found or presented.

In non-religious context the *holy grail* refers to the greatest good or the ultimate goal that is attempted to be reached, but usually cannot actually be reached due to practical or theoretical constraints.

In multi-language software development, there also is such a holy grail: *Seamlessness*.

2.1.7 Seamlessness

Interoperability is called *seamless*, if developers using a software module do not need to be aware of the implementation details (such as the used programming language, etc.)[20]. *Seamlessness* is the degree of seamless interoperability that has been achieved.

Seamlessness depends on several factors:

- *Encapsulation* - The ways in which different programming languages are able to bundle related data objects and/or instructions has to be mapped sensibly.
- *Information hiding* - For instance hiding certain implementation details that are explicit in one language but transparent in another (e.g. use of *pointers* is explicit in C++ but it is transparent in Java).
- *Intelligent type system mapping* - Types of data objects in one component written in one language have to be sensibly mapped onto corresponding types of another programming language (e.g. C has several integer types each with an explicit maximum capacity; in Ruby there is only a single integer type[2] that has an arbitrary maximum capacity).

Possibly more factors exist, but these are not identified in this thesis.

In multi-language interoperability, absolute seamlessness (i.e. 100% seamless interoperability) is, in general, not achievable, especially not when languages are being used with very distinctive feature sets.

[2] This type may have several implementations but this is kept transparent to the user.

2.2 Scenario 1: Productivity

Here, we outline our first scenario, but before we do this, we need to introduce some additional terminology that is only applicable to this scenario. We shall, therefore, do this now.

2.2.1 Terminology

Within the context of software engineering, a *measure* provides a quantitative indication of the extent, amount, dimensions, capacity or size of some attribute of a product or process Pressman [57] (E.g. the number of errors).

There are *direct* and *indirect* measures. Examples of *direct* measures are lines of code (LOC), execution speed and memory size. Likewise, *indirect* measures include functionality, quality, complexity, efficiency, reliability, maintainability (and many other -abilities). A *direct* measure, is also said to have a *natural unit*.

A *measurement* is the act of taking a measure. Furthermore, a *metric* relates one or more data points resulting from a measurement in some way (E.g. the number of errors per software module, or per line of code).

Finally, an *indicator* is a metric or combination of metrics that provices insight into the software process, project or product itself[58] (for example 0.01 errors per line of code may be *good* whereas 0.5 errors per line of code may indicate *bad*).

2.2.2 Software Productivity

In economic terms, *productivity* is the ratio between the amount of goods or services produced and the labor or expense that goes into producing them (Jones [49]). Intuitively, when extended to software engineering, this would lead to the assumption, that *software productivity* is the ratio between the amount of software produced to the labor or expense of producing it. In theory, this sounds pretty solid, in practise it appears to be fuel to much debate.

The problem is, that productivity in general represents a *metric*. For example, 40 hours of labour produces 120 items, which means, three items are produced during one hour of labour. This implies that both input and output can be quantified. But how does one quantify the output of software production? Perhaps its better to first ask ourselves, what is software ?

While there are (arguably) better definitions for the term *software*[3], within the context of *Software Productivity*, software is often seen as one or more computer programs, each comprising of lines of source code[4].

Again, intuitively, it is tempting to think that lines of source code would be the measure to use when

[3] One interesting definition of Software is that of Bergstra [24]: "[Software is] whatever can be transported via a glasfiber of at least one meter length except physical forces and pure optical energy."

[4] In the light of this, perhaps it's better to use the term Programming Productivity rather than Software Productivity.

quantifying software production. However, measures of programming productivity and quality in terms of lines of code, and cost of detecting and removing code defects are inherently paradoxal (Jones [48]). The paradox can be found in that lines of code tend to emphasize longer, rather than efficient or higher quality programs. Also, higher-level programming languages are penalized, when compared to lower-level programming languages, such as assembly languages, when lines of code are used per unit of effort. This is because higher-level languages allow for shorter programs to be written.

Furthermore, Jones [49] states, lines of code, of themselves, are not the primary deliverables of a software project, and customers often do not know how many lines of code there are in the software they are buying. A consumer buys or uses software based on what it *can do* and not on *how it was programmed* or *developed*. This means, that in software development the (economic) value of the produced goods are not measured in the same units as the units of development. So a better input/output unit needs to be defined. One that reflects the *utility value* of software based on the *function* that [the] software is intended to perform (Jones [49]).

If we use the *functional value* (indirect measure) of the software in preference over the *lines of (source) code* (direct measure), we can give a better definition of the term Software Productivity:

Software Productivity is the ratio between the functional value of software produced to the labor and expense of producing it (Mills [53]).

2.2.3 Functional Size Measuring

In 1979, Allan Albrecht published a paper [19] which laid the foundation of what was to become known as the subject of *Functional Size Measuring* [33]. In this paper, Albrecht describes a method for measuring the size of software in terms of functionality, which he called Function Point Analysis (FPA).

Over the past few decades this has evolved into the 'IFPUG' method [43], which is being managed by the International Function Point Users Group. Besides this method for functional size measuring, several other variations have come into existence. Each of these extends the field of application for function point analysis, which was originally limited to medium to large scale, administrative, information systems, and business software.

Counting in FSM

All functional size measuring methods have in common that they are *indirect* measures. The functional size is typically the result of a weighted sum of a set of measurement parameters[5] that *can* be counted directly [57].

This sum can then be post-processed by applying a series of complexity adjustment values. This is done by assigning a set of questions[6] a complexity value on a scale from 0 to 5.

[5] Examples are: The *number of user inputs*, *user outputs*, *user inquiries*, *files*, *external interfaces*, etc.
[6] These are questions such as: "Is performance critical?"

Computing function points (FP) is then done using the following relation:

$$\texttt{FP} = \texttt{count-total} \times [0.65 + 0.01 \times \sigma F_i]$$

where:
`count-total` is the result of the weighted sum discussed above.
σ is the size of the weight.
F_i is the complexity adjustment question.

With the advent of newer, and more refined, methods for functional size measuring, variations apply.

Once function points have been measured, they can be used to determine software productivity, quality, and so on. Examples are: Errors per FP, defects per FP, cost (i.e. $ or €) per FP, and so on.

2.2.4 Influences on Software Productivity

Software productivity depends on many factors. In [21], Victor Basili and Marvin Zelkowitz define five important factors that influence software productivity:

1. *People factors* - The size and expertise of the development team.
2. *Problem factors* - The complexity of the problem at hand, the number of requirements.
3. *Process factors* - Analysis and design techniques that are used, programming languages and CASE tools available, and review techniques used.
4. *Product factors* - Reliability and performance of the computer-based system.
5. *Resource factors* - Hardware and software resources.

In fact, Barry Boehm, an established scientist in the field of software estimation and productivity, in an interview states that: *"The people factor revolves around capability, experience, collaboration, and retention. You'll usually have a productivity factor difference of over 10 if you're doing the same job with qualified as opposed to unqualified people."*[51].

Although, as we can see, there are many factors that influence software productivity, in this thesis we focus in on a particular factor, the *process* factor, and even more specifically: The choice of *programming languages* used.

2.2.5 Software Productivity and Programming Languages

So far we have discussed two ways of sizing software. *Lines of code* and *functional size measurement* (expressed in (full) function points). When both ways of sizing software are related, some interesting phenomenon can be seen. The relationship between number of lines of code, and number of function

points, depends on the quality of the design *and* that of the programming language used (Pressman [57]). In [18], Albrecht and Gaffney publish quantified results for this, where they relate the *functional value of software* to *source lines of codes* and *development effort prediction*.

Programming language	LOC/FP (average estimate)
assembly language	320
C	128
Cobol	105
Fortran	105
Pascal	90
Ada	70
php	67
Java	31
object-oriented languages	30
fourth generation languages (4GLs)	20
code generators	15
spreadsheets (excel programming)	6
graphical languages (icons) (draw-a-program languages)	4

Figure 2.1: Relationship between programming language, source lines of code and functional value. Source: [57], [63]. The results for *Java* and *php* are preliminary and may suffer from sparse data [63].

Higher-level versus Lower-level Programming Languages

Of course, intuitively, this is nothing new. Our intuition already tells us that it is probably more productive to write a certain program in C than it is to write that same program in an assembly language.

Likewise, according to table 2.1, similar things can be said about *Ada versus* C or *Java versus Pascal*.

In such pair-wise programming language comparisons there is always a *higher-level* programming language compared to a *lower-level* programming language. A higher-level programming language hides more details[7] of the underlying computer system from the programmer than a lower-level programming language.

The relative scale we use here is intentional; in a Java versus C comparison, Java is clearly the higher-level programming language and C the lower-level programming language. But in a comparison between C and an assembly language, it is evident that C can be seen as the higher-level of the two.

It is also important to note that, while the programming language generations play a role in the higher-level versus lower-level distinction, it is also possible to distinguish higher-level from lower-level languages within the same language generation.

[7] These are details such as: architecture, instruction set, memory layout, memory management, etc.

Ruby

```ruby
def concat( s1, s2 )
   return( s1 + s2 );
end
```

C

```c
char* concat1( char* s1, char* s2 )
{
   int destlen = strlen(s1)+strlen(s2);
   char* res = (char*) malloc(destlen+1);

   strcpy( res, s1 );
   strcpy( res+strlen(s1), s2 );
   res[destlen] = '\0';

   return( res );
}
```

```ruby
def concat( s1, s2 )
   return( s1 + s2 );
end
```

```c
void concat2( char* s1, char* s2, char* dest )
{
   int destlen = strlen(s1)+strlen(s2);

   strcpy( dest, s1 );
   strcpy( dest+strlen(s1), s2 );
   dest[destlen] = '\0';
}
```

Figure 2.2: String concatenation in two 3GL languages.

Example Comparison

To illustrate the difference between such a higher-level language, and a lower-level language, we shall give a small example. In this example, we will show how two different *third* generation languages(3GL), C and *Ruby*, are typically used to implement a common problem: String concatenation (see figure 2.2).

We give actually two solutions for the C case, which we will explain in a bit. What at least becomes clear from the example is that something which is regarded trivial from the perspective of *functional value* becomes pretty non-trivial in a lower-level language such as C, while it remains reasonably trivial in the case of the higher-level language, Ruby.

What is happening in C is that, since memory is managed explicitly, one must decide up front who is going to take care of reserving memory for the concatenated version of the string. In the top-most case, memory is reserved by the function itself, but this delegates the responsibility for releasing that memory, at some later stage of execution, to the party that calls the function. (This, of course, causes the additional burden of communicating this behaviour through API documentation.)

The bottom version of the C implementation requires the calling party to reserve memory for the concatenation result up front. This requires the *calling* party to *know* of details which it should not have been bothered with in the first place.

Needless to say, it is details such as these, that makes it less productive to program in a lower-level programming language, such as C, than it is to program in a higher-level language, such as Ruby.

Indeed, this is a very rude illustration, but differences become more clear when both programming

languages are used, side by side, for an extended period of time.

2.2.6 80/20 Conjecture

In the early days of C, developers welcomed the productivity and portability gain it provided over lower-level languages, such as assembly languages. But, since it was a higher-level language, C provided less execution performance than did an assembly implementation. The result was that, in those early days, programmers often had to resort to implementing small portions of their application in assembly, trading in some of the productivity and portability gain for a little more computational performance.

Moore's Law

In 1965, Gordon E. Moore did an empirical observation that the number of transistors on an integrated circuit, for minimal component cost, doubles every two years[54]. In reality, the number has doubled every 18 months, or so, for the past four decades.

As time progressed, there was less and less need for doing things in assembly, in part because compilers improved, and in part because the processor power increased.

But time progressed even further, giving computers much more execution and memory performance than a few decades ago. Eventhough the theoretical limits come into view, Moore's law seems to hold for at least several chip generations to come.

The nice thing about Moore's law is that it works both ways. Either for the same money you will get roughly double the processor power, or you will get the same processor power for roughly half the price.

Increase in Software Productivity?

But what about the increase in software productivity?

Inappropriate measures, such as the perception that software productivity increases at rates of only 4-6%, are usually derived from counting lines of code[60]. In fact, Boehm and Basili [27] note that it has been stuck at about 10 lines of code per person per day for years. This is not so surprising, since it is a measure for what a human can produce, however, as Mary Shaw[60] puts it very nicely:

> *"It hides the very real progress in software abstractions, which contribute by providing more powerful design elements for those lines of code to present."*

So, on one hand we have, hardware performance, which, following Moore's law roughly doubles every two years and on the other we have software productivity, the increase of which is appearing to lag behind.

We think, that one way to look at Mary Shaw's words is to say that, at present, a line of source code means more, because it is written in a newer programming language, with a higher level of abstraction, but, on the down side, also with less execution performance. However, in order for software productivity to catch up a little, we think, there is room to trade in some processor power for a gain in productivity.

In some ways, this is no different than what happened in the early days of C, back then one could say that:

Conjecture:

> Most source code of a software product (~80%) is actually non-computational intensive (support) code that runs in only a small portion of the execution time of a program (~20%), while a relatively small percentage of the source code (~20%), because of computational intensive algorithms, consumes a large part of the execution time of a program (~80%).

Over the past few decades, as time progressed, and programming languages got better, more consise, more productive (more higher-level), why can't we say this right now?

We think it is about time for history to repeat itself. This time taking C as the lower-level language and with a modern higher-level language; the game stays the same, only the players change.

2.2.7 The Scenario

As we can see, there many diverse factors that influence software productivity. Many of these are well beyond the scope of this chapter (and also of this thesis).

However, if we keep other factors that influence software productivity (as displayed in section 2.2.4) constant, then a development team, equally skilled in two different programming languages, one language being considerably more productive than the other[8], should be able to be more productive if multi-language software development is used. The productivity gain would be achieved by shifting implementation of the non-computational (support) code (~80%) to the more concise and productive, higher-level, programming language, while leaving only the computational intensive algorithms to be implemented in the faster executing, but more elaborate, lower-level programming language.

2.3 Scenario 2: Multi-Disciplinary Software Development

As software products get larger and more feature rich, so do the development teams that have to create the software. Of course, modularizing the design, and dividing it into components, helps in making larger software products with ever-growing development teams.

But this way of dividing the workload across a *homogenous* group of development teams with similar skills and experience is not what we wish to address here.

[8] Consider, for example, table 2.1.

By *multi-disciplinary software development* we depict a way of developing software with a *heterogenous* group of development teams (or invidual contributors). Each team has different skills and experience, and each team contributes its own specific expertise to the creation of the final product.

2.3.1 Higher-level versus Lower-level: Multi-Disciplinary Point of View

One way of looking at these heterogenous development teams is by looking at differences in expertise or skills, and in particular: Differences in skills in higher-level and lower-level programming languages.

In the previous section we outlined a scenario in which the *same* development team, skilled in both the lower-level *and* the higher-level language, would develop in both languages. In the development of larger software products, it is more common to divide the workload between different development teams. One team specializes in the features that need to be implemented in the lower-level programming language, and another team specializes in the features that need to be implemented in the higher-level programming language.

This has several benefits, the most important of which is that development in these lower-level languages is expensive, so it pays off to use good, quality, developers, with a lot of experience[9]. Unfortunately, good, quality, developers in these lower-level languages are hard to find. This makes them –ironically enough– expensive per hour. So either way, you want to keep these development teams small.

Fortunately, we already provided some argumentation for writing the bulk of the source code, which is assumed to be non-computational intensive, in a higher-level language. Because these programming languages are often easier to master, personnel is easier to find, so these development teams can be bigger or there can be more of them.

Video Game Industry

One example where multi-disciplinary software development takes place, today, is in the video game industry. As gaming consoles have become more powerful with each generation, the games have become bigger and bigger too.

The times that a computer game could be created by a small and single development team are long gone. Modern games are created using tens, if not hundreds of individuals, each with a different background, ranging from core programmer, via game logics and A.I., to art, (3D) modelling, and design (Phelps and Parks [55]).

All these individuals (divided into teams) contribute to the final product. Even within the programming teams several subcategories can be distinguished.

Because of the increase in processing power, not all parts of a commercial game have to be written in C or C++ (or even optimized assembly) anymore. More and more parts of a video game can now be written in higher-level languages, by people who have different concerns than those involved in making

[9] Recall our quote from Boehm about influences on software productivity from section 2.2.4.

the game engine, the physics simulation, or the surround sound run smoothly, as it is implemented in C or C++ (or assembly still).

These developers (and development teams) focus on things, such as, game logics, A.I., game plots, and things like that. They develop in high-level scripting languages, because it helps them to focus on their task at hand, or it allows them to exploit the dynamic possibilities of such languages to improve game ai([39, 55, 61]).

However, these different teams need to cooperate, because, eventhough their concerns may be distinct, the things that they work on, may possibly be very related. (After all, they are working on the same product, a video game.)

As a small example, consider for instance, the following: A 3D and physics engine developer is working on the game engine. He "sees" a 3D character (an object) that is walking in a virtual environment. He also "sees", that another object is in its vicinity. While he observes, he is, perhaps, focussing on how smooth, or rather, how choppy, the simulation is.

What this engine developer, however, does *not* know, is that this character is, in fact, *"Thorgrim the Grudgebearer"*, who automatically must engage his "Orc enemy, *Urgat*," whenever he is near. This, is the concern of the game logic and A.I. developer, who works in the high-level scripting language.

So, while there is the game engine developer, who creates all these seemingly interesting possibilities: Smooth skeletal animation, realistic physics simulation, objects that can attach themselves onto other objects, etc.

It is the higher-level game logic developer that gives meaning to these possibilities[10]: E.g. "Thorgrim, sitting on his throne, as it is being carried by four men".

Therefore, there has to be a clear and powerful multi-language interface that connects the lower-level 3D and physics engine to the high-level game logic and A.I. scripting environment.

2.3.2 Power to the People

Another way of looking at multi-disciplinary software development –and one that excites us most– is by looking at a certain phenomenon that is becoming more and more prevalent these days:

As a result of an over-abundance of more and more powerful computerized equipment, the past few decades, our society, and the public, has undergone a great technical development. Because of this, and the availability of enabling appliances and computer (web)applications, the role of the consumer is gradually changing into that of a producer as well.

The result, of these developments, i.e. the prosumer[11], and related phenomena, such as, mass customization, consumer integration and open innovation[65, 56], have long been predicted almost 35 years ago.

[10] For the sake of simplicity, we ignore other skills, such as, 3D modelling, art and design here.
[11] In our case a portmanteau of *producer* and *consumer*[15].

The wikipedia article on *prosumer*[15] gives such a clear and concise description of the origins, that we think it is best to quote the following two paragraphs:

> In 1972, Marshall McLuhan and Barrington Nevitt suggested in their book Take Today, (p. 4) that with electric technology, the consumer would become a producer. In the 1980 book, The Third Wave, futurologist Alvin Toffler coined the term "prosumer" when he predicted that the role of producers and consumers would begin to blur and merge (even though he described it in his book Future Shock from 1970). Toffler envisioned a highly saturated marketplace as mass production of standardized products began to satisfy basic consumer demands. To continue growing profit, businesses would initiate a process of mass customization, that is the mass production of highly customized products.
>
> However, to reach a high degree of customization, consumers would have to take part in the production process especially in specifying design requirements. In a sense, this is merely an extension or broadening of the kind of relationship that many affluent clients have had with professionals like architects for many decades.

The results of prosumption, today, are omnipresent on places like the internet, ranging from mere community enabling message boards (where for example, the inexperience new comer is helped by the experienced fellow user, until the new comer, himself, becomes an experienced user, who can then help out yet other new inexperienced users), to blogging sites, and what have you not.

With the advent of community-driven development, and the wider growing acceptance of open-source and free software, the possibilities for consumers to actually contribute to the development of the products that they use are also coming into reach. This leads to new ways of developing products, by means of open innovation, where everybody is allowed to participate and contribute. And, at present, even some of the, traditionally, commercial parties are now opening their eyes, as they are joining in on the happening. Consider, for example, the transition of the Java platform license to the GPL.

Mozilla Firefox

In the particular example of the transition of Java to GPL, we give an example of a process tool, a product development tool, but this extends to end user products as well. Consider for example the, in the open-source community, very popular web brower, *Mozilla Firefox*[35].

This software has been possible by user contributions from all over the world. Of course, the core engine of Firefox is mainly written in C/C++, which leads to contributors, and a development team, with a very specific type of skills.

However, –and this is where things get particularly interesting from a multi-language software development point of view– a large portion of the functionality (although not core functionality) of the web browser, is developed in a higher-level, and easy to master, scripting language.

This has allowed developers, that are not skilled enough in the lower-level language, to contribute in the final product. It has resulted in a nice blend of contributions by developers with a different view on the

final product. In this way, the brower's graphical user interface can be developed by people who have experience in usability and user interfaces.

Also very interesting are the nearly 2,000 add-ons and extensions[4] that have become available by the aid of contributing developers who are not even part of the core browser's development team. Most of these contributions are from spare time amateurs to semi-professionals who saw an opportunity to add a feature, a functionality, that allowed them to make the browser better fit their needs. Once the extension is written, it can be shared with the rest of the community, so that others may benefit from it, or even adapt the extension to their own liking, as they too, gradually evolve from being just a consumer and enter the realm of prosumption.

It is functionalities such as these, that is, functionalities that the original (core) developers had not anticipated because they, for instance, did not foresee the application of their software for a certain particular purpose, that makes this sort of software development very interesting.

And, of course, –don't let us forget that this is a thesis about multi-language software development– where there is software development in different languages, there is a need for a multi-language interface that connects these languages.

2.4 Scenario 3: Best of Both Worlds

So far, we have given two theoretical arguments for multi-language development: Productivity and multi-disciplinary software development. For those who are not convinced yet, we conclude by outlining a scenario that simply focuses on the practical benefit of multi-language development: Combining the best language features of different programming languages to get *the best of both worlds*.

2.4.1 Language Features

A programming language, is characterized by a set of language features. These features influence how the programming language is used, which contexts it is best used in, and for what purpose. Using a programming language that is not well suited for a particular purpose, may prove to be counter effective.

As outlined in the introduction we only focus on general purpose imperative programming languages, but even within this specific group of programming languages there is a collection of language features differentiating one programming language from the other.

To illustrate this, let us give a few examples of such features:

Compiled versus Interpreted - Software development in one programming language (e.g. C, C++, or Java), or execution environment (native hardware, Java Virtual Machine), may require the program to be *compiled* before it can be executed. In other programming environments, or execution environments, such as Perl, Ruby, Python, or other scripting languages, the programs can be used directly because the source code of such programs is *interpreted* directly.

This distinction, makes such language fit for very different purposes. Because the scripting languages don't need to be compiled, no compiler is needed, and because of the omitted compilation step, this makes for a very swift development strategy. On the other hand, the resulting end product, because of the interpretation, will have a much lower execution speed performance.

Interactive Code Experimenting and Testing - Some programming environments, because of the interpreted nature of the programming language or execution environment, provide an interactive prompt([32, 37, 47] and others). This prompt can be used to interactively test certain portions of the source code for correctness directly. In our experience, with these interactive development tools, this leads to an impressive productivity boost, because the feedback loop is even shorter than that of traditional interpreted programming languages (see above).

Memory Management - Programming languages can also have differences in memory management strategies. Some languages require explicit manual memory management, while others can be equipped with garbage collection algorithms to perform automatic memory management.

Execution Errors - Different programming environments also differ in the way they handle execution errors. Cardelli [30] argues that it is useful to distinguish between two kinds of execution errors: execution errors that cause computation to halt immediately and execution errors that go unnoticed –for a while– and later cause arbitrary behaviour. They are called *trapped* and *untrapped* errors respectively. This feature influences speed of debugging, which in turn is likely to influence speed of development.

Operating System Interaction - Some programming language allow for a very tight integration with the underlying operating system, while others are running on a virtual machine, making the distance to the underlying operating system intentionally large.

Semantic Gap - Some programming languages have a larger *semantic gap* between them and the underlying physical representation (e.g. the computer hardware) than others. For instance there is a small semantic gap between C and the computer hardware[45], while there is a larger semantic gap between Java and the underlying computer hardware (which is not in the least caused by its virtual machine).

Portability - The size of the semantic gap also influences the portability of the resulting source code. In this regard, assembly languages are less portable than C source code which, in turn, can be less portable than Java source code, and so on.

Native versus VM - Within the group of *compiled* languages, there is yet another distinction: That of *natively compiled* versus *virtual machine (VM) compiled*. Programs that are compiled into *native* code, are transformed into machinecode instructions, to be executed by the operating system and physical hardware. Whereas, programs that are compiled into *virtual machine code* (sometimes called bytecode [50]), are to be executed by a virtual machine, and by this, circumventing the host operating system and hardware. This increases the *semantic gap* and *portability* of the programs that are written in languages that are compiled to virtual machine code, but it usually also means that interaction with the underlying operating system is decreased.

2.4.2 The Scenario

So far, we have seen a small selection of possible language features, and that these language features may influence the fitness for a certain purpose.

By doing multi-language development using two or more programming languages with distinct feature sets, we can achieve an increase of the effectivity of any single such programming language. The language interface then allows access to certain *language features* that are not available in the current programming language.

For example, if software development mainly takes place in *Python*, and access to explicit implementation details, such as, *pointers*, *explicit memory layout*, or *memory management* is needed, then a fall back to a lower-level language, that provides such language features (e.g. C), may be required.

This too validates the use of multi-language development.

Chapter 3

Related Work

The multi-language interoperability problem is not a new problem. As a result of this, there are many, many existing solutions. It would make this thesis at least twice as large if we were to go into detail of each and every one of these. For this reason, we have chosen to simply categorize the existing solutions, to make it more clear where our solution, which is presented in later chapters, fits in.

Where applicable, we will go into more detail. For instance, if the discussed solution has more similarities to our solution, it is useful to go into a little more detail.

3.1 Distributed Versus Non-Distributed

While there are other classifications possible[20], we think it is useful to categorize approaches to the multi-language interoperability problem into two distinct groups:

1. *Distributed* approaches
2. *Non-distributed* approaches

What we mean by this will become clear in the next few sections.

3.1.1 Distributed Approaches

In our categorization, a distributed approach to solving the multi-language interoperability problem is characterized by the way two or more software components, written in different programming languages, communicate. If this communication occurs by means of *Input/Output (I/O) streams*, then we say it is a distributed approach (regardless whether the actual components reside on the same machine or not).

Furthermore, we identify the more traditional *client-server* architectures, where a custom protocol is defined between a client and a server. This allows both client and server to be written in different

programming languages. However, multi-language interoperability may not be of primary concern to these architectures.

Next to that, there are the more formal approaches that actually aim to solve the multi-language interoperability problem by providing an architecture with a single, standardized, distributed, protocol (or sometimes even a collection of protocols).

For instance, there are the *broker-based architecture*s. Under this approach we classify such solutions as[1]:

- Corba[46, 67];
- ILU[36];
- PolySpin[20];

Furthermore, we have the more recently added *Service Oriented Architecture*s (sometimes more colloquially referred to as *web services*, but SOA possibly implies more than just that). As these approaches are relatively recent, there is no concensus yet that umabiguously defines these approaches, but we think it is safe to say that it concerns an architecture of loosely coupled services. Examples of these approaches are[2]:

- Enterprise Service Bus (ESB)[31];
- ToolBus[23, 40];
- Web services (e.g. approaches using SOAP[34], REST[10], etc.);

And then, there is the *unix pipe* approach, which, by our classification, can definately be seen as a distributed approach to the multi-language interoperability problem: Two shell programs (be it a *script* or a compiled program), written in different programming languages, *can* be made to work together using the *unix pipe* by means of *input/output streams*. The actual protocol, however, differs from program to program and pipe to pipe.

On an interesting note, the *ToolBus* (categorized above) can be used to interface to (shell) programs by means of the *unix pipe* facility.

3.1.2 Non-Distributed Approaches

A *Non-distributed approach*, in our categorization, implies an approach that, in contrast to a distributed approach, *does not* use input/output streams to establish a form of communication between two or more software components, written in different programming languages.

[1] These lists are not exhaustive.
[2] Idem.

Typically this means, that special features have to be implemented in order to allow for multi-language interoperability. These approaches are more similar to what Barrett et al. [20] refer to as *low-level* approaches, eventhough, they are not strictly the same.

On a special note: The choice of non-distributed approaches to multi-language interoperability does not exclude using distributed approaches, both approaches can easily coexist in large distributed software projects. The actual distinction we make is in the way the communication between the components is established.

There are several ways in which two software component,s written in different languages, can be made interoperable, but it often requires that one of the two software components is somewhat bilingual in that it incorporates the above mentioned *special features* to allow for the communication. Typically, this means that there must be some mechanism available to share data objects and/or call functions *across the border*.

The three most prominent approaches we see are:

1. *Embedding* a scripting engine inside a certain component;
2. the ability to *introduce* new functions or methods cross-border; and
3. *compiling* different languages into a common new language.

3.1.2.1 Embedding a Scripting Engine

In this approach, a scripting engine is embedded as part of a software component. The scripts are then interpreted by this component and usually one or more of the following interactions occur:

- The results of the executed scripts are returned (and sometimes converted into a result that is usable in the host programming language).
- The host makes special functions or data objects available that can be accessed, altered or invoked during script execution.

An example of the former would be *Java Terms*[41].

For the latter option, many examples exist. From embedding special purpose or custom scripting engines in projects, such as, *ID Soft's* popular game series, *Quake*, where a custom scripting engine with C-like syntax, QuakeC, was used to script game AI in; to embedding more general, existing scripting engines, such as (but certainly not limited to): *LUA*, *Python*, *Ruby*, *Tcl/Tk*, etc.

3.1.2.2 Introducing New Functions

The second approach is characterized by the possibility to introduce new functions from one side of the border to the other. Typically, one programming language, or execution environment, allows new

function calls to be introduced, that can invoke methods or functions across the border, similar to *remote procedure calls*. This facilitates the communication.

Examples of such approaches are:

- Java Native Interface - Here *"native java methods"* can be introduced that can be invoked directly from within the Java Runtime Environment (JRE). The native java methods are implemented as specially prepared C functions.
- Ruby Extension API - Here specially prepared functions, implemented in C, can be registered as ordinary Ruby methods.

For other programming languages (e.g. Tcl/Tk, Python, etc.) similar solutions exist.

3.1.2.3 Compiling Into a Common Language

The third solution is inspired by the thought that, eventually, programs can be rewritten as a sequence of machine instructions, regardless of the programming language it is written in. It seeks to define a single programming language that incorporates all important aspects of all supported (other) programming languages. These are then projected onto –compiled into, if you will– this one programming language:

Once two software components written in different programming languages have been compiled into the single target programming language, they are be able to communicate (e.g. access common data objects or invoke eachother's functions).

The most prominent in this field, and the one of the most recent additions, is filed under ECMA-standard 335, also known as the *Common Language Infrastructure*. With its most widely used implementation: Microsoft's *.Net*.

3.2 A Closer Look

As a part of our language selection process (which we shall discuss later in chapter 5), we have looked into some of the existing solutions in more detail, therefore it is useful to introduce these here in a little more detail too.

3.2.1 Java

Java is a programming language originally developed by Sun Microsystems and released in 1995. Java source code is typically compiled into bytecode that is to be executed by a virtual machine. This Java Virtual Machine (JVM) is part of the Java Runtime Environment (JRE) which also comes with a wealth of standard libraries.

Because of the virtual machine design, Java supposedly allows for a write-once run-anywhere software development strategy, where programmers need not worry about the portability of their programs, since the whole virtual machine is ported to new target platforms, instead of the individual applications.

Java Native Interface

To interface Java, and its virtual machine, with the underlying native operating system, it comes with the *Java Native Interface*.

In order to establish the interface to the native operating system, one writes specially prepared functions in C, which can then be bound to the JVM, as if they were ordinary java methods, by declaring them native. See figure 3.1 to illustrate this.

```c
#include <jni.h>
extern "C" JNIEXPORT jobject JNICALL
  Java_NativeExample_doSomething( JNIEnv* env, jobject obj, jobject arg )
{
  jobject res = NULL;
  jclass  cls = env->GetObjectClass( obj );
  // 1.  do type mapping/conversion Java -> Native

  // 2.  do native things

  // 3.  do type mapping/conversion Native -> Java  and  return result
  return( res );
}
```

```c
public class NativeExample
{
  native MyResultObject doSomething( MyJavaClass obj, Hashtable arg );
}
```

Figure 3.1: A specially prepared C function (top) is bound to Java as a native method (bottom).

3.2.2 Common Language Infrastructure

The Common Language Infrastructure, is an open specification that defines a system that allows software components that are written in different, but supported, programming languages to seamlessly interoperate. According to [9] the CLI specification consists of four parts:

1. *The Common Type System (CTS)* - "A set of types and operations that are shared by all CTS compliant programming languages."

2. *Metadata*

3. *The Common Language Specification (CLS)* - "A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages."

4. *Virtual Execution System (VES)* - "The VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime."

As stated previously, the CLI is an open standard and it is filed under ECMA-standard 335.

There are currently three alternative implementations of the ECMA standards, of which Microsoft's .Net[3] is the most widely known implementation. The two other implementations are:

1. *The Mono Project[6]* - An open source implementation, sponsored by Novell.

2. *DotGNU[2]* - The part of the GNU project that aims to provide a *free* implementation of .Net.

Because .Net is by far the most widely known implementation of the above ECMA standards, we will use it as a proprietary eponym for any implementation of the CLI in the remainder of this thesis.

Common Intermediate Language

As we described earlier in this chapter, interoperability approaches similar to the CLI are categorized by us as approached that compile into a common language. In the CLI this language is called the *Common Intermediate Language* or CIL in short.

Common Language Runtime

The *Common Language Runtime* (CLR) is often confused with the CLI, however, the CLI is the specification and the CLR is an implementation of the execution environment that is specified by the CLI.

Implications of the CLI

There are a few implications resulting from the design of the CLI. First of all, it implies that for a language to be supported by the CLI it must be able to project itself onto the CIL.

Secondly, tricks that are supported by *dynamic* languages, such as *Ruby*, *Python* and so on are in general not supported by the CLR.

So while the CLI is a technical achievement in its own right, the range of languages that it supports is not unlimited.

Encapsulation Flaw

In theory, .Net promises absolute seamlessness between all software components written in languages that are supported by the CLR (which must be incorporated into .Net, of course). However, as Beugnard [25] shows, in practise, the seamlessness appears to be less than absolute because of slight differences in encapsulation between various object-oriented languages.

3.3 Are Dynamic Languages the Future?

The fact that we are aiming for a language interface between a higher-level language and a lower-level language, makes this a very interesting question. (It is likely that we are going to select a dynamic language as the higher-level language.)

Well, are they? Of course, nobody can see into the future, however, recent developments by both Sun and Microsoft (the driving forces behind Java and .Net) are showing increased interest in dynamic languages; both software giants are extending their frameworks by adding support for more *dynamic languages*.

Dynamic Support for Java

Sun is currently trying to add dynamic support to their Java Virtual Machine. This implies alterations (or additions) to the bytecode specification. For more information on this subject see [7].

DLR

Also Microsoft is joining in on the *Dynamic* future, by acknowledging the limitations of the CLR. Hard work is being done on the DLR: *The Dynamic Language Runtime*[8].

Chapter 4

Goals and Requirements

In this chapter we identify the goals and requirements for our multi-language software development solution.

We have three sources from which we derive these. First we limit the field of application by setting a few requirements that act as a starting point. Then we derive additional goals and requirements from the three scenarios that we presented in chapter 2. Finally, we supplement the established list with goals and requirements derived from existing literature.

The difference between a goal and a requirement is that a requirement always defines a bounded and measurable outcome, while a goal represents something of which you do not know up front to what extent you will reach it (i.e. when it is impossible to define such bounds). In this chapter, goals and requirements are specified in mixed order.

4.1 Field of Application

First we limit the field of research, by setting a few requirements, that act as a starting point for our solution.

The first requirement that we set is to address multilingual interoperability, rather then polylingual interoperability. We want to interface components written in one of two programming languages.

Requirement 1: *Present a solution that addresses the multilingual rather than polylingual interoperability issue.*

Three moments in time can be distinguished in which interoperability between two software components *A* and *B* is required and to be added[20]:

1. before both *A* and *B* have been written.
2. after *A* has been written, but before *B*.
3. after both *A* and *B* have been written.

We only want to target the first two situations, the third option, which is sometimes called "megaprogamming"[20, 26], is really a complex problem in its own right, which we conveniently ignore in this thesis.

To make it easier to address these two situations later, we shall describe the first situation as having two *original* software components. The second situation shall be described as a situation with an *existing* software component.

Requirement 2: *Present a solution that targets both interoperability between two original software components as well as interoperability between an original and an existing software component.*

In the case of interfacing with an *existing* software component, this existing software component may have been written without taking multi-language interoperability into account. We must therefore be able to establish a fully effective interface from just one side (i.e. from the original component).

Requirement 3: *Present a solution that makes it possible to establish the interface from just one side.*

4.2 Productivity Revisited

In chapter 2 we argued in scenario 1 that an increase in software productivity may be expected –taking all factors in consideration, of course– when multi-language software development is used, by shifting certain portions of the software to either a higher-level programming environment, or a lower-level programming environment.

Validating or falsifying such an hypothesis directly, would require both a solution and several case studies to quantify the productivity between software development that uses the solution, and software development that does not use the solution. Apart from the fact that, because of the many factors that influence it, software productivity is very hard compare across software projects, doing such extensive case studies is well beyond the grasps of this Master's Thesis.

We therefore keep to the existing argumentation for doing multi-language development (the expected productivity gain) and assume this to be true.

Instead, we shall focus on the underlying assumption, that introducing multi-language development into a sofware project does not *by itself* impact the productivity so much, that the overall expected productivity gain will be lost.

This assumption forms one of the main starting point for our research. We are going to attempt to minimize the impact of adding multi-language development to such a software project. For this, we think, there are two important factors that influence software development productivity, when multi-language software development is added, these are:

1. *Interface creation and maintainability*
2. *Efficiency of the interface in use*

4.3 Interface Creation and Maintainability

4.3.1 Interface Creation

When introducing multi-language software development, before a multi-language interface can be used, it must first be established. This requires interfacing different languages with different features. For example, a sensible mapping must be established between the language features of different languages. The result of such efforts leads to the construction of a multi-language interface. However, these efforts distract the attention from the original software project's goal, which was not multi-language software development per se. (Pleaser remember: To the ignorant consumer, functional value of software is more important than how that software is constructed.)

Therefore, in order to minimize the impact of introducing multi-language development into the software development process, the construction of the multi-language interface must be kept simple and easy to create.

Goal 1: *Present a solution that keeps it simple to construct the multi-language interface.*

4.3.2 Maintainability

Once a multi-language interface is constructed, it must be maintained. If, for instance, the interface of an existing third party component changes, then these changes must be reflected back to the multi-language interface.

We see three possible risks regarding the maintainability of a multi-language interface:

1. *Synchronization errors -* If it takes up too much time to maintain the interface, there is a risk that the interface between the third party component gets out of sync with the first party software project, because original project goals (the functional value to the consumer) will be more important than keeping the interface in sync.

2. *Heterogenous mapping -* If multiple people are responsible for creating and maintaining the multi-language interface, there is a risk that the mapping between certain features (e.g. types) becomes heterogenous: one developer uses the multi-language interface to apply one possible mapping, another developer uses the multi-language interface to apply another mapping. Heterogenous mapping adversely affects both the usability and maintainability of the multi-language interface between two software components.

3. *Black boxing -* If the person, or persons, responsible for the original multi-language interface have left the development team[1], then there is a risk that important information concerning the interface (and its construction) is lost. In the most extreme case the multi-language interface becomes a black box, where no current team member knows any details concerning the multi-language interface at all, making it harder, or even impossible, to maintain the interface adequately.

[1] This is not uncommon after the release of product version 1.0.

For the productivity of multi-language software development it is therefore important to minimize these three risks.

Goal 2: *Present a solution that minimizes the risk of: a.* synchronization errors*, b.* heterogenous mapping *and c.* black boxing*.*

4.4 Efficiency in Use

A multi-language interface between two software components written in different languages must also be efficient during use. If using the interface is too cumbersome, then any productivity gain introduced by the more productive language may be lost.

We identify several factors of which we think may positively affect the efficiency-in-use of a multi-language interface, these are:

- *Increasing seamlessness*
- *Dynamic boundaries*
- *Separation of concerns regarding portability*

4.4.1 Increasing Seamlessness

If we attempt to interface a software component written in a lower-level programming language, which is considered to be less productive and more elaborate, to a software component written in a higher-level programming language, which is considered to be more productive and more concise, then it is important that the factors that cause the lower-level programming language to be less productive and more elaborate do not influence the productivity and conciseness of the higher-level programming language.

In section 2.1.7 we gave a definition for seamlessness. As we stated there, seamlessness of multi-language interoperability is influenced by various factors, for instance:

- *Encapsulation* - The ways in which different programming languages are able to bundle related data in the form of objects, structures, records, functions, procedures, etc.
- *Information hiding* - For instance hiding certain implementation details that are explicit in one language, but transparent in another (e.g. pointers are explicit in C++, but their use is transparent in Java).

We add the following two requirements to increase seamlessness:

Requirement 4: *Present a solution that provides at least the same level of encapsulation to the higher-level programming language as the lower-level programming language does.*

Goal 3: *Present a solution that hides unnecessary implementation details of the lower-level programming language from the higher-level programming language as much as possible.*

4.4.2 Dynamic Boundaries

In requirement 2 we require to target the multi-language interoperability problem for two situations:

1. *Original software components* - Both components have yet to be developed.
2. *Single-sidedly developed components* - One of the two components has already been developed (possibly without taking multi-language interoperability into account).

While there are possibly more ways to distinguish situation 1 from situation 2, one distinction seems very prevalent: In the case of situation 2, the interface (i.e. the *boundary*) that separates both modules is very clear and in fact complete (although third-party development activities[2] may still cause changes to the interface be made).

On the other hand, in the case of situation 1, it is not always clear from the start what has to go to which end of the fence. Leading to a more dynamic boundary that stabilizes as development progresses.

Our observation is confirmed by Phelps and Parks [55], who, in their research of multi-language development, note that, during the development of their Muppets game engine, it was not always clear which functionality was to go to which side of the boundary.

This points us to our next requirement:

Goal 4: *Provide a solution that makes it as easy as possible for adapting the boundary between the software components, making it easy to change what goes to which side of the fence.*

4.4.3 Portability

When a software developer develops a computer program and subsequently publishes it, for example, by making it available for download, he will, more than often, be confronted with the problem, that the application does *not* behave in the same way, or does not even work at all, on the computer of the end user.

These problems, are likely caused by *portability* issues.

"*Porting* is the process of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed."[14]. Subsequently, *portability* is a measure for *how well* this software is capable of being ported. There are no real metrics to express portability, or to relate the portability of one piece of software to another.

[2] For example when a new version of the component is released.

However, as any seasoned software developer with experience with both C and Java can tell you: Once the Java Virtual Machine is ported to, and deployed on, a target computer system, a java application is unmistakably easier to port and deploy than any C program is.

This is because when a C application has to be developed, it must take *portability issues* into account for each of the expected target platforms, *before*, or at least *during*, development. Then, before the application can be deployed, it has to be separately compiled for each target platform.

If, at some later stage, new target platforms are identified that had not been taken into account during development, then, enter *portability hell*, because, chances are, the application will not compile for the target platform.

In a sense, this is history repeating itself. Because in the early days of C, it was C that provided the great portability benefit over, for example, lower-level assembly languages, or even worse, micro code. These were often strongly tied to a specific processor or machine.

Now let us get back to efficiency in use...

One of the reasons a Java application is easier to develop is its support for the *separation of concerns* with respect to portability of an application: The responsibility of the portability of the application is taken out of the hands of the application developers; it is delegated to special *porting* developers. These *porters* focus on porting the execution environment, not the individual applications that make use of it.

Once an execution environment (e.g. the Java Virtual Machine and Runtime Environment) has been ported to a new target platform, all previously developed applications are able to be deployed on it. This reliefs the application developer from the burdain of taking target platforms into account, before, during, and after development of his application.

Requirement 5: *A solution must be presented that allows for the separation of concerns regarding portability related issues.*

4.5 Multi-Disciplinary Software Development Revisited

In scenario 2 of chapter 2 we presented a scenario, where multi-language software development could be used in an environment where developers of different skills and discliplines are to cooperate together.

The idea was, for example, to easily disclose the functionality of components, written in lower-level languages, to higher-level languages, that can be used by people, who are less, or not even at all, skilled in these lower-level languages.

This scenario, too, benefits from increasing seamlessness, because it will shield the less skilled developers, who work in the higher-level language, from the many complicating details of the lower-level language. However, since seamlessness is already on our list of goals, we do not have to add it here.

4.6 Effectivity

In scenario 3 of chapter 2 we argued that if we combine the effectivity of two programming languages with distinct features, that we may increase the effectivity of any single programming language.

For example, C gives access to pointer arithmetic and explicit memory access which is useful sometimes, while Java hides explicit memory management using a garbage collection scheme, which is useful at other times.

This scenario, of course, assumes that an effective mapping of all language features is possible.

Furthermore, earlier, in section 4.3.2, we argued that for the sake of maintainability, the multi-language interface should be easy to construct and maintain.

Then we added goals and requirements, of which we think, will have a positive influence on the above, such as, seamlessness, information, etc.

However, the simplifications that are likely to be introduced in order to meet such goals and requirements may lead to a solution that is not always able to interface any two software components A and B (again, one of which may already been written, possibly without taking multi-language interoperability into account).

Furthermore, from previous experience (which we will be sharing in the next chapter), we know, that similar solutions to what we have in mind, have proven to be ineffective when interfacing with some (of the more complex) existing components.

We do not want to fail because of similar faults, for us, it is therefore very important, that any solution be 100% effective, regardless of simplifications added for the sake of maintainability or seamlessness.

Requirement 6: *Provide a solution that always allows an effective interfacing between two software components A and B, regardless of any simplifications added for the sake of maintainability or seamlessness.*

4.7 Known Issues in Multi-language development

We have also searched existing literature for known issues concerning multi-language development. Phelps and Parks [55] summarize these in four common sources of problems:

1. *Lack of documentation*
2. *Performance*
3. *Memory management*
4. *Threading*

These four common sources of problems are not the focus points of our research. However, ultimately our solution has to address these problems sooner or later. So where applicable, we want to provide starting points for each of these subjects to prevent coming up with a solution that completely disqualifies itself with respect to any of these problems.

4.7.1 Lack of Documentation

During the construction of their research game engine, called Muppets, which uses an explicitly programmed interface using the *Java Native Interface* (JNI), they have found, that:

- Few resources exist about how to let multiple languages *behave* together.
- Existing documentation typically focuses on *simple* and *non-threaded* examples passing only *basic* data types (such as, integer and floating-point types).
- It is not clear how to handle type casting.
- How to debug across multiple languages.

To these points, we would like to add the following point of our own:

- Documentation about how to handle more complex types is lacking.

It is hard to form a usable requirement from Phelps and Parks observation, since it is very hard to determine when enough (adequate) documentation is provided. So we add the following goal:

Goal 5: *Strive for a solution that requires less (detailed) documentation.*

The documentation that afterwards still *is* required is then perhaps more easily covered (in part) by this thesis.

4.7.2 Performance

The word *performance* can mean a great many things and –more importantly– can have a *different meaning* to *different people*. To most people, when referring to the term *performance*, it means one or more of the following[62]:

1. *Computational performance* - How many computations can be done per amount of time? (E.g. how *fast* is a program?)
2. *Memory footprint* - How much memory (RAM) is occupied while a programming is being used?
3. *Startup time* - How much time is between starting the program and and the time it is ready to be used?

4. *Scalability* - How scalable is a program, solution or algorithm if more instances are used?
5. *Perceived performance* - How responsive is the program in the experience of the consumer or end user?

4.7.2.1 Phelps and Parks on Performance

Phelps and Parks, in their paper, only address performance as *computational performance*. They note that, in multi-language software development, much performance is lost by *moving things across the bridge*.

From the paper it is not clear whether this applies to both *objects*[3] and *function calls*, but we assume it does.

They do state that if it is fairly expensive to do a cross-lingual function call, relative to a local function call, then, from a performance perspective, it is best to minimize the number of calls crossing the bridge. Choices like these typically impact the API or interface of the software modules. This is, in correspondence, to our 80/20 conjecture, by which we argue that if performance is important, the computational algorithms are best isolated and done in the lower-level language.

However, since we also require via requirement 2, that our solution must be able to handle existing components (which may not have been written with multi-language development in mind), this is not under our control.

Following from scenario 1 in chapter 2 it is likely that our solution will be interfacing a more productive, higher-level, programming language, with less compational performance, to a less productive, lower-level, programming language, with more computational performance.

We assume that, for the sake of productivity, less prominent features, that do not require much computational performance, be programmed in the higher-level programming environment.

So a sensible computational performance requirement for our solution could be, to provide a computational performance that is comparable to the performance of the higher-level programming language. Similar things can be said on *startup time performance* and *perceived performance*.

Goal 6: *Provide a solution with a* computational, start up *and* perceived *performance that strives to be comparable to the performance of the higher-level programming language as much as possible.*

4.7.3 Memory Management

"*Memory management* is the act of managing computer memory. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed."[13]

Some parts of memory management[4] are handled at operating system level, completely transparent to

[3] *things* occupying memory space
[4] For example, relocation, protection, sharing, logical and physical organization.

the running program, or the programming language that this program was written in.

However, the two activities mentioned in the definition, *allocating* portions of memory, and *freeing* it for reuse when it is no longer needed, are tasks that have to be performed by any program or execution environment.

This can be done either *explicitly* (e.g. `malloc` in C) or *implicitly* (e.g. `Object.new` in Ruby). Tracking whether memory is no longer used can either be done *manually*, such as in plain C, or it can be done *automatically* using various garbage collection schemes. An automatic garbage collector can be part of a programming environment by default (e.g. Java, Ruby), or it can be an add-on (e.g. the boehms garbage collector for C and C++ [29, 28]).

The problem of memory management is large, because with the multi-language interoperability problem, it is likely that there are differences in the way memory management is taken care of by the different programming languages.

4.7.3.1 Object Sharing

Memory management problems occur mostly because of *object sharing*, i.e. the fact that a software module references an object from a different software module.

Two particular memory management problems that are likely to surface between two software modules *A* and *B*, and an object *x*, are:

1. *Dangling pointers*
2. *Memory leaks*

A *dangling pointer* is characterized by the following occurrences:

1. B internally references object x.
2. A references object x from B.
3. B removes the internal reference to object x.
4. B reclaims the memory to object x.
5. A still references object x which now no longer exists.

Of course, one could argue that B should not have reclaimed the object, but unthinkingly retaining all allocated memory is likely to cause memory leaks.

A *memory leak* is characterized by the following occurrences:

1. B internally references object x.

2. A references object x from B.
3. B removes the internal reference to object x.
4. B *does not* reclaim the memory to object x.
5. A removes references object x.
6. Neither A nor B references object x, but it is still in memory.

As we see, memory management can be a delicate issue in multi-language development. This is increased even further, if we consider requirement 2, where we require that we must be able to interface to existing software modules. Such modules may not have been developed with multi-language interoperability in mind.

Because memory management is such a complicated problem in itself, and because it is not the focus point of this thesis, we only seek starting points for addressing memory management using our solution in the future.

Requirement 7: *Provide starting points for addressing memory management using the found solution.*

4.7.4 Threading

"A *thread* in computer science is short for a thread of execution. Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does."[17]

Problems with threading are the fourth common group of multi-language interoperability problems identified by Phelps and Parks.

Likely problems and questions that they identify regarding threading are:

- How to synchronize native versus non-native threads?
- How to handle thread locking?
- References available in one thread may not be available in other threads.

One thing Phelps and Parks propose is to prefer non-native threads over native threads because of their positive effect on portability.

Unfortunately, threading and multi-language development becomes even more delicate, considering that we seek a solution to the multi-language interoperability problem in which one of the two software modules may already have been written (and possibly without even taking multi-language development into account). Since the multi-language threading problem, like the multi-language memory management

problem, is such a very complicated one in itself, we only seek starting points for our solution in regarding multi-language threading.

Requirement 8: *Provide starting points for addressing threading using the found solution.*

Chapter 5

Language Selection

From the previous chapter, we now know that we are looking for a solution to the multilingual[1] interoperability problem. In this chapter we answer the question: *"Which languages do we interface?"*

To this, we first pose some selection criteria which are, like the goals and requirements from the previous chapter, derived from the three scenarios of chapter 2. Then we discuss which alternatives have been considered, what existing work there exists, and why these are considered inadequate.

5.1 Selection Criteria

From the thesis's focus, which we established in the introduction, we know that all language that we consider in this thesis must be general purpose *imperative* programming languages. We do not consider other languages, such as, but not limited to, domain specific, declarative, functional, or logical programming languages.

Our selection criteria are then directly derived from the three scenarios of chapter 2.

5.1.1 Productivity versus Execution Speed

In our first scenario, *Productivity*. We have seen that there are programming languages that, under the right conditions, may be more productive than others. That is, we have seen that less source code is required to implement the same functional value in a variety of different programming languages (figure 2.1).

Then, we posed our 80/20 conjecture. In this we stated that most source code actually fulfills a supporting role that runs in only a fraction of the total execution time of a program, while only a small portion of the source code runs in most of the total execution time of a program.

[1] Recall its definition from section 2.1.2.

We then made the assumption that, from a productivity point of view, there may a productivity gain, if the non-computational intensive, supporting, source code (~ 80%) was to be programmed in a more productive programming language. A more productive programming language, however, has a tendency to be slower to execute, leading to the suggestion that the computational intensive portion (~ 20%) of the source code was to be programmed in a programming language that has higher execution speed. Such programming languages tend to be less productive.

So as a selection criterium we pose that the two languages that are to be interfaced meet the following requirements:

1. One of the programming languages must be considerably more *productive* than the other.
2. The less productive programming language must have a considerable *higher execution speed performance*.

5.1.2 Feature Set

Next, in scenario 3 (section 2.4) we explained that different programming languages can have different language features. Then we made the assumption that interfacing two languages with different feature sets should increase the *effective* of any single such programming language.

Obviously, the criterium that follows from this scenario is that, if we want to maximize the outcome of our assumption, we need to select two programming languages which have *very distinct feature sets*.

5.1.3 Easy Language

In scenario 2 (section 2.3) we argued that, in large software projects, there are different people with different skills working together on the same project. Such projects can be both in a *commercial* or a *non-commercial* setting.

We argued, that a large group of developers, or contributors, develop in a higher-level language, that is easier to use, and easier to master.

The criterium that follows from this scenario is then: One of the programming language has to be *an easy to master programming language*. A *scripting* language is a typical example of an easy to master programming language.

5.2 Ruby and C

After some considerations, we have selected to interface *Ruby*[37, 1] and C[44]. There are various reasons for selecting these languages which, as can be seen in table 1, all match the selection criteria.

In addition to these criteria, we have selected C, because it is a small and well-specified programming language. C is often used as the principle programming language for operating systems, both for desktop computing and embedded systems. Because of this, the main interface to communicate with such an operating system would be using C.

We think it is important to also select a programming language that has a large and active user base, because then the list of available extensions, add-ons and packages grow, which may aid future productivity because of software reuse.

In the light of this, we selected Ruby because it meets the selection criteria and, in recent years, has started to gain a lot of popularity. The language itself has been in existence since 1993 and was inspired by Ada, Eiffel, Smalltalk, with influences from other scripting languages. Through the years Ruby has developed into a stable scripting platform and the influences from these other programming languages have lead to a seemingly productive –although, as yet, no real world figures could be found– and easy to master programming language[2].

We also selected Ruby, because the official interpreter is written in C. In fact, Ruby objects are mere wrappers around the C objects in which they are implemented. We seek to exploit this design decision in our solution (and in future developments) because it simplifies our design, it helps us to increase seamlessness, and it can possibly be used to address memory management issues.

Ruby	C
interpreted language	compiled language
object oriented	procedural
scripting language	non-scripting language
easy to master	harder to master
dynamic language	static language
concise/more productive	elaborate/less productive
slow execution speed	fast execution speed
automatica memory management	explicit memory management
memory hungry	memory efficient
no overloading	no overloading

Table 5.1: Some language features of Ruby and C compared

This table is by no means a complete listing of all language features of either of the two languages, but it gives a good impression of the differences in feature sets.

5.3 Consequences for the Goals and Requirements

Our choice for interfacing Ruby and C has a few consequences for some of the goals and requirements from the previous chapter.

[2]This is drawn from our own experience and from the enthousiasm of those who have come into contact with it.

5.3.1 Shared Libraries

Because of requirement 2, our multi-language interface between Ruby and C must target the interfacing between two original software components, as well as the interfacing between an original component and an existing component. This requirement can be made more concrete. If we consider any existing component A to be written in C, then there already is a very popular way for publishing such semi-finished products in C: *the shared library*.

A *shared library*, also called *dynamic (link) library*, is a semi-finished product. It is a container in which a certain, but confined, amount of functional value is collected that shares a specific goal or purpose. For example, handling of XML files, handling of secure network connection using specific protocols, and so on. This functionality is packaged separately from an end user product, because ultimately the goal is that multiple end products will be able to share this functionality. This maximizes software reuse and it helps improve productivity. A shared or dynamic link library is loaded and linked into an application (an end product) at runtime, so that the application can make use of its functionality.

So, from our choice of languages and from requirement 2 we derive that our solution must target shared C libraries.

5.3.2 Run-time

Our choice for interfacing Ruby and C also affects requirement 5 (section 4.4.3). In this requirement, we require that our solution must allow for easy software deployment and delegate portability related issues as much as possible.

Since portability related issues are intrinsic to software development in C[3], it is not possible to relief the end user of portability related issues so long as he writes C source code. Ruby, however, is more like Java, where the execution environment is ported, rather than the individual application.

Furthermore, there are a lot of existing software modules available as shared C libraries that are already ported to, and that can be deployed on, a number of target platforms. If we want to relief the end user of portability related issues, we must present a solution to which the end user can delegates these portability related issues, so that it allows the end user to stay in the already easy, and portable friendly, Ruby environment.

One solution that we can think of that allows for all of this, is to let the solution establish the interface between Ruby and C in *run-time*. Because once our solution is confirmed to be working on a target platform, it should work for establishing an interface with any shared C library on that platform.

[3]This is, of course, less so with C than with, for instance, assembly languages, but stil.

5.4 Considered Alternatives

Before we set out to develop our own solution, a number of alternatives have been considered, two of these will be discussed in this section.

5.4.1 Java & The Java Native Interface

There are a number of arguments that can be used to invalidate the choice of Java and The Java Native Interface to meet our goals and requirements. These are both theoretical and technical in nature.

5.4.1.1 Theoretical Arguments

To start with our first argument: Java is not a *scripting language*.

There are a number of scripting engines readily available for Java, so one could say: "Why not interface Java to a scripting engine?" These scripting engines, unfortunately, are all of the "*embedded type*" that we previously discussed in section 3.1.2.1.

In a multilingual interoperability solution, this would make Java, not C, the lower-level language. But Java and its execution environment (JRE) are baded on a *virtual machine*. This is is much too high a level to be effectively used to interface with the native operating system. Hence, the existince of the Java Native Interface.

Neither High-level nor Low-level

This brings us to the most prominent theoretical argument with choosing Java as one of the languages in a multilingual interoperability solution: In our notion of higer-level versus lower-level languages, where does Java go? In our sense, it is neither a relatively high nor a relatively low-level language. It is true, Java is not C, but it is not Python or Ruby either; it is something in-between.

This makes it a less than ideal choice for our higer-level versus lower-level multilingual interoperability research.

Next, we will also present several technical arguments that invalidate Java as our choice of language. For the sake of argument, we will disregard the presented theoretical arguments during the discussion of the technical arguments.

5.4.1.2 Technical Arguments

There are also a number of technical arguments that can be made against choosing Java and the JRE for our research. We will discuss the most prominent of these.

Java is, by design, a compiled language. As we have seen, the compilation of java source code results in byte code which is to be interpreted by a virtual machine, the JVM. We see a few problems resulting from this design.

Limited Reflection and Dynamic Programming Support

First of all, Java comes with reflection support, however, this only extends to reflecting existing classes, methods and variables. Although things might be changing a bit in this regard (recall section 3.3), when our research was started, there was no support for creating new classes and or methods at run-time. In Ruby this is, by its very nature, very easy to do and our solution makes use of it extensively.

Extra Native Code

Secondly, the Java Native Interface, requires *native functions* to be implemented in C, after which this is compiled into a shared library. This does not only apply when writing two original software components, one in C, and one in Java, but it also applies to interfacing *existing* software components to Java. This makes it a lot less easy to interface to an existing component, because there is always a man (or rather, shared library) in-the-middle. Furthermore, this shared library must be written in C, and so portability related issues may apply. As an illustration:

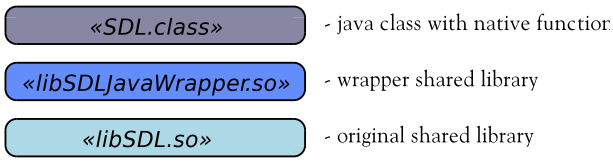

Figure 5.1: Interfacing to existing libraries: The Java Native Interface requires an extra wrapper library.

Programmatic Nature

In addition to the previous argument, recall from figure 3.1 in section 3.2.1, that, in JNI, specially prepared functions have to be written. In order to use this to interface to an existing shared library, these functions must then contain source code that: 1. performs the type mapping from Java to C, invoke the actual library function, convert the result back to a Java type.

This approach leads to a programmatic interface creation strategy, and by this increases, rather than minimizes, the risks that we have previously identified regarding the maintainability of the language interface: *Synchronization errors*, *Heterogenous mapping* and *Black boxing*.

5.4.2 .Net and The Common Language Infrastructure

In the related work chapter, we have already seen that .Net (an implementation of the Common Language Infrastructure), achieves multi-language software development by compiling all supported languages into a single intermediate language. Eventhough the range of languages that are supported by this intermediate language is limited, a lack that is hopefully being remedied (to some extent) by the DLR[4], this approach makes it particularly fit for developing software where all components are developed as part of the framework. (For example, with either two original software components to be written in .Net, or with existing software components that have been written in .Net).

To interface .Net with existing components, such as, shared libraries, that have not been developed within the framework, it comes with two facilities called *DllImport* and *P/Invoke*.

5.4.2.1 DllImport and P/Invoke

One interesting thing that sets .Net apart from a solution such as the Java Native Interface is that it comes with an extra facility that makes it possible to load a shared library and bind its functions at run-time without the need for an extra wrapping shared library. This is made possible by *DllImport* and *P/Invoke*[5]

DllImport allows to dynamically load the shared library at run-time, while *P/Invoke* allows to invoke the functions that may be contained.

For instance, the following source code fragment allows for the successful loading of the standard POSIX' C library, and subsequent invocation of the `usleep` function (which sets the calling application to halt for the given amount of time, given in micro seconds).

```
1  using System.Runtime.InteropServices;
2
3  public class LIBC
4  {
5     [DllImport("libc.so")]
6
7     public static extern void usleep( int ms );
8  }
```

Figure 5.2: Dynamically loading `libc.so` and binding the `usleep` function in .Net.

The dynamic loading and binding of functions works out rather nicely and no extra wrapping library is needed in .Net, as can be seen in figure 5.3.

[4] See section 3.3.
[5] The P stands for *Platform*.

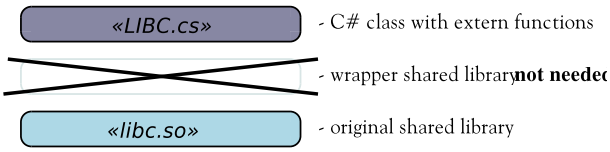

Figure 5.3: Interfacing to existing libraries in .Net: No extra wrapper library is required.

Marshalling

The function argument types are automatically converted from whatever the current language is (e.g. C#) to their C counterparts by a process that .Net calls *marshalling*. The default Marshaller can map all built-in simple C types such as *Integer Types* and *Floating-point Types*.

Things become a little harder when *Structure Types* are required. There is a Marshaller that allows one to define some C structures in .Net to facilitate this to some extent:

```
1  [C#]
2  [StructLayout(LayoutKind.Explicit, Size=16, CharSet=CharSet.Ansi)]
3  public class MySystemTime
4  {
5    [FieldOffset(0)]public ushort wYear;
6    [FieldOffset(2)]public ushort wMonth;
7    .  .  .
8    [FieldOffset(12)]public ushort wSecond;
9    [FieldOffset(14)]public ushort wMilliseconds;
10 }
11
12 class LibWrapper
13 {
14   [DllImport("kernel32.dll")]
15   public static extern void GetSystemTime(
16             [MarshalAs(UnmanagedType.LPStruct)]MySystemTime st );
17 };
```

Figure 5.4: Marshalling Structure Types with .Net.

Unfortunately, soon after we started using the marshalling support of .Net, we encountered the limitations of these facilities.

First of all, source code in .Net (in our example C#), is supposed to be platform independent. However, the explicit field offsets that are often mandatory to get the correct structure layout are, of course, not portable. In other words: There is *no separation of concerns* regarding the portability of the source code. This, as we have argued before, is likely to decrease the productivity of developers using this strategy. Especially if the structures become more complex, as some structure types in C nest other structures, which nest, yet, other structures. How does one even begin to calculate the offsets in these cases?

Yet, the explicit field offsets are merely a nuisance when it comes to the ineffectivity of the standard

(struct) Marshallers offered by .Net.

To illustrate the limitations, suppose that we would like to map the following API:

```
/* xmlDoc:  An XML document.  */
typedef struct _xmlDoc xmlDoc;
typedef xmlDoc *xmlDocPtr;
struct _xmlDoc {
   void            *_private;  /* application data */
   xmlElementType  type;       /* XML_DOCUMENT_NODE, must be second !  */
   char            *name;      /* name/filename/URI of the document */
   struct _xmlNode *children;  /* the document tree */
   struct _xmlNode *last;      /* last child link */
   struct _xmlNode *parent;    /* child->parent link */
   struct _xmlNode *next;      /* next sibling link  */
   struct _xmlNode *prev;      /* previous sibling link  */
   struct _xmlDoc  *doc;       /* autoreference to itself */

   /* End of common part */
   ...
};

/* xmlNode:  A node in an XML tree.  */
typedef struct _xmlNode xmlNode;
typedef xmlNode *xmlNodePtr;
struct _xmlNode {
   void            *_private;  /* application data */
   xmlElementType  type;       /* type number, must be second !  */
   const xmlChar   *name;      /* the name of the node, or the entity */
   struct _xmlNode *children;  /* parent->childs link */
   struct _xmlNode *last;      /* last child link */
   struct _xmlNode *parent;    /* child->parent link */
   struct _xmlNode *next;      /* next sibling link  */
   struct _xmlNode *prev;      /* previous sibling link  */
   struct _xmlDoc  *doc;       /* the containing document */

   /* End of common part */
   xmlChar         *content;   /* the content */
   struct _xmlAttr *properties;/* properties list */
};

xmlDocPtr xmlReadFile( const char *URL,
                       const char *encoding,
                       int options );
```

Figure 5.5: Excerpt from an XML parsing API.

Now, the elements are not simple types anymore, they are *pointers* to yet *other structures*. In .Net it is not possible with the default struct Marshaller to address such structures in a satisfactory way.

One can choose to use P/Invoke's notion of the builtin generic pointer type, `IntPtr`. This allows one to map a pointer inside a structure (or any other pointer, for that matter) as an opaque pointer to a black box. This allows one to use the black box pointer as a stand-in for wherever a reference to that pointer is required.

One *could* use this approach, when the C API is to be used as follows:

```
1  IntPtr mydocument = xmlReadFile( "myfile.xml", null, 0 );
2
3  xmlElementType type = GetTypeOfLastChild( mydocument );
```

Figure 5.6: Example of an intended API usage.

However, suppose the API was intended to be used in the following way (which is not uncommon in C):

```
1  xmlDocPtr mydocument = xmlReadFile( "myfile.xml", NULL, 0 );
2
3  xmlElementType type = mydocument->children->last->type;
```

Figure 5.7: Example of a *different* intended API usage.

The above *"intended API usage"* is not possible when using the generic `IntPtr` as a stand-in.

To address the shortcomings of the simple default Marshallers, .Net allows one to programmatically implement custom Marshallers for each separate structure that is needed, as part of the interface. However, if we would go down *this* path, we are soon back to where we started: Programming rather than specifying, like with the *The Java Native Interface*.

The ineffectivity goes actually further than this. To effectively interface with any shared C library's API, one must be able to effectively map any C type: *Function pointers* anyone?

5.5 Existing Work

For *arranged marriages* between Ruby and C, there is existing work. This work uses *Ruby* as a starting point,and it is discussed in the following sections.

5.5.1 Ruby Inline

First of all, there exists a solution called, *Ruby Inline*. Information from the Ruby Inline project page states[38]:

> "Ruby Inline is an analog to Perl's Inline::C. Out of the box, it allows you to embed C/++ external module code in your ruby script directly. By writing simple builder classes, you can teach how to cope with new languages (fortran, perl, whatever)."

We see several problems with this approach. First, it mixes multiple programming languages within the same software module, or in fact, even within the same file. This is almost never a desirable or good idea from the point of view of a proper coding standard.

Next, people are actually faced with constructing the interface by hand, which is not a good idea from maintainability point of view.

Then, since C and Ruby are always inter-mixed, even when interfacing to existing shared libraries, this approach puts the burden of portability on the user that develops the multilingual software application.

5.5.2 Ruby Extension API

By itself, Ruby comes with a relatively simple to use extension API interface[6]. The Ruby Extension interface allows users to write extensions to Ruby, in C, by writing specially prepared C functions and objects. These are then made available to be used from inside Ruby.

The Ruby extension API can also be used to bind existing dynamic libraries to C. Because Ruby itself is written in C, and all objects in Ruby are reflected directly by objects in C, this extension interface offers a very powerful solution, because you can let any external C code thorougly interact Ruby's-under-the-hood-C.

Still we see shortcomings similar to the previous solution. To start with, users, again, will have to put effort in the actual construction of the interface. Furthermore, because the construction effort is going to be made from within C, there is an extra penalty, because C is a less productive programming language than Ruby (see chapter 2).

Furthermore, like the previous solution, there is no clear separation of modules by means of an interface, except when binding an existing external C library to ruby, in which case this C library provides this interface.

Finally, to bind such an external C library to Ruby, one needs to create a Ruby extension (written in C) for each such library. These libraries have to be compiled to native code and deployed accordingly (making sure the compiled code works on the target platform). That is, there is no such thing as interfacing shared C libraries to Ruby in only *run-time* using the Ruby extension API, not by default anyway.

5.5.3 Ruby/DL

Coming as a part of Ruby, but mostly abandoned, there is a small Ruby extension called *Ruby/DL*. *Ruby/DL* can load shared libraries that have been written in C, at run-time, in a way similar to .Net's `DllImport`. It also allows invocation of simple functions which is similar to .Net's `P/Invoke`.

We say that it is mostly abandoned, because we have seen very little evidence of it being used in real world examples, and the development on the extension seems to have stopped. We see the following reasons that may have led to this situation:

- *Lack of documentation* - The extension is very poorly documented (a small and insufficiently detailed readme). For details, people have to look at the source code (which is, of course, harder to read).

[6] It is simple in comparison to Java's JNI, which is relatively hard to use.

- *Incomplete* - The extension itself is incomplete, reducing the chances of it being effective in a production environment.
- *Very spartan interface* - It has a very spartan interface, making it:
  - a. not easy to *establish* an interface to a shared library's API
  - b. not easy to *maintain* the interface if the shared library's API changes
  - c. *not efficient in use*: It provides *neither* sufficient encapsulation or seamlessness[7] *nor* does it provide support for dynamic boundaries.
- *Poor performance* - The extension has a poor performance (execution speed-wise).
- *Lack of use* - All previous points lead directly to the situation where, although interesting in idea, the extension is not used much (or even at all).
- *Loss of developer's interest* - This is probably the result of all of the above: If nobody uses it and new things come on their path, developers lose interest.

Starting Point for Our Research

This is where our research starts. We have taken some of the dynamic loading ideas of .Net's *DllImport* and *Ruby/DL* and then created a new a solution. We have set up a multi-language development framework that, unlike all existing solutions that we have seen so far, *attempts* to address *all* of the goals and requirements set out in the previous chapter.

5.6 Main Thesis

Now that we have completed our language selection in this chapter and discussed its consequences, we can combine this with the most important goals and requirements from the previous chapter to form the main hypothesis of this Master's Thesis:

Main thesis:

> "Is multi-language software development in Ruby and C possible using a run-time language interface under the condition that the interface is easy to establish, maintainable, efficient in use and effective in all required circumstances?"

In the next chapter we will attempt to address this thesis in a clear and structured manner. The chapter is then followed by an experiments chapter to verify and validate our results.

[7] It suffers from problems similar to .Net's custom marshalling.

Chapter 6

DLX

In this chapter we present our solution to the multi-language interoperability problem in which software components written two different programming languages, *Ruby* and C, are to be interfaced. Our solution is called *DLX*, which stands for ***D****ynamic* ***L****oading e****X****treme*.

We shall describe our solution in a few steps. First we describe its *field of application*, after which we will discuss the *design decisions* that we have taken. Then we shall describe, in detail, *the architecture* of the solution that we created. This followed by several sections that highlight specific details of our solution.

6.1 Field of Application

DLX is a solution to the multilingual interoperability problem (as opposed to polylingual interoperability problem, see section 2.1.2) and it should be classified under the non-distributed approaches from chapter 3.

Our solution makes it possible for two software components, one written in C, and one written in Ruby, to be interfaced. It can be used to interface two *original* software components or to interface an *original* software component to an *existing* software component[1]. The existing software component may be, or may have been, developed without taking multi-language interoperability into account.

Generality of the Solution

While our solution currently focuses on interfacing the higher-level, interpreted, language *Ruby* to the lower-level, compiled, language C (qualifying for the particular name Ruby/*DLX*), there is no reason to believe that it cannot be equally well applied to languages similar to *Ruby* (such as *Python*, *Perl*, etc.). This would then lead to particular names, such as, Python/*DLX*, Perl/*DLX*, etc.

We do restrict ourselves to C as the lower-level, general purpose, programming language, for reasons set

[1] The existing software component must be a shared or dynamic library.

out in more detail in section 5.2.

6.2 Design Decisions

6.2.1 Ruby-centred Development

The first decision that we have made is to put the higher-level language (which is Ruby in our case) at the *center of development*. Software components written in the lower-level language (C in our case) are then seen as an add-on.

This decision is reinforced by our 80/20 conjecture from section 2.2.6. This conjecture leads to a software development approach, where use of C is to be restricted to only computational intensive portions. All other (supporting) source code is better written in de higher-level and more productive programming language which, in our case, happens to be Ruby.

In a more concrete aspect this decision leads to a typical modus operandi, where a *Ruby script* is executed, which at some point of execution loads one or more shared libraries (written in C) into memory.

6.2.2 Run-time

The second decision is to do it all in *run-time*. We already identified one possible argument for choosing run-time over compile-time in section 5.3.2. In the end, each of the following three arguments has led us to choose for run-time over compile-time:

1. Ruby itself is an interpreter-based language, making it a very run-time oriented language. Letting *DLX* work in run-time connects with this in a very natural way.

2. Choosing for run-time, like .Net's DllImport c.q. P/Invoke and unlike Java's JNI, voids the need for an extra shared C library to establish the interface. This way, once Ruby/*DLX* is available on a target system, the functionality of any shared C library can be disclosed to Ruby, and to the end users, without ever having to write or compile a single line or code.

3. As we identified earlier, choosing for run-time, is one way to implement the separation of concerns regarding portability. Because once *DLX* is confirmed to be working on a target platform, it should work for establishing an interface with any shared C library on that platform.

6.2.3 *Specification* not Programming

We liked the way external shared libraries are interfaced using .Net's DllImport c.q. P/Invoke. Unlike Java's JNI and Ruby's extension API, in .Net the multi-language interface is *specified* rather than *programmed*. Therefore we also choose to use *specification* over *programming*.

However the specification that is allowed in .Net is woefully inadequate to establish an interface with just any shared C library. For this it is too limited; important features, like proper support for Structure Types, Union Types, Array Types, Function Pointer Types (e.g. callbacks) is lacking.

In addition to this, a multi-language interface between .Net and an existing shared C library often requires explicit offsets to be given as part of the specification. This does not allow for the separation of concerns regarding portability. The *Ruby/DL* extension that we introduced in section 5.5.3 contains an algorithm –which we improved upon– that allows to determine these offsets automatically for any type.

API Specification Syntax Similar to C

We have chosen for an API specification syntax that is *very similar* to the original C API specification syntax. Such a specification makes it easy to specify a new interface, and it helps to minimize the three maintainability risks that we identified in section 4.3.2:

1. *Synchronization errors -* The similar specification will make it as simple as possible to reflect changes in the shared library's C API back to the *DLX* specification. (There are also other factors that can cause synchronization errors. These shall be addressed separately later in this chapter in section 6.6.2.2.)
2. *Heterogenous mapping -* The similar specification prevents a heterogenous mapping that would have been possible if the interface were to be programmed by several individuals. Programming gives freedom, specification limits such freedom.
3. *Black boxing -* In contrast to having an ad-hoc programmed solution for every shared library that is interfaced to Ruby, *DLX* is a separate solution, with its own maintainers. The simple specification helps, because it will always be clear where the boundary between Ruby and C is.

However, as we stated previously, using a specification rather than programming limits freedom. We must be cautious that our specification, unlike .Net's which appeared to be too limited, *still* allows us to effectively establish an interface to any shared C library.

C API Specification Syntax, But Not Quite

Although we have chosen for an API specification syntax that greatly resembles the original API specification in C, it is not *exactly* the same syntax.

First of all, our syntax uses a blend of C, inside the original Ruby syntax. Ruby has proven to be a very flexible language and creative use of the language has allowed for doing many things that were not anticipated up front. By choosing to use this blend of C inside original Ruby syntax, we hope that some of Ruby's flexibility may become useful to *DLX*, some day, in a way that we cannot see at this point.

Secondly, our syntax allows for some extra specification to be added which can be useful for a number of things:

C

```c


#ifndef __EXAMPLE_LIB_H__
#define __EXAMPLE_LIB_H__



typedef struct Simple
{
   char     c;
   short    s;
   int      i;
   long     l;
   char*    string;
   float    f;
   double   d;
} Simple;



struct Bitfield
{
   int      bf1 :  4;
   int      bf2 :  3;
};

typedef struct Bitfield Bitfield;

typedef struct NotSoSimple
{
  Bitfield*        bf;
  Simple*          simple1;
  Simple**         list;
  struct Simple    simple2;
  char* (*simple_callback)
                   (char* s);
  char* (*notsosimple_callback)
                   (Simple* s)
                   ;
  int (*notsosimple_callback2)
                   (NotSoSimple* nss)
                   ;
} NotSoSimple;



int simpleFunction( double d );
NotSoSimple* loadNSS( char* filename );
void printSimple( Simple* simple );

#endif /* __EXAMPLE_LIB_H__ */
```

DLX

```
#!  /usr/bin/env ruby
require 'dlx/DLXImport'

module ExampleLib
 extend DLXImport

 dlload 'example' #load the dynamic c library

 class Simple < struct "Simple",
 [
   "char",    :c,
   "short",   :s,
   "int",     :i,
   "long",    :l,
   "char*",   :string,
   "float",   :f,
   "double",  :d,
 ]
 end
 typealias( "Simple", "struct Simple" );

 class Bitfield < struct "Bitfield",
 [
   "int :  4", :bf1,
   "int :  3", :bf2,
 ]
 end
 typealias( "Bitfield", "struct Bitfield" );

 class NotSoSimple < struct "NotSoSimple",
 [
   "Bitfield*",       :bf,
   "Simple*",         :simple1,
   "Simple**",        :list,
   "struct Simple",   :simple2,
   "char* (*simple_callback)(char* s)",
                      :simple_callback,
   "char* (*notsosimple_callback)
                      (Simple* s)",
                      :notsosimple_callback,
   "int (*notsosimple_callback2)
                      (NotSoSimple* nss)",
                      :notsosimple_callback2,
 ]
 end
 typealias( "NotSoSimple", "struct NotSoSimple" );

 extern "int simpleFunction( double d )"
 extern "NotSoSimple* loadNSS( char* filename )"
 extern "void printSimple( Simple* simple )"

end
```

Figure 6.1: A side-by-side comparison of C versus DLX.

- *Optimization* - For example, we have used extra specification to introduce a form of late binding to improve start up performance.
- *Increasing seamlessness* - For example, in Ruby the size of an array is part of the array object, in C it is not. Introducing extra specification, will allow us, in some cases, to improve the encapsulation of C arrays when used from Ruby. (For a more detailed explanation see 6.4.7.)
- *Memory management* - We experimented with connecting Ruby object finalization to C object lifecycle and memory management. In this experiment we use extra type specification that is not part of the C API specification (see section 7.6).

Example DLX Specification

Putting all things together, one ends up with something that looks like C inside Ruby. However, unlike Ruby-inline from section 5.5.1, it is used only for specification purposes, no program (source) code of multiple languages is being mixed.

To get a preview of what it looks like, we give an example side-by-side C API specification (e.g. a header file) versus a *DLX* specification in figure 6.1.

6.3 The DLX Architecture

In this section we describe how we put all of the previous decisions together and what is needed to make everything come together: *The DLX Architecture*.

First we give an architectural overview of DLX and its components. Then we will shortly describe how they work together in typical use. The second half of this section is dedicated to the description each of the components in more detail.

6.3.1 Architectural Overview

In figure 6.2 we give an overview of DLX and its components. In the center, *DLX* is shown with all of its components. To the right we schematically depict an example shared library that is to be loaded. To the left we give the Ruby/*DLX* code that is used to establish the interface.

To use an external shared C library from Ruby using *DLX*, first a Ruby module is created that acts as a namespace entity. This namespace entity is where all symbols and related type information, that are contained in the shared library, are bound to (①). To make this possible, the module must be declared to `extend DLXImport`. This sets up the module to be used as the binding entity described above. It also makes certain functions (methods) available that are used to establish the interface (②).

Once the module has been properly created, the *Library Locator/Loader* is invoked using the `dlload` method to locate and load the library (③).

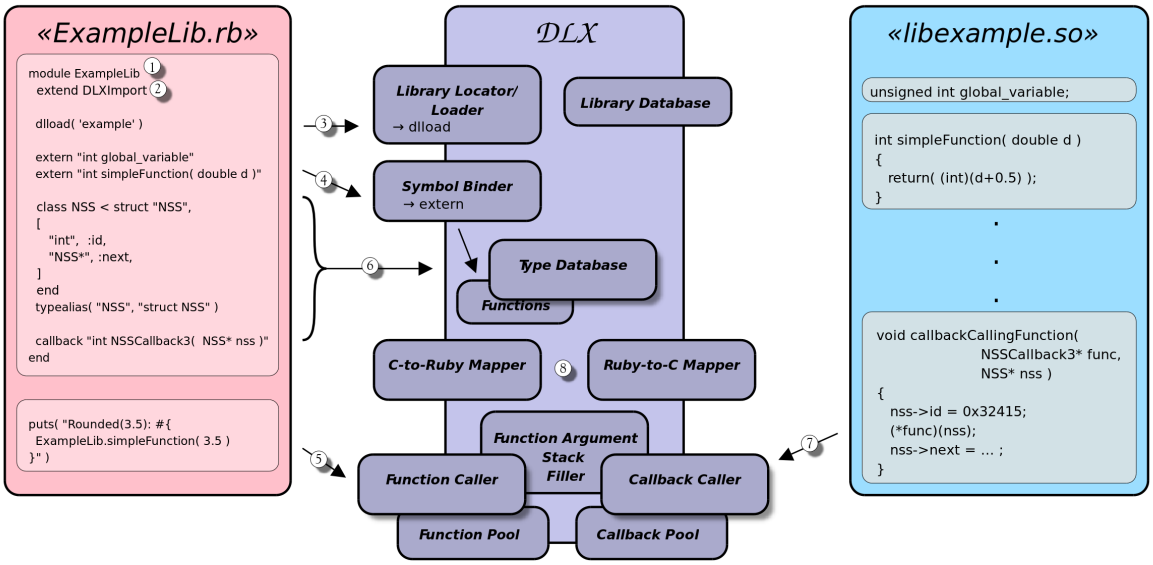

Figure 6.2: The *DLX* Architecture

After the library has been loaded, one can start to disclose its functionality to Ruby using the `extern` method. This invokes the *DLX Symbol Binder* which locates the functionality inside the loaded library and makes it available to be used from Ruby (④).

The functions that have been bound by the *Symbol Binder* are now ready to be invoked. Invoked functions are handled by the *DLX Function Caller* (⑤).

However, at this point, one can only bind fairly simple functions or global variables that only use built-in types such as *Integer Types*, *Floating Point Types*, or *Arrays* of these types.

In order to be able to bind functions or global variables that make use of more complex types such as *Structure Types*, *Union Types*, or *Callbacks*, these datastructures have to be introduced into the *DLX Type Database* (⑥).

Some shared libraries contain functions that invoke callback functions as part of their algorithm. *DLX* comes equipped with a *Callback Caller* component, that allows such functions to perform callbacks to pure Ruby methods, without requiring any additional C code; everything is performed at run-time (⑦).

We conclude out overview of the *DLX* Architecture by noting, that all marshalling is performed behind-the-scenes by the two type mapping components: The *C-to-Ruby Mapper* and the *Ruby-to-C Mapper* (⑧). (The actual type mapping is discussed in detail in section 6.4.)

6.3.2 Library Locator/Loader

The Library Locator/Loader –as the name implies– consists of a *Locator* part and a *Loader* part.

The *Locator* is used to find the correct library on the target system. To this it faces several challenges caused by Operating System differences:

- First of all, it has to cope with *naming* differences; a shared library in Windows™ may be called `SDL`.*dll*, while that same library on most POSIX compliant operating systems is called *lib*`SDL`.*so* and on –yet another popular operating system– MacOS X, the library is known under the file name *lib*`SDL`.*dylib*.
- Secondly, the Locator has to deal with *location* differences: Libraries on Windows™ are located typically in the `%SYSTEM%` and `%PATH%` paths, or inside the current directory; libraries on GNU/Linux are usually found by scanning the current directory, `$LD_LIBRARY_PATH` or one of the paths configured in `/etc/ld.so.conf`.

Once the correct library has been found on the target system, the *Loader* is used to dynamically load the library into memory. The loader also faces problems due to differences in the functions that are provided by the Operating System to this extent:

- On GNU/Linux, the facility *dlopen* is used to this extent.
- Windows has a similar facility called *LoadLibrary*.

The *DLX* end user, however, is shielded from all this Operating System trickery by the public interface of the Locator/Loader pair in the form of `dlload` as we saw previously. The *DLX* end user only provides the Operating System independent name, regardless of the actual Operating System on which Ruby/*DLX* is currently running. (In our example, the Operating System independent name would have been "SDL".)

6.3.3 Symbol Binder

A shared library can hold two types of objects:

1. global variables; and
2. functions (in compiled form).

After the library has been loaded into memory, these objects become accessible at certain locations within this memory. To find out *where* each function or variable is located in memory, every shared library comes with a *symbol table*. It is this table that links the original names (as they appear in the source code) to corresponding addresses in memory.

To read out the symbol table, each of the dynamic library loading facilities from the previous section comes with its own counterpart:

- In Windows™ the symbol table counterpart of `LoadLibrary` is called *GetProcAddress*[2].
- On GNU/Linux the symbol table counterpart of `dlopen` is called *dlsym*.

Each of the above functions returns the address of (read: a pointer to) the C object that they refer to.

After performing basic well-formedness tests, the *DLX Symbol Binder* takes two different actions based on the type of the object that the symbol refers to.

1. If the object is a global variable, then the object is wrapped inside a DLXGlobal object, that takes care of referencing and (un)wrapping Ruby and C objects using one of the *Mapper* components (denoted by ⑧ in figure 6.2).
2. If the object is a function, then more complicated actions are taken. These are described below.

To bind a C function to Ruby, the Symbol Binder performs a type sanity check using the Type Database, to ensure that no types are used that have not yet been inserted into the database. Meaning that, if no complex datastructures have been declared, only simple types such as *Integer Types*, *Floating Point Types* or *Arrays* of these types can be used as part of the function's prototype.

After the sanity check has been performed, the processed function prototype is input into the Type Database. Then, a pure Ruby function is created with the symbol's name, that references the processed prototype, so that it may be used later by the *DLX Function Caller*.

6.3.3.1 Late Binding

Every binding takes up space in memory and incurs a start up penalty. Because shared libraries (and their APIs) are typically much larger and provide many more functions than the average program will every use, the *DLX Symbol Binder* has the ability to *delay the binding* (see figure 6.3).

```
1    extern "int simpleFunction( double d )", delay
```

Figure 6.3: Delaying a binding to a function in *DLX*.

Whenever a function's binding is delayed, there is only a simple test performed to verify well-formedness of the function prototype. No proxy functions are created or added to the Type Database and no type checks are performed. The function prototype of a function of which the binding is delayed is collected and stored as-is, until it is time to be processed.

The binding is delayed until the first time the function is actually used. This is made possible by a Ruby facility that calls one particular method whenever a certain method could not be called because it did not exist at the time it was called. The particular method is called *method_missing*.

[2] In Windows CE™ it is called *GetProcAddressA*.

Consider for example, the module `ExampleLib` from figure 6.2. Whenever a method is called on `ExampleLib` that does not yet exist, it is caught by its `method_missing` method. By extending `DLXImport` (in step ① of the overview), the default method (which raises an exception) has been overridden by a function, that forwards the failed call too the *Symbol Binder*. The Symbol Binder then checks if there previously has been declared an external function by that name whose binding was delayed, if this is true, then it performs the binding, in the same way, as it would have done at program start. If there was no such function by that name, there must be some other error, in which case the Symbol Binder raises an exception, and program execution is terminated by Ruby[3].

6.3.4 Function Caller

Once a shared library's function has been bound to Ruby, it is ready to be invoked, just as one would invoke an ordinary (pure) Ruby method. The delegation of the function's invocation to the actual C function is the primary task of the *DLX Function Caller*.

The *DLX Function Caller* component consists of two subcomponents:

1. The *Function Argument Stack Filler*; and
2. The *Function Pool*.

These two components are functionally very related and we will discuss their details in the following two subsections.

6.3.4.1 Function Argument Stack Filler

Ruby/*DLX* consists of only a single binary[4], there is never any additional code required to bind a shared C library to Ruby via *DLX*. In order to achieve this, *DLX* must be equipped to call any function regardless of its prototype. However, with all those shared C libraries, some written and some still unwritten, it is not hard to imagine that there are nearly an infinite number of prototype combinations possible for any number of arguments.

Of course it is unfeasible for *DLX* to incorporate all of these possible prototypes into the binary. To make do with far less functions in the final binary, *DLX* reconstructs its own argument stack, a pushdown stack, that is created in local address space. Eventhough it is not part of the language specification itself, Harbison III and Steele Jr. [44] state that it is common practise for implementors to do this.

All arguments that are passed to the *DLX Function Caller* are converted into their C counterparts using the *Ruby-to-C Mapper* and are then pushed onto the stack, taking into account any mandatory language-specified alignment and/or widening or narrowing casts. It is the sole task of the *Function Argument Stack Filler* to ensure that this happens per the C language specification or as close as possible.

[3] Unless the exception is caught, of course.
[4] A Ruby extension that is called `dlx.so`.

6.3.4.2 Function Pool

Because it is not reliable –or portable, for that matter– to use the above the pushdown stack directly to pass the arguments to any function coming from the external shared library, we use a trick, of which we even believe it to be Clean C (or as close as possible): We use a *Function Pool* of proxy functions, each taking an increasing parameter stack size. Every proxy function then performs an actual hard-coded invocation of the shared library's function, which is passed as a pointer, by manually passing all individual bytes as `long int` values of our reconstructed argument stack as parameters to the function. A source code fragment is given in figure 6.4.

```
 1  long long_caller0( generic_long_function* function, long args[MAX_ARGS] )
 2  {
 3         return( (*function)(  ) );
 4  }
 5  long long_caller1( generic_long_function* function, long args[MAX_ARGS] )
 6  {
 7         return( (*function)( args[0] ) );
 8  }
 9  long long_caller2( generic_long_function* function, long args[MAX_ARGS] )
10  {
11         return( (*function)( args[0], args[1] ) );
12  }
13  long long_caller3( generic_long_function* function, long args[MAX_ARGS] )
14  {
15         return( (*function)( args[0], args[1], args[2] ) );
16  }
```

Figure 6.4: Excerpt from the *Function Pool*: Actual invocation using proxy functions.

Return Types

There is one thing that is not addressed by the push-down stack approach and that is the handling of the return type. To put in short: it is not possible to invoke a dynamic function using a proxy function whose return type is not compatible[5] with the return type of the dynamic function itself. Therefore the pool has to be extended with proxy functions for each differently sized return type that is to be supported. In the current release of Ruby/*DLX* return types of `sizeof(long)` and `sizeof(double)` are supported[6].

Function Pool size

While the current architecture poses no limitation to the Function Pool size, thus limiting the maximum argument stack size of the functions that can be invoked by the *DLX Function Caller*, the Function Pool *is*, however, created at compile-time and is hard-coded. If functions are to be called with exceptionally large parameter lists in their prototype, then *DLX* will not be able to call such functions without recompiling with a larger Function Pool size. Having said that, the current release of *DLX* can handle function prototypes that have an argument stack size of upto 80 bytes (or 20 `long int`s),

[5] For type compatibility within C, see section 5.11 of [44].
[6] These are, respectively, four and eight bytes on a 32-bit intel architecture.

Although it is an unorthodox approach, the above strategy has not yet produced any non-working *DLX* binary, eventhough *DLX* has already been tested on a variety of different machines, operating systems and architectures (even more than we have formally tested and verified to be working in section 7.2.1).

6.3.4.3 The invocation process

Now that the underlying details of *Function Caller*'s invocation are known, we can simply summarize the invocation process itself by the following six steps:

1. Retrieve the function prototype information from the *Type Database*[7].
2. Using the *Ruby-to-C Mapper*, map all parameter types to their equivalents in C.
3. Invoke the *Function Argument Stack Filler*.
4. Select the appropriate proxy function from the *Function Pool*.
5. Invoke the proxy function and capture the return value.
6. Invoke the *C-to-Ruby Mapper* to convert the value back to a Ruby value.

6.3.5 Type Database

As stated previously in the overview, to bind functions that use more complex datastructures, such as, *Structure Types*, *Union Types* or *Callback Types*, these datastructures have to first be introduced into the *DLX Type Database*.

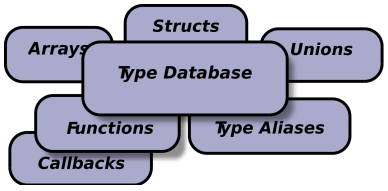

Figure 6.5: The *DLX Type Database* and its major components

The *DLX Type Database* can hold information about the following six C types:

1. *Structure Types*,
2. *Union Types*,
3. *Array Types*,

[7] To speed this up, a local cache in C is used.

4. *Function Types*,

5. *Callback Types*, and

6. *Typedef Names* in the form of *Type Aliases*.

Although some components cache direct references to relevant type information in order to speed things up a little, the *Type Database* is the central location to keep track of all type information. It is the responsibility of the *Type Database* to ensure that all type information that is contained is consistent (i.e. it does not contain *incomplete* types). As such, it plays a large role in the error prevention measure that we call *Type Closures* (see section 6.6.3).

6.3.6 Callback Caller

6.3.6.1 Callbacks

Some shared libraries come with one or more callback calling functions. A callback calling function α is a function that expects to receive a pointer to a C function β that is supplied by a calling function γ. The supplied C function (β) is then invoked –or *called back* if you will– mid-function as part of function α's control flow, temporarily giving the control to the supplied function β. See figure 6.6 for a schematic representation of the control flow.

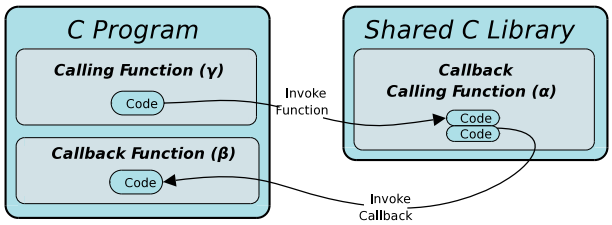

Figure 6.6: Callbacks in C.

6.3.6.2 Communicating Callbacks

There are two ways for the calling function γ to supply the required function pointer β. It can either supply the value *directly*, as one of α's input parameters, or it can supply them *indirectly*, where the callback function β is supplied in the form of a field of a *Structure* or *Union Type*.

In *DLX*, callback constructions are handled by the *Callback Caller*. *DLX* supports callback calling functions that require callback functions to be supplied either indirectly or directly (see also section 6.4.10).

6.3.6.3 Multi-language Callbacks in *DLX*

Unfortunately, callback constructions that involve more than one language are more complex than was previously shown in figure 6.6. The *DLX Callback Caller* distinguishes four different control flows that can occur using callbacks in a Ruby/C combination as can be seen in figure 6.7.

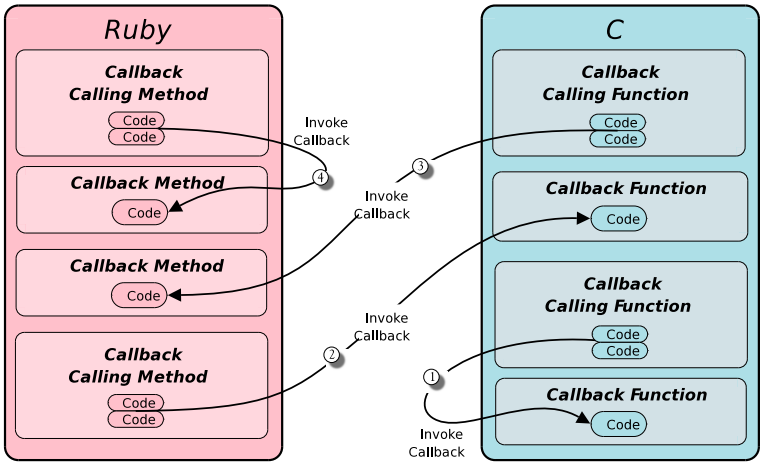

Figure 6.7: The four possible callback control flows.

The four flows that are distinghuised are:

1. Inside a C function, a callback is invoked that still points to a function that is also residing on the C side (①).
2. From inside a Ruby method, a callback is invoked that points to a function that exists in C (②).
3. From inside a C function, a callback is invoked that points to a method in Ruby space (③).
4. From inside a Ruby method, a callback is invoked that points to a method that also resides in Ruby space (④).

6.3.6.4 Callback Pool

In *DLX* to supply a callback function to a callback calling C function, one cannot simply supply a Ruby method instead; Ruby methods are interpreted source code while C functions are executed as compiled source code. To supply a callback function one can only supply a pointer to compiled C code. So again, like the *Function Pool*, we create a *Callback Pool*. The Callback Pool consists of a collection of hard-coded and compiled-in proxy functions that can delegate the control flow to interpreted Ruby methods after which the result is then returned to the callback calling function.

The actual invocation process is very similar to the *Function Caller* invocation process[8], which was discussed earlier. For more information we refer to that section for the details.

[8]One small difference may be, that some proxy functions are able to call Ruby methods instead of C functions.

Callback Pool Size

Because the *Callback Pool* consists of precompiled proxy functions, and since there is no easy or portable way of adding new proxy functions at runtime, there *is* a limit to the number of callback functions that can be in use at one particular moment in time. Once this limit has been reached, an exception will be raised by *DLX*. There is no theoretical limit to the number of callbacks that *DLX* can support, however, if more callbacks are required for a particular application, recompilation will be necessary. In the current release *DLX* supports up to one hundred callbacks that may be in use at one particular point in time.

6.4 Type Mapping: From C to Ruby and Back

In this section we describe how we map C types to Ruby (and vice versa) using the *DLX C-to-Ruby* and *Ruby-to-C* object mappers. We will also explain why this proves to be such a challenging task, and how we solve, or intend to solve, many of the problems we have encountered. For several harder mappings, a small discussion section is added to discuss the current mapping, its shortcomings and possible future improvements.

6.4.1 C Types

According to *The C: A Reference Manual* (Harbison III and Steele Jr. [44]) the C programming language specifies the following nine types (in no particular order):

1. Integer Types
2. Floating-point Types
3. Pointer Types
4. Array Types
5. Enumerated Types
6. Structure Types
7. Union Types
8. Function Types
9. The Void Type

Furthermore, in C, variables must be declared before they can be used. The declaration is used to assign one of the above types to such a variable (with the exception of the *Function Type*; a variable can only be declared as a pointer to a function). The type systems is used to detect relatively simple errors at compile time. You cannot assign a value to a variable whose types are not compatible. Sometimes a type cast is needed enforce type compatibility, however this is not always possible.

Special names can be given to specific instances of (combinations of) the above types, introducing *new* types, which can be seen as *aliases* to the actual types. This is done using `typedef` which invokes the typedefinition facility.

Values occupy memory space. The amount of memory that is used can differ per type. To determine the amount of memory space that a value of a specific type will need, the `sizeof()` operator can be invoked.

For more detailed information on C and its types, we refer to the above book. However, we will go through each of the C types in the following sections as we describe how we mapped the C types to Ruby and vice versa.

6.4.2 Ruby Types

There are two ways in which you can look at the type system used by Ruby. The first way is to look at it from the language side. The other way is to look at it from the way it is implemented. Ruby itself is implemented in C and to understand the way we have mapped C types to Ruby, both must be understood in a little more detail.

Ruby Types 1: As Seen From the Language Itself

In Ruby all values have a type, variables do not. Furthermore, all values are considered to be objects and all variables are references to objects. Any value or object can be assigned to any variable. All objects are descendants of the Ruby class `Object`[9].

Ruby's notion of a constructor is embodied by a special function, called `initialize`. When a new object in Ruby is instantiated from a class (using the `new` method), then `initialize` is automatically invoked as part of the new object's initialization.

To Ruby, a class is merely a way of encapsulating a collection of attributes and operations (methods) on these attributes. In fact, to Ruby, public attributes and methods are the same thing. However, to Ruby, a *class* is not the same as a *type*.

Ruby doesn't use *nominal* typing as many other imperative programming languages do. Ruby uses, as it's followers call it, *duck typing*: "If it walks like a duck, acts like a duck and sounds like duck, then it probably is a duck".

Duck typing is a form of *structural* typing, in which the type of an object is not specified by the name of the object's class, but it is specified by the methods that are defined for a certain object, and the semantics that go with these methods: Classes and their names are just a convenient way of addressing certain instances, or implementations, of these collections of attributes, methods, and semantics.

This means, that two object, one object of class A and the other of class B, can be considered to be of the same type, if both classes provide exactly the same methods and semantics. However, if only the methods

[9] `Object`, in turn, is a descendant of `Kernel`, but this detail bares little importance.

are the same, while the semantics of (some of) these methods are different, then the types are *not* the same.

In the light of this, it is unlikely that `Bankrobber` and `Artist` are of the same type, since `Bankrobber.draw`(gun) and `Artist.draw`(painting) are unlikely to have the same semantics (if they would, it would have probably ended disastrous for the `Artist`).

Ruby Types 2: Implementation Detail

Another way of looking at Ruby types is by the way they are implemented. As already stated in chapter 5, Ruby itself is implemented in C. All Ruby objects, as seen from C, are of type `VALUE`, which is `typedef` defined as the *Integer Type* `unsigned long`.

In the implementation of Ruby there are two different types of values, leading to two distinct ways of interpreting `VALUE`s. Usually a `VALUE` is to be interpreted –and subsequently type cast– as a pointer to a C structure[10]. The C structure is the actual (implementation of the) object.

However, there is an exception to this rule: `Fixnum`, `Symbol`, `true`, `false` and `nil` are directly stored as part of the `VALUE` (i.e. encoded within the `unsigned long`). By using some pointer trickery, and the assumption that all pointers point to a memory address that is aligned on four or eight bytes (for 32-bit and 64-bit architectures respectively). This assumption guarantees that the low 2 bits in a pointer will always be zero, making them available to be used for distinguishing immediate values from non-immediate values.

Please note that, to Ruby itself, the distiction between immediate and non-immediate values is transparent. Any notion of pointers is always kept hidden from the user.

6.4.3 Integer Types and Floating-point types

We start our description of the type mapping between C and Ruby with two relatively simple types: *Integer Types* and *Floating-point Types*.

Integer Types

In C, there are three sizes of signed integer types, denoted by the type specifiers `short`, `int` and `long`, in nondecreasing order of size. C itself does not specify the range of integers that the above types represent, only that a `short` shall be representing a smaller maximum integer value than an `int` shall represent and so on. For each signed integer type there exists a corresponding unsigned integer type.

To Ruby there exists only one (signed) Integer Type, which may be implemented in different ways. On implementation level, there exists the `Fixnum`, which is an *immediate* value, holding at most

[10] This requires `sizeof(void*) == sizeof(long)` for Ruby to be compiled, which is also an important assumption in the current implementation of *DLX*.

`sizeof(long)-1` bits[11]. There also exists an Integer Type implementation called `Bignum`, an immediate type, holding an arbitrary amount of bits (which is only limited by the currently available memory, or limitations implied by the operating system on which the interpreter is running).

With regards to Integer Types, Ruby/*DLX* performs a simple conversion from C to Ruby. The conversion is facilitated by Ruby's C extension interface. If values are small enough (in bit size) a conversion is made to a `Fixnum`, if the values are too large, a conversion is made to a `Bignum`. Because Ruby's `Fixnum` is unable to contain unsigned values larger than `sizeof(long)-2` bits, they are converted to `Bignum`s whenever a value crosses this boundary.

The character type `char` is also an integral type. It is always converted into a `Fixnum`.

Floating-point Types

There are two sizes for representing floating-point numbers in C: `float` and `double`, for single and double precision. A third permissible floating-point type specifier is `long double`, but it is currently not implemented as part of Ruby/*DLX*. With regards to the range of values that each type specifier can represent: The same assumption that applied to the integer type specifiers also applies to the floating-point type specifiers.

Ruby has only one Floating-point Type, `Float`, which, unlike the `Fixnum`, is *not* an immediate value, making its use somewhat costly. Eventhough the `Fixnum` is not an immediate type, from the point of view of a Ruby program, it however *is immutable*; you cannot alter a Ruby floating point number directly, you can only create a new Ruby floating-point number (which is initialized with a different floating-point number in C).

Floating-point types in C are always converted to Ruby's `Float` type using Ruby's C extension interface. When translating a Ruby `Float` back to C, the type information from the *DLX* specification is used to automatically determine the original type.

6.4.4 Pointer Types

In C, it is possible to pass objects by their actual value, or by passing a reference to that object (i.e. the memory location of that object). These two forms are called: *Call by value* and *call by reference*, respectively. It is common for simple types, such as, Integer Types and Floating-point types, to be passed by value, while other, larger[12], objects are passed by reference.

As we have discussed previously in the beginning of this section, in Ruby, pointers are only visible at implementation level. Their use inside a Ruby program is kept transparent to the user.

It is possible in C to create a pointer out of pretty much any object, and of any type, by simply taking the memory address location of that particular object. To understand how this is mapped to *DLX*, the

[11] The least significant bit is reserved as distinguishing factor between immediate and non-immediate values.
[12] This property makes them expensive to pass around by value.

easiest way is to distinguish different pointer types and discuss how each of these are mapped. In *DLX*, we distinguish the following five types of pointers:

1. *Pointers to objects of Integer Types and Floating-point Types*
2. *Pointers to objects of Structure Types and Union Types*
3. *Pointers to functions*
4. *Pointers to arrays of objects*
5. *A generic pointer type*

Of these, only the first and last type will described in this section. Internally, *DLX* handles objects of types 2, 3, and 4 always as pointers. Their details are discussed later, when we describe the mapping of these particular types.

Pointers to Objects of Integer Types and Floating-point Types

A pointer to an object of either an Integer Type or a Floating-poing Type is mapped, in *DLX*, as an array with a single element. This happens either automatically, for example, as the return type of a function, or when accessing a structure's field, or it happens explicitly.

For instance, if a pointer to an object of such a type must be passed to a function, then an array of size one is created in Ruby and passed to the C function. This allows access to the value from inside Ruby and lets *DLX* pass the value as a pointer.

We have not found a way to make this usage more seamless than simply explaining the API semantics of such a function in Ruby terms as: This function expects to be passed an array of type integer (or floating-point).

The Generic Pointer Type

In *DLX* there also is a *generic pointer type*. A pointer of this type can represent any pointer in C. It is implemented in the `DLXCPtr` class, the class that acts as the super class of the particular classes for each of the remaining object types: Structure Types and Union Types, Function Pointer Types, and Array Types.

Any C object can be wrapped inside an object of class `DLXCPtr`, by means of the `wrap` class method, that is especially created for this. Subclasses have their specific implementations of this method. Furthermore, although normally kept transparent to users, for the sake of not compromising the effectiveness of our solution in any circumstances, access to the memory address of any `DLXCPtr`, or derivative, can be revealed by means of the `addr` and `addr=` methods[13].

[13]The last form allows one to mutate an existing `DLXCPtr` object, so that it can be made to point to a different address.

Object comparison versus object equality

In Ruby, there is a difference between object *comparison* and object *equality*. Object *comparison* (implemented by the "==" method) is used to test if two objects represent the same value. The result of such a test depends on the semantics of class of objects that is compared. For instance:

```
1  >> 1 == 1
2  => true
3  >> [] == []
4  => true
5  >> [1,2] == [2,1].reverse
6  => true
7  >> 1 == 2
8  => false
9  >> [1,2] == [2,1]
10 => false
```

Object *equality* (implemented by the "`equal?`" method), however, is used to test if two objects are literally the same object. (One might say it is Ruby's equivalent of a pointer equality test, but it also applies to immediate values). For example:

```
1  >> 1.equal?(2-1)
2  => true
3  >> [].equal?( [] )
4  => false
5  >> [1,2].equal?( [2,1].reverse )
6  => false
```

Here, we can actually see from the first result that integers are really immediate types, since the literal numeric value 1 is really the same object as the result of the expression $(2-1)$. We also see the difference between object *comparison* and object *equality* for the following tests, which now evaluate to `false`, while they previously evaluated to `true` in the comparison test.

The distinction between object *equality* and object *comparison* is very important, to Ruby and also to *DLX*, because –as we shall see later– more than one Ruby object can wrap the same object in C[14]. Thus, to test for object equality, it was not sufficient to simply test for Ruby object equality. We therefore reimplemented the "`equal?`" method to test for pointer equality of the underlying pointers instead.

6.4.5 Structure Types

To form new types of more complex datastructures, C has a *structure* type (for an illustration, please see fig. 6.1). A structure type is a compound* type, similar to the types known as *records* in other programming languages[44]. Its elements are named, and usually called *fields* or *members*. Each field of a structure must have a type. All types that have been defined up to the point, where the structure is defined, can be used as the type of a field. A structure in C is typically used to encapsulate related data objects.

[14] A feature that unfortunately also complicates memory management a bit.

* *Com·pound*

noun:	a whole formed by a union of two or more elements or parts
adj.:	consisting of two or more substances or ingredients or elements or parts
source:	Merriam Webster's Online Dictionary

Ruby Equivalent

In Ruby, as in most other object oriented programming languages, a *class* is used to encapsulate both datastructure *attributes* and *behaviour*. Behaviour is implemented by means of *methods*, which are simply an equivalent of functions in C, except that they are always defined as part of a class or a module.

An attribute (called an instance variable) in Ruby can have various access modifiers. Such a modifier can be used to restrict access to the attribute to certain types of objects. From a usage perspective, a publicly accessible instance variable in Ruby is exactly the same as an ordinary method; to Ruby or any Ruby programmer, there is no distinction between either a publicly accessible instance variable on one hand, and ordinary methods on the other.

Structures in *DLX*

To increase the level of seamlessness, in *DLX*, for every structure type that is defined, a similar class in Ruby is defined. Furthermore, to prevent accidental specification errors, the class is created as a subclass of a dynamically constructed super class. This class is created by the *DLX* `struct` command. The dynamic super class (a "`DLXStruct`") cannot exist by itself, enforcing a named subclass to be created.

The input to the `struct` command is deliberately made remarkably similar to the original C specifications for reasons set out at the beginning of this chapter. An example of a structure definition can be seen in figure 6.1.

DLX Structures and Seamlessness

The chosen specification syntax increases seamlessness in various ways:

- First of all, it bridges the (datastructure) type namespace, in C, to Ruby, making the type names available on both sides of the fence.
- Secondly, as we have seen above, classes are the natural language constructs in Ruby to encapsulate datastructure attributes and behaviour; from them, objects are instantiated. Objects are the building blocks of any object oriented programming language. And so end users of Ruby/*DLX* (i.e. programmers) will instantly be able to use them.
- The classes can be extended with pure Ruby attributes and behaviour (methods). In other words, they behave just like any ordinary Ruby class. To end users there is no difference.
- Furthermore, a class offers at least the same level of encapsulation as a C struct.

Object Instantiation and Initialization

In *DLX*, a C structure object is internally always handled as a pointer to that structure object. From a usage perspective, the process is similar to the wrapping of generic pointers from the previous section.

There are a few differences, however. These differences are mainly in the way:

1. *DLX* structure objects are handled internally;
2. existing C objects are wrapped;
3. new objects are created; and
4. member fields are accessed (since a generic pointer has no member fields).

To an end user of *DLX* there are three important methods available for descendants of the `DLXStruct` class:

`DLXStruct.wrap(`*<address or DLXCPtr>*`)` - This method is used to wrap an existing C pointer[15] inside a newly created Ruby object of the corresponding type. This is somewhat similar to the `DLXCPtr.wrap` from the previous section, with the difference that an explicit subclass is meant here. For example, `Simple.wrap(`*<address or DLXCPtr>*`)` (see figure 6.1), would create a new `Simple` object that wraps the existing C object.

`DLXStruct.size` - This method will give the size that an object of the corresponding structure type would hold in memory. For example, `Simple.size` (see figure 6.1) would give the size of a C `struct Simple` in run-time, similar to C's `sizeof` operator, which works at compile-time.

`DLXStruct.new(`*<parameter list>*`)` - This method creates a new Object in C using C's memory allocation facility (`malloc`). The size of the allocated memory is determined using the `size` method (see above). The resulting allocated memory is then wrapped inside an object of the corresponding class. For example, `Simple.new(`*<parameter list>*`)` (see figure 6.1), would create a new `Simple` object that wraps the newly allocated memory space. The optional parameter list is passed on to the constructor's invocation (`initialize`).

Attributes and Member Fields

For each field of a structure, a method with corresponding name is dynamically created. The input is, of course, taken from the *DLX* specification that is part of the input to the `struct` command. For of the above each *get* method, a corresponding *set* method is created, which is the name of the field method postfixed with an "=".

The returned type of the accessed field is used to automatically wrap the value inside an object of the corresponding Ruby class.

[15] The C pointer may be specified by a numeric value or a `DLXCPtr`.

While in general, member field access of a C structure is implemented by means of the above get and set methods, there are a few exceptions. In order to increase seamless, we have added special default behaviour for some member field accesses to make the common case easy. Member field types that are applicable to such altered behaviour are:

- Pointers-to-pointers
- Nested structures
- Embedded strings
- Embedded callback functions

We shall discuss the first three in the following subsections, the fourth option, embedded callback functions, is discussed as part of the section on callback functions (see section 6.4.10).

Pointers-to-pointers

When *DLX* encounters a pointer-to-pointer, then it assumes that an array is referenced. The returned value is thus an array of one level of pointers higher. This means, that a `Simple**` is returned as an array with elements of type `Simple*`, and a `Simple***` is returned as an array elements of type `Simple**` (which is an array of an array of elements of type `Simple*`), and so on. Naturally, the fact that the array contains pointers to `Simple` structures is still shielded from the Ruby programmer.

Of course, the default behaviour, which works for most cases, may be wrong from time to time. In such a case, an explicit type cast is unavoidable (see section 6.4.12).

Nested structures

In C, a structure can be nested inside another structure, which, in turn can be nested inside yet another structure, and so on.

```
1  struct Point { ...  }
2
3  struct Triangle
4  {
5     struct Point p1;
6     struct Point p2;
7     struct Point p3;
8  }
```

Figure 6.8: Nested structures in C.

Nested structures are fully supported in *DLX*. With nested structures, however, the limitations of what can be made seamless and what not, come into view. Some things we can hide from the user by applying a clever automatic translation (based on the information taken from the *DLX* specification). Other things, unfortunately, become very dangerous from an end user perspective.

To explain what we can make seamless, consider the C structure in figure 6.8. In C it is common for such a structure type to be typically handled as a pointer. When the structure is used, and the fields of the nested `Point` structures are

accessed, this results in a rather awkward mix of both `.` and `->` operators:

```
1 Triangle* t = newTriangle();
2 t->p1.x = 0; t->p1.y = 1;
```

In Ruby, there is no difference between operators `.` and `->`. In fact, Ruby does not even have the `->` operator, only the `.` operator. For the sake of seamlessness, *DLX* users always use the `.` operator, *DLX* internally automatically switches between the right operation depending on the current context. So despite the fact that `p1` is a nested structure, Ruby programmers would write:

```
1 t = Triangle.new
2 t.p1.x = 0; t.p1.y = 1;
```

Unfortunately, sometimes hiding information can become dangerous. Because of seamlessness, we have made *DLX* such, that all member field accesses appear the same. This means that there is no difference between accessing a pointer to a structure, or accessing a nested structure. But this can be a dangerous thing, for instance: What is the correct behaviour for assigning new values to a nested structure?

The current behaviour of *DLX* does, under the hood, what is typically done in C: a memory copy of the new structure on top of the nested structure. This, of course, results in duplication of information. In C, this is explicit, so if something strange happens, the programmer can be blamed. In Ruby, however, this may lead to strange behaviour. Because, all of a sudden, two objects exist, and a mutation in in an attributes of one object, is currently *not* reflected to all objects that have become cloned copies of the original object.

We think that this is an example of where the boundaries of seamlessness come in to view. One possible solution may be to keep track of all cloned objects and keep them synchronized under the hood. However, we are unsure if, and if so, how this may impact the effectiveness of *DLX*. For instance, when it needs to interface to a shared library that doesn't expect such automatic behaviour –or shall we say: auto*magic*?

Embedded strings

Similar things that we discussed about assignment of nested structures can, of course, also be said about assignment of strings that are embedded into a structure (see figure 6.9). The current implementation is also similar to that of assigning nested structures. Again, memory is copied, except that this time the actual copy is performed by C's string processing utilities (i.e. `strcpy`).

```
1 struct NameRecord
2 {
3    char given_name[25];
4    char surname[25];
5 };
```

Figure 6.9: Embedded strings inside a structure.

6.4.6 Union Types

A *Union Type* in C provides a means of combining several different types into a single common type. This new type can be made to hold values of any of the types that were used as part of the definition of the union type. For instance, the union type definition in figure 6.10, can hold either an `int`, a `struct MouseEvent*`, or a `struct ButtonEvent*`.

C

```c
union SDL_Event

{
  int type;
  struct MouseEvent* mouse;
  struct ButtonEvent* button;
}
;
typedef union SDL_Event
       SDL_Event;
```

DLX

```
class "SDL_Event" < union
"SDL_Event"
[
  "int"                ,  :type   ,
  "struct MouseEvent*" ,  :mouse ,
  "struct ButtonEvent*",  :button,
]
end
typealias( "SDL_Event",
          "union SDL_Event" );
```

Figure 6.10: An example of a union definition in C and *DLX*.

The *Union Type* is in many ways very similar to the *Structure Type* from the previous section. However, different from the *Structure Type*, a *Union Type* can only contain a single *value* at a time. A value of a union is accessed as a particular type by referencing its respective member field.

Defining union types in C is almost identical to defining structure types, except that the `struct` keyword is replaced by the `union` keyword.

Ruby Equivalent

Ruby does not have an equivalent to C's union type. In Ruby, polymorphism is ordinarily achieved by inheritance, and through mixins[16].

In Ruby's form of polymorphism, however, the objects automatically take the right "shape". Let us explain this.

Consider that there are two subclasses, `Square` and `Circle`, of a class `Shape`. `Shape` prescribes the `draw()` method, which are implemented by `Square` (which draws a square) and `Circle` (which draws a circle). Now suppose a method `makeRandomShape()` returns a random `Shape` objects (which is either a `Square` or a `Circle`), then, whenever a `Shape` is returned, it automatically assumes the *right shape*. By this we mean that, if the `draw` function is invoked on a `Shape` that is, in fact, a `Square`, then a square is drawn, whereas if the `Shape` had been, in fact, a `Circle`, then a circle had been drawn.

[16]The ability extend a class's functionality by including modules with cross-class, reusable, functionality.

In C, with unions, this is not so. While the returned union object would be able to take the shape and form of any of the declared member field types[17], the distinction has to be made manually.

For this, usually a *trick* is used in C: By giving all of the union's field types a common first field, say, a type identification field –such as an enumerated integer for instance–, this type identification field can then be guaranteed to exist, regardless of the actual type of the union. This allows for a distinction to be made regarding the actual type of the union, so that it can assume the correct shape by explicitly accessing the appropriate member field of the union.

Unions in *DLX*

In *DLX*, unions are currently mapped in a way that is similar to structures. This means that the above C-like behaviour is still explicit in *DLX*. We therefore also follow the C specification syntax: To declare a union in *DLX*, simply replace the `struct` keyword with the `union` keyword, the rest is identical.

Because the "common first field" trick appears to be exhibited very often in C, when unions are used (but *not* always[18]), we may be able to introduce extra type specification syntax to automate this in future versions of *DLX*.

6.4.7 Array Types

According to the reference manual, if *T* is any C type except `void`, or "function returning...", the type "array of *T*" may be declared. Values of this type are sequences of elements of type *T*.

In Ruby, an array is an object that can hold a number of arbitrary objects stored as a sequential list.

Arrays in *DLX*

Array support is handled in *DLX* by the `DLXArray` class. In the current stable release of *DLX* a specific subclass of the `DLXArray` is created for any of the following types:

1. Any built-in Integer Type (e.g CharArray, ShortArray, IntArray, etc.)
2. Any built-in Floating-point Type (e.g. FloatArray and DoubleArray, etc.)
3. Any struct or union class that is specified (e.g. PointArray, SimpleArray, etc.)

Like the Structure Type described previously, the array classes can be used to either `wrap` an existing C object as a *DLX* array, or they can be used to create a `new` array of the corresponding type with the specified size.

[17] It can, of course, only do so, one at a time.
[18] Sometimes the distinction is made based on a separate, non-encapsulated object.

Important to note is that, currently, arrays that are automatically wrapped (e.g. from a function call's return value or a structure's member field access) do *not* have their size encapsulated as part of the *DLX* array object. Their size must be derived separately. (How this is done depends on the used C API.)

DLX Arrays and Seamlessness

To come up with a *DLX* Array implementation and mapping has proven to be quite a challenge[19]. This is because there are quite a few difficulties to overcome. We shall highlight these as briefly as possible in the next couple of subsections.

Differences

One reason why it has proven to be quite a challenge to map C array types to Ruby arrays is because there are so many differences between the two.

Differences in Content-type restrictions - Ruby arrays can hold any mix of objects of different types. In C, an array is always declared as "an array of type *T*", where *T* can be any type barring any exception set out in the beginning of this section. This implies that the array can *only* hold objects of that *particular* type.

Differences in Encapsulation - In Ruby, the size of an array is encapsulated inside the array object, a C array does not encapsulated its size. It is just a portion of memory allocated where a sequence of elements of a certain type *T* can be stored. *How many* elements can be stored at the particular memory location is only known implicitly, or explicitly by storing this size information in a separate variable. Subsequently, in Ruby, you cannot make a mistake by referencing elements beyond the array's boundary, since this is checked by the interpreter. In C references outside an array's boundary is not prohibited, and this may lead to undetermined behaviour.

Differences in Size boundaries - In C, the size of an array, that is, the number of elements that can be stored in the array object is fixed. If more elements need to be stored, then extra memory must be explicitly reserved before such elements can be stored. In Ruby, the size of an array is dynamic, with only a small but configurable starting size. If more elements are stored in the array, it is automatically resized to hold a larger maximum number of elements.

The `char*`: Byte Array or String?

Another challenging mapping is that of a very special type of array. Namely, the *array of char*, also denoted in pointer notation as `char*`. Very often, this type of array denotes a sequence of characters, or a string of text. Because arrays do not encapsulate their size, this poses a problem for handling strings in C: how many characters to process in this string? As a workaround to this limitation, strings in C embed their size by means of string termination character (`'\0'`).

[19] The current development release of *DLX* is now on its third reimplementation.

Since Ruby is implemented in C, it is not suprising that ordinary strings in Ruby are also implemented under-the-hood as this sequence of characters. This makes it more straightforward to establish the mapping... but only once it is known that it is a string of text that is stored inside a certain array object.

Sometimes, however, such an array of `char`(acters) means little more than just that: a sequence of bytes. In such a situation, there is no string termination character.

So the difficulty for *DLX*, here, is not: "How to map a `char` array to either a byte array or a string?" Rather, because of the ambiguity in the type specification, it is: "How to determine whether the considered object is either one or the other?"

In the future, it may be a good idea to introduce a different specification to distinguish between real strings and sequences of bytes (e.g. `char*` versus `byte*`). (This would be a trade-off between the maintainability and the usability of a multi-language interface established with *DLX*.)

Currently, by default, `char` arrays are mapped to byte arrays. However, to keep our interface at least 100% effective, we have added a method, `to_string`, to the `DLXCPtr` class that allows any `DLXCPtr` to be interpreted –type cast, if you will– as an ordinary string.

Iterators

One big difference between Ruby arrays and C arrays is the way one typically iterates through their elements. The best way to see this, is by looking at a side by side comparison (see figure 6.11).

C

```c
char* ary[] = {"This", "is",
               "an", "array"};
int size = 4;
int i = 0;

for( i = 0; i < size; i++ ) {
   printf( "%s\n", ary[i] );
}
```

Ruby

```ruby
ary = ["This", "is",
       "an", "array"]



ary.each { |elt|
   puts( elt )
}
```

Figure 6.11: A side-by-side comparison of array iteration in C versus Ruby.

The main difference that can be observed from the source code fragments above is that iteration in C is done by using a for-loop, with an explicit reference to, and subsequent test against, the size of the array. In Ruby, on the other hand, iteration is done by a special iterator function, that receives a `Proc`[20]. Then, for each step in the iteration, the specified `Proc` is invoked with the next element of the array as its argument. The actual size of the array is never explicitly mentioned or considered.

Currently, *DLX* array iteration cannot be performed in a way that is natural to Ruby, because the size

[20] A `Proc` in Ruby is a closure (a type of anonymous function).

```
1  # ary is an instance of a
2  # DLXArray (e.g.  StringArray)
3  size.times{ |i|
4     puts( ary[i] );
5  }
```

Figure 6.12: Array iteration over *DLX* Arrays

of an array is not always encapsulated in a *DLX* array (it is only so, when an array is created explicitly). Therefore it is necessary to fall back on the method depicted in figure 6.12.

On the other hand, it *is* possible to store wrapped C objects inside an ordinary Ruby array, in which case of course, Ruby iteration is simply possible.

Contiguous versus Non-Contiguous

One important aspect of arrays that cannot be kept hidden from the users for the sake of seamlessness is the difference in the way an array can be laid out in memory.

For instance, in C, there is a difference between:

- `struct Triangle ary[10]`; and
- `struct Triangle* ary[10]`.

To illustrate the differences, a schematic representation of the memory layout is given in figure 6.13. Clearly visible is that while one of the arrays is laid out as one large contiguous block of memory, the other array is much smaller, containing pointers to separate objects that may be scattered across a program's memory.

In general, this sort of difference cannot automatically be resolved by *DLX*. Therefore, it cannot be kept hidden from the user.

Because in *DLX* there is no notion of pointers, it becomes hard to explain this difference to the user. However, since we see no other parts of our type mapping having to deal with a similar problem, we have decided to compromise seamlessness of our solution in favor of its effectiveness (which is obviously most important to us).

The compromise consists of the introduction of extra terminology, new to Ruby users, but without explicitly mentioning the notion of pointers (which are still kept hidden and are still automatically dereferenced when needed). The new terminology is taken, as-is, meaning, that from now on, there are simply *two* different types of arrays. The semantics of the function call will make clear as to *which* type of array is required, and they are just supplied as-is.

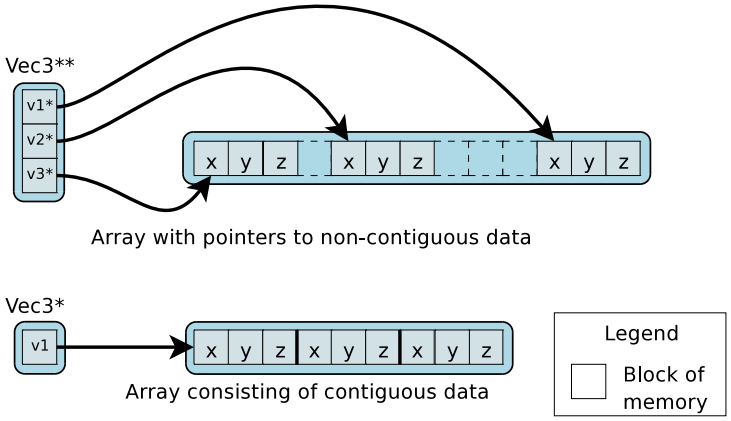

Figure 6.13: Contiguous versus non-contiguous memory layout of arrays

We call the `struct Triangle ary[10]` a *contiguous array* and the `struct Triangle* ary[10]` a *non-contiguous array*. We think that this terminology concisely depicts the differences, without having the notion of pointers becoming too predominant.

With the existence of, now, *two* separate subtypes of arrays, also come the added functions that are needed to make this distinction when arrays are created. So we add two functions that are only defined for arrays to the already known and common functions `new` and `wrap`. In the end, there are four functions to create and wrap C array objects as DLX arrays:

<DLXArray class>.new - This function is used to *create* an ordinary *non-contiguous* array.

<DLXArray class>.wrap - This function is used to *wrap* an ordinary *non-contiguous* array that already exists in C.

<DLXArray class>.new! - This function[21] is used to *create* a *contiguous* array.

<DLXArray class>.wrap! - This function[22] is used to *wrap* a *contiguous* array that already exists in C.

Discussion

Array support in *DLX* is still a moving target. The development version of *DLX* is now on its third implementation for arrays.

It is important to make a distinction between things that simply cannot be made absolutely seamless, and things thay *may eventually* become more or less seamless.

[21] Note the exclamation mark.
[22] Idem.

For example, when interfacing to an existing C API that expect arrays to hold only a specific type of objects, then such an array cannot ever be used to hold just about *any Ruby* object. This is a hard constraint that cannot be worked around.

On the other hand. It may be possible to improve encapsulation of data and size in *DLX* array objects.

Sometimes it is possible to derive an array's size directly from the API (either automatically or using extra type specification syntax). Consider for example the type definition in figure 6.14. Here, the size of the `Point*` array is given in the member field "`count`", if it would be possible to relate these two fields in the *DLX* specification, then we would be able to encapsulate both size and data in *DLX* arrays automatically. There are also other examples for automatically encapsulating size and data in *DLX* arrays, but, due to an ever growing thesis size, we cannot elaborate on this.

```
1  typedef struct
2  {
3     int count;
4     Point** points;
5  } Polygon;
```

Figure 6.14: Sometimes the size of an array is specified in a separate member field.

If encapsulation of data and size in *DLX* arrays is achieved, it opens possibilities for further increasing of seamlessness:

- automatic boundary checking
- automatic resizing
- iterator support

6.4.8 Typedef Names

As we have seen previously in the beginning of this section, special names can be given to specific instances of (combinations of) most of the built-in C types. This is done using the `typedef` command.

DLX has support for something similar, which is called the *type alias*. It is invoked using the `typealias` command. Once a type alias is input into the type database, it can be used anywhere within the embedded C syntax as a stand-in for the original, usually more elaborate, type name. The similarities between the syntax of C's `typedef` command and *DLX*'s `typealias` command are obvious, and translation between the two are self explaining. An example of type aliasing, and its similarities to C's `typedef`, can be seen in figure 6.1.

When type aliases are used in this way, their purpose is mainly to allow the, in the *DLX* specification embedded C API, to be as similar as possible to the original C API specification.

Another use for type aliases is to specify special instances of arrays. For instance, the following source

code fragment creates a new class called `Quaternion` which, when instantiated, will be implemented by a `FloatArray` of size 4:

```
1  typealias( "Quaternion", "float[4]" )
2  Quaternion.new( 1.2, 1.3, 1.4, 1.5 )
```

There are also other uses for type aliases, they shall be introduced later in this chapter (see section 6.6.3).

6.4.9 Enumerated Types

An enumerated type in C, is an Integer Type that is given a special name and meaning and a list of possible values (that are not enforced).

Ruby does not have an equivalent of such an enumerated type.

Enumerated types in *DLX* are therefore emulated. This is done by creating a type alias of the enumerated C type to `int`. This ensures that the *DLX Type Database* will recognise the enumerated type name in subsequent *DLX* specification. Then a Ruby `module` is created with the same name as the type alias. The possible values of the enumerated C type are then specified as Ruby constants inside the created module. For example:

```
1  typealias( "SDL_eventaction", "int" );
2  module SDL_eventaction
3   SDL_ADDEVENT = 0
4   SDL_PEEKEVENT = 1
5   SDL_GETEVENT = 2
6  end
```

6.4.10 Function Types

Function types are handled specially in *DLX* (see section 6.3) and, as such, they are outside the scope of either the *DLX Ruby-to-C Mapper* or the *DLX C-to-Ruby Mapper*.

However, there is an exception: Callbacks *are* handled by the *DLX Type Mappers* and we shall discuss this mapping next.

Callbacks

We have already seen that *pointers to functions*, also commonly addressed to as *callbacks*, in *DLX* are handled by the *Callback Caller*. We have also seen which different forms of muti-language communication via callbacks are supported by *DLX*.

The fact that one can take the *pointer* to a function, turns such functions into objects that can be passed around (in contrast to a mere function in C, which is static, and can only be *bound* to a Ruby namespace

module using the *Symbol Binder*). Because a callback can be passed around as an object, it must therefore be sensibly mapped to Ruby.

DLX distinguishes two ways for a callback to be represented in Ruby:

1. A *standalone* callback object that is not part of any other object.
2. A callback that is *part of* a structure or a union (i.e. in the form of a member field).

To pass a callback function to a callback calling function explicitly, or to receive a pointer to a function as the return value of another function, a standalone callback object must be created in *DLX*. To be able to create such a standalone callback object it must be specified as part of the *DLX* type specification (for example, see line 3 in figure 6.15). This implicitly creates a new class with the callback's name, whose objects, when instantiated, can be used to wrap such callbacks. An example of wrapping a Ruby callback is given in line 5 of figure 6.15.

When a callback is part of a structure or a union[23], it is handled different from ordinary fields. Recall that to a Ruby programmer, public attributes in Ruby are indistinguishable from methods that implement operations or other behaviour.

To increase seamlessness, when a structure's field is accessed that references a callback, this does not retrieve the callback object, but invokes the callback function itself instead (line 11 in figure 6.15). We feel that this behaviour is the most likely behaviour that Ruby programmers would expect. If, on the other hand, such a field must be supplied to a C function that expects a standalone object, then it must be explicitly accessed. This is denoted by a trailing exclamation mark (see line 13 in figure 6.15).

It is also fairly simple to change the callback member field to point to a new callback by either supplying a standalone callback object explicitly, or by supplying a receiver/method-name pair, which will create a callback object implicitly (lines 8 and 9 in figure 6.15).

```
 1  def ruby_callback_method( text ); ...  ; end
 2
 3  callback( "char* (*SimpleCallback)( char* )" );
 4
 5  standalone_callback = SimpleCallback.new( self, :ruby_callback_method );
 6
 7  obj = NotSoSimple.new;
 8  obj.simple_callback = standalone_callback;
 9  obj.simple_callback = self, :ruby_callback_method;
10
11  obj.simple_callback( "Hello World!" );
12  ExampleLib.callback_calling_function( standalone_callback, "Hello World!" );
13  ExampleLib.callback_calling_function( obj.simple_callback!, "Hello World!" );
```

Figure 6.15: In *DLX*, callbacks are mapped to Ruby in several different ways.

[23] For an example, please see lines 36-38 in figure 6.1.

6.4.11 The Void Type

The void type in C has two distinct purposes:

1. To denote the absence of any particular type, where it mainly is used as the return type of a function that does not return a value.
2. As a pointer (`void*`), to denote the generic pointer type.

Ruby Equivalents

In Ruby, every method must return a value, even if the value is `nil` (the Ruby null pointer). Pure Ruby methods are not required to use a `return` statement; if no such statement is given, then automatically the last referenced value is returned, defaulting to `nil`.

The Void Type in *DLX*

In *DLX*, the *Ruby-to-*C and C*-to-Ruby* Mappers automatically convert between `void` and `nil` when appropriate.

Use of `void` as a generic pointer (`void*`) was already discussed in section 6.4.4.

6.4.12 Type Casting

Normally in Ruby, type casting is neither possible nor required. Because of its structural typing, the type of a Ruby object is determined by the methods it supports. Inheritence or mix-in support takes care of the rest.

In C, however, an explicit type cast is sometimes required to interprete an object as a different (but compatible) type. In *DLX*, such type casting is supported by means of the "`wrap`" method that is defined for all of the pointer types. By rewrapping objects inside new Ruby objects of a different type, a C object will be interpreted differently.

Consider, for instance, the following example:

```
1  class Square < struct "Square",
2  [
3   "Point*",     :offset,
4   "Dimension*", :dimension
5  ]
6  end
7
```

```
 8  class ColoredSquare < struct "Square",
 9  [
10   "Point*",     :offset,
11   "Dimension*", :dimension
12   "Color*",     :color
13  ]
14  end
```

Suppose an object `my_square` is wrapped as a "`Square`", but it is in fact more than just a "`Square`", such as, a "`ColoredSquare`". Then, so long as `my_square` remains a square, it is not possible to access the "`color`" member field. For this, a cast is required that rewraps the object `my_square` as a "`ColoredSquare`":

```
1   my_colored_square = ColoredSquare.wrap( my_square )
```

Explicit type casting for simple types such as Integer Types or Floating-point Types is neither possible nor necessary: Type casts for such types are performed automatically by *DLX* in the background.

Discussion

The explicit type casting that is described here, while effective, has a negative impact on seamlessness, since in Ruby the same effects are achieved by inheritence and mix-in support.

As a future improvement, it may be a good idea to research if we can avoid explicit type casts altogether (without affecting the effectiveness of *DLX*) by better intergrating the C types with Ruby's inheritence or mix-in support.

6.5 Memory layout reconstruction

In the previous section, we have seen all the types that are specified for the C programming language. We have seen how C objects were mapped to Ruby objects. Some of the (more simple) C values were directly converted into Ruby immediate values. Other (more complex) C objects were wrapped inside Ruby objects, while the actual C objects remained as allocated in and by C.

These more complex C objects, most notably, the structure types –but some of this also applies to union types and array types– represent a collection of values at certain memory locations.

Because of our decision to establish an interface with a shared C library *at run-time*, in order to access any of the subobjects that may be stored at such a location, *DLX* must also be able to reconstruct the memory layout of these locations *at run-time*.

To correctly do this, we must be able to answer the following questions correctly for an object of any type:

1. *What is the size of an object of this type?*

2. *What are the alignment characteristics of an object of this type?*

They are important questions, because if we get the answers wrong, chances are, that not even the simplest interface can be established.

This section is about answering these two questions. The algorithms that we present here are improved (and corrected) versions of the original algorithms as we derived them from the *Ruby/DL* source code (see section 5.5.3).

6.5.1 Object size and alignment

To understand how the memory layout reconstruction works, it is best to separate the *simple types* from the *non-simple types*.

In this section, a *simple type* is any type of which we can statically determine its size and alignment characteristics at compile time. Simple types are thus:

1. Integer Types
2. Floating-point Types
3. Pointer Types

Subsequently, a non-simple type is any type that is not one of the above types.

To be able to answer the two questions *"What is the size of an object of this type?"* and *"What are the alignment characteristics of an object of this type?"* at run-time, the algorithm that we use consists of acompile-time *and* a run-time part.

The *compile-time part* of the algorithm is used to determine platform-dependent size and alignment characteristics[24] of the simple types. The *run-time part* of the algorithm depends on this to correctly determine the size and alignment characteristics of the non-simple types at run-time.

The algorithm, part I: Compile-time part

1. First, all supported types, simple and non-simple, are given a unique numeric identifier.
2. Then, at compile time, a *size map* is created. This size map is used to map the numeric type identifiers to the actual byte sizes of allocated objects of these types on the target platform. As was explained previously in section 6.4.3, the C language specification *does not dictate* actual byte sizes for any of the integer or floating-point types. The actual allocated size of an object of such a simple type depends on the selected target platform. Therefore it is determined using the compile-time `sizeof` operator. The size map is stored as part of the `dlx.so` library for later reference at run-time.

[24] These depend on the target platform, i.e. the used compiler, operating system, architecture.

3. Also created at compile-time is the *alignment map*. This is a map, like the size map, that is used to store identified alignment characteristics of the simple types, so that they can be looked up at run-time for later reference. To identify these alignment characteristics, a series of introspective structures is created to expose these characteristics (see figure 6.16). By doing this, there is a risk of non-portability, but our algorithm relies on predictability of the alignment characteristics, not on the outcome of a particular alignment characteristic (i.e. the alignment characteristics must not be random, but we do not depend on hard-coded characteristics for just a set of compilers).

```
 1 typedef struct { char c; void* x; } s_voidp;
 2 typedef struct { char c; short x; } s_short;
 3 typedef struct { char c; float x; } s_float;
 4 typedef struct { char c; double x; } s_double;
 5 static int ALIGN_MAP[] =
 6 {
 7  (sizeof(s_voidp) - sizeof(void *)),
 8  (sizeof(s_short) - sizeof(short)),
 9  (sizeof(s_float) - sizeof(float)),
10  (sizeof(s_double) - sizeof(double)),
11 };
```

Figure 6.16: Excerpt of the alignment map.

The algorithm, part II: Run-time part

Once the alignment characteristics of the simple types have been recorded, the run-time part of the memory layout reconstruction algorithm can be used. The run-time part is used to calculate the memory layout of non-trivial objects such as Structure Types. A compound type such as these can have a very different physical memory layout than can be be expected from the type's specification. For instance, it is not uncommon for the type specification in figure 6.17 (top) to be laid out internally like in figure 6.17 (bottom).

The gaps are caused by the alignment characteristics of the used compiler (which in turn depends on things such as target operating system and architecture).

While it may seem fairly random and undoable to calculate such memory alignments in run-time, in reality, the characteristics are very predictable for most compilers and architectures. There are only a few things that need to be taken into account:

- First of all, not a great many number of architectures like to stack types inside a structure without gaps (however, if they would, then the memory alignment characteristics of the built-in types would expose this and alignment will take place accordingly). For technical reasons, it is common that architectures like to align objects of certain types to four (32-bit) or eight (64-bit) byte boundaries.
- For most compilers/architectures, the final alignment, c.q. size, of a structure depends on the alignment characteristics of the largest type it contains.

```
1  struct Foo
2  {
3     char c;
4     short s;
5     int i;
6     char c2;
7     float;
8     short s2;
9  };
```

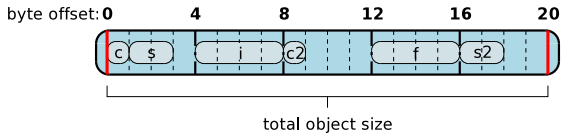

Figure 6.17: Example of alignment characteristics. Top: Example structure in C. Bottom: Physical layout of the structure.

The memory layout reconstruction algorithm that is used by *DLX* depends on the predictability of these characteristics and on the alignment characteristics of the simple types that were exposed as part of the compile-time part of the algorithm.

To calculate the actual memory layout of arbitrary compound types, the run-time algorithm in figure 6.18 is used.

Basically what is done is:

1. For each member field's type, calculate its size and alignment characteristics.
2. If necessary: realign.
3. Update the offset value that acts as the start parameter for the next field.
4. Update the structure's own alignment type.
5. Once all field layout characteristics have been calculated: finish up by calculating the alignment, c.q. size, characteristics of the structure itself using the current structure alignment type (which we discussed earlier).

To improve performance, the calculated memory layout is cached, so that accessing member fields can be done very quickly.

```
1  def align( address, align )
2     res = address
3
4     if( align != 0 )
5         d = address % align
6         if( d == 0 ) # do not align
7             res = address
8         else  # crosses alignment boundary, realign
9             res = address + (align - d)
10        end
11    end
12
13    return( res )
14 end
15
16 def calculate_field_layout( field_type, field_offset, struct_size )
17    orig_offset = field_offset
18    field_size = 0
19
20    field_offset = align( orig_offset, ALIGN_MAP[field_type] )
21    struct_size = field_offset - orig_offset
22    field_size = typeinfo_sizeof( field_type )
23    field_offset += field_size
24
25    return[ field_offset, struct_size ];
26 end
27
28 def calculate_struct_layout( fields )
29    offset = 0
30    size = 0
31    alignment_type = TYPE_CHAR
32    foreach field in fields
33        field_type = type(field)
34
35        offset, size = calculate_field_layout( field_type, offset, size )
36
37        # update the alignment type for the whole structure
38        if( alignment_type != TYPE_VOIDP )
39            if( SIZE_MAP[field_type] > SIZE_MAP[alignment_type] )
40                alignment_type = field_type
41            elsif( SIZE_MAP[field_type] == UNKNOWN )
42                alignment_type = TYPE_VOIDP
43            end
44        end
45
46        size = align( offset, ALIGN_MAP[alignment_type] )
47    end
48 end
```

Figure 6.18: Run-time memory layout algorithm in pseudo-Ruby

Note: The `typeinfo_sizeof()` function in line 22 is not defined here: It returns the size of an arbitrary type using either the SIZE_MAP or a previously calculated type size (for example a structure size).

6.5.2 Bitfields

According to the reference manual[44], C allows the programmer to pack integer components into spaces smaller than the compiler would ordinarily allow (for example, a 3-bit wide integer field with eight possible values). Such integer components are called bitfields.

While not used often, some C APIs actually *do* use bitfields, and because of this, *DLX* supports bitfields too.

Do not bother to try to derive bitfield support from the run-time algorithm given in this section. The version given here is actually a simplified version of the one that is actually in use. The actual memory layout reconstruction algorithm is much more complex because of the bitfield support. We do not give a detailed explanation of the bitfield algorithm in this thesis.

The main reason for this is that, while the algorithm described in this section appears to be working on several compiler/architecture combinations (and it is expected to work on even more combinations than we have been able to verify ourselves), bitfields are likely to cause portability issues.

The reference manual *does state* that, although bitfields are likely to cause portability issues, the packing of bitfields remains *predictable*. And we have found this to be true for at least two different compilers, TCC[22] and GCC[42], albeit each with its own characteristics. We had to derive these by hand[25] (as opposed to the alignment algorithm given in this section, which does this automatically for structures that do not contain bitfields).

So for *DLX*, while bitfields are supported on some compiler/architecture combinations for the sake of effectiveness when interfacing to an API that uses bitfields, it is best to avoid them.

6.5.3 Other uses of the algorithm

Parts of the memory layout reconstruction algorithm are also used by the Function Caller and Callback Caller to calculate the correct offsets when passing function type arguments. (This is done during push-down (argument) stack reconstruction.)

6.6 Error Handling

During software development many things can go wrong. There are many classifications on software errors, but this is not the place to go into the details of such classifications.

When discussing errors and error handling in relation to *DLX* we make two distinctions.

First of all, we think it is useful to make a distinction between two causes for errors, these are:

1. Errors caused by using the language interface itself.

[25] A task that takes about an hour to do.

2. Errors that occur *during* software development on either side of the interface.

Secondly, we think it is useful to make a distinction between *trapped* and *untrapped* execution errors (recall its definition we gave in section 2.4.1).

6.6.1 Trapped and Untrapped Errors in C and Ruby

First we will discuss errors that occur *during* software development on either side of the interface.

In C, some execution errors may be untrapped. Untrapped execution errors in C are often caused by mistakes made during (explicit) memory management. Dangling pointers are one such example that results from bad memory management. This is why execution errors in C often result in a system segmentation fault. Without the aid of debugging tools, a defensive coding style and/or the use of assertions it is often not clear at which point in the C source code the error manifested itself, let alone the location where the error actually originates from.

In Ruby, an execution error is always trapped sooner or later. It is also very clear where an error occurs in Ruby, since Ruby always prints a stack trace in which the most recently called methods are printed along with their locations (file names and line numbers).

There are errors that are trapped both in C and Ruby at program start. These are typically simple errors, such as *syntax errors*. On the other hand, there are also errors that are trapped in C at compile-time (i.e. *before* program execution), which are only trapped at run-time (i.e. *during* program execution) by Ruby. An example can be found in calling a non-existent function or method.

Consequences for Software Development with *DLX*

So what are the consequences of errors made either in Ruby or C?

To Ruby, when a error occurs, it remains trapped and handled as is, so nothing much is different from developing software in Ruby standalone. However, to the untrapped errors in C, some interesting concequences follow from the use of *DLX* in multi-language software development.

Because we call C functions from Ruby, if an error (segmentation fault) manifests itself inside such a function, then the segmentation fault (which is actually the operating system trapping the error) is caught by Ruby. By catching the error, it gives *DLX* the opportunity to display a message in which it says: The error occurred in this C function. Unfortunately, one of two situations may still be the case:

1. An error that originates from one function that manifests itself in the same function.
2. An error that originates from one function that manifests itself in another function.

Still, we think it is interesting to see that, even without the aid of a specialised debugging tool or any

additional programming tricks, we are still able give some indication as to where an execution error in C occurred. In standalone C development (see above), this is not the case.

6.6.2 Trapped and Untrapped Errors in the *DLX* Language Interface

Let us now discuss errors caused by using the language interface itself.

DLX uses a specially prepared specification in combination with a *smart* memory layout reconstruction algorithm to reconstruct the memory layout of arbitrary C types, at run-time, and in a platform independent manner[26]. We go to great lengths to ensure correctness of this algorithm, but still many things can go wrong when using *DLX*; *to err is human* and as with any development activity, errors are always waiting to be made.

We attempt to trap as much of these errors as possible. In order to do this in a structured manner, *DLX* distinguishes the following types of errors:

1. *specification errors* - For example, incomplete or inconsistent type specifications and illegal use of type aliases belong to this category of errors.
2. *usage errors* - For example, when an integer type is passed to a function parameter that expects a pointer type.
3. *synchronisation errors* - These are errors caused by a version mismatch between the version of the C header information on which *DLX* type specification was based and the version of the shared library that is loaded at run-time.
4. *internal errors* - Errors that are trapped because of illegal states (e.g. dereferencing a null pointer while a null pointer is not expected).
5. *external errors* - Errors that are untrapped by any of the above measures, but that are trapped as part of external safety mechanisms that have been built into the loaded shared library. (For example, Gtk and ODE have many of these internal consistency checks. We will get back to this in section 7.3.)
6. *untrapped errors* - Errors that have slipped through the mazes of the net and cause undefined behaviour.

Of these errors the first three are the most likely to occur. Therefore, we will go into these in a little more detail.

6.6.2.1 Specification Errors

As far as specification errors are concerned, basically three things can go wrong.

[26] So far, bitfields are the only feature that require special compiler support.

1. An incomplete type is specified and used before its specification is completed.
2. During translation from the C header specification to the *DLX* specification a structure or union's field is omitted or added.
3. During translation from the C header specification to the *DLX* specification some types are erroneously exchanged (e.g. `char` ↔ `int` or `float` ↔ `double`).

Errors of this nature are likely to confuse our memory layout reconstruction algorithm and errors resulting from this are hard to debug.

The first form of specification errors is trapped by a special *DLX* error measure, called type closures, which basically prevents errors of this nature. Type closures are discussed in detail in section 6.6.3.

Unfortunately it is hard to detect the second and third form of specification errors. But during our test period we have not encountered these errors because of the translation process that we used: First, the relevant type specification was looked up in the C header file. Then, this specification was copy-pasted to our *DLX* specification file, after which we proceeded with the translation of the C type specification to a *DLX* specification. Since these two specifcations are very much alike, this is a straight-forward process.

It appears to be hard to make errors of this kind using this procedure. However future changes to the C API may also allow the second form of specification errors to be made, for instance, when new fields are added as part of the API changes. These errors, in our terminology, are called *synchronisation errors* which are discussed next.

6.6.2.2 Synchronisation Errors

We identify two causes for synchronisation errors:

1. A shared library is loaded with a different version than the version of the C header type that was used for the *DLX* specification.
2. The API changes of a new release version of a shared library are *not correctly* reflected back to the *DLX* specification.

Shared Libraries

The first form of synchronisation errors is not specific to *DLX*. In fact, when using dynamic libraries in general, these types of synchronisation errors can occur. This is one of the reasons why large software vendors, most notably on Windows™, supply their own set of shared libraries, *even* when the library is already present on the target system. In Linux (and other unices), where collections of applications are typically compiled by the various distributions this is less often the case. However, so long as a binary application, that depends on a shared library, is installed on a system with a *different* version of that shared library than against which the binary was compiled, then such a system remains susceptible to these kinds synchronisation errors.

There are, of course, measures to prevent these kinds of errors. The first measure, is taken by the compiler and run-time environment: Whenever a shared library is compiled, it is embedded with a version signature from the compiler. This version signature is compared to the version signature of run-time environment. If there is a mismatch between these two, the library will not be loaded.

It is important to detect this, because if, for example, the compiler makes changes in the way it represents structures or unions in memory (i.e. the alignment characteristics change), then libraries compiled against older versions of this compiler will be incompatible. In *DLX*, the existence of the `dlx.so` library[27] always ensures compatibility of the memory reconstruction algorithm with the current version of the compiler. If the `dlx.so` is attempted to be loaded or when itself is trying to load a library with different alignment characteristics, it will simply not load.

The second measure that can be taken, is not an enforced measure, but many proper libraries use it to detect slight differences between the API version of the library against which a certain application executable was compiled, and the API version of the library against which it is linked at run-time. For this, the measure uses two things: One or more *preprocessor macro definitions* to specify the version of the API at compile-time, and, an API function that returns the version of the shared library at run-time. To illustrate this, please look at the source code fragments given in figure 6.19.

```
1  // MyLib ver.  1.7.1
2
3  #define MAJOR_VERSION 1
4  #define MINOR_VERSION 7
5  #define PATCH_LEVEL 1
```

```
1  // MyLib ver.  1.2.0
2
3  #define MAJOR_VERSION 1
4  #define MINOR_VERSION 2
5  #define PATCH_LEVEL 0
```

```
1  // Application compiled
2  // against mylib ver.  1.7.1
3  #include <mylib.h>
4
5  int main( void )
6  {
7   check_version( VERSION_MAJOR,
8                  VERSION_MINOR,
9                  PATCH_LEVEL );
10  // the rest of this function
11 }
```

```
1  // MyLib ver.  1.2.0
2  #include <mylib.h>
3
4  void check_version( int major,
5                      int minor,
6                      int patchlevel )
7  {
8   if( major != VERSION_MAJOR ||
9       minor != VERSION_MINOR )
10  // error message and exit graciously
11 }
```

Figure 6.19: Many sensible libraries use a library version check as part of their initialization routine. The check is used to ensure that the library against which an executable was linked at compile-time is compatible with the library against which the executable is linked at run-time.

To the left we see an application that is compiled against a *different* version of the shared library than against which it is linked at run-time (which is depicted in source code form to the right).

Now, if there is a mismatch (or incompatible version) detected during a version check, then the library can bail out graciously, without untrapped run-time errors ever having taken place.

[27] The only library that *DLX* consists of.

This approach, as it is shown here in C, can be used in *DLX* in exactly the same way. Except that the preprocessor macro definitions are replaced by Ruby constants.

All of the libraries that we have used during our experiments provide some variation of the above described mechanism and we succesfully used it as part of our *DLX* specification. (For an example, please see section 7.3.)

Reflecting API changes

Another form of synchronisation errors arises, when an API change in the shared library is not reflected back correctly to the *DLX* specification. We argued in section 6.2.3 that one way of reducing this risk is to make use of a specification that is very similar to the original C specification, thus making it as easy as possible to keep both of these synchronised. However, so long as this remains a manual task, errors may occur. We therefore have started to research if it is possible, and to what extent, to automate the translation and synchronisation process. This has proven to be harder than originally anticipated. More on the discussion about the automated approach can be found later in this chapter in section 6.8.1.

6.6.2.3 Usage Errors

DLX currently provides a limited form of detecting usage errors. If a pointer type parameter is passed to a function that expects an integer type parameter, or when a floating-point type value is passed to a function that expects an integer type parameter, the error is trapped in Ruby style with a gracious error message and a method call stack trace.

However, most usage errors are likely to be caused by passing pointers of incompatible types as a function's parameters. While it would be interesting to see if we can trap such usage errors in the future, currently, these type of errors are not trapped by *DLX*.

6.6.3 Type Closures

The order in which types are added to the *DLX Type Database* is fairly strict. This is partially caused by the dynamic environment in which the specification is done; at many points during the specification, ordinary ruby (program) code can be executed. Because there is no "*end of specification*" indicator[28], types that are already in the database and functions that have already been bound might be invoked or used before new types are introduced into the database. Therefore the types that are contained in the database must always be consistent. If this were not the case, incomplete types may be used which would certainly wreak havoc in our memory layout reconstruction algorithm.

DLX prevents accidental errors in the *DLX* type specification, by attempting to create a *type closure* at every new definition of a type or typealias.

[28] We do not want such an indicator, as it limit the freedom of Ruby's dynamical programming style.

A *type closure* in *DLX*, is a type-specific form of a *transitive closure* where the relation R in *x* R *y* can be seen as *x consists of* type *y* (or *x references* type *y*).

Let us explain this by means of an example. For the sake of simplicity, we reuse the distinction from section 6.5, that every type in *DLX* is considered either a *simple* type or a *non*-simple type.

Simple types, such as, Integer Types and Floating-point types, do not reference any other types, they are seen as *terminals*. Non-simple types,such as, a Structure Type or a Union Type (but also a Function Type, etc...) are then *non-terminals*, since they reference certain other types as part of their type specification.

We have seen previously in section 6.5, that the memory layout reconstruction algorithm relies on the fact that all such referenced types are known to correctly determine any type's size and alignment characteristics.

If this were not the case, then it is very possible to get at a point where a type is used from an incomplete specification, wreaking havoc. Consider, for example, the following type definition:

```
1  class Point < struct "Point",
2  [
3     "int", :x,
4     "int", :y,
5  ]
6  end
7  typealias( "Point", "struct Point" );
```

In the above case, we have a "`Point`" class that depends on the size of a type `int` –and the way this is laid out in memory– to determine how to layout the type "`Point`" itself. In this case, "`Point`" references only simple types, but what happens if it were to reference non-simple types as well? Let's investigate...

Consider the following non-simple type:

```
1  class Circle < struct "Circle",
2  [
3     "Point", :center,
4     "int",   :diameter,
5  ]
6  end
```

Other than the previous example, this example uses both a simple type (`int`) and a non-simple type ("`Point`"). To use this type in *DLX*, it must know exactly how to layout this type in memory, so that you for example may access the diameter field. Fortunately, because we have previously defined the non-simple type "`Point`", *DLX* already knows of this type, and so it knows how to access the diameter field, without trying to access it at the wrong address in memory. But what would happen if we reference a non-simple type that hasn't been defined yet? This is the tricky part of the story, so again, we will investigate...

First we define yet another non-simple type:

```
1  class Square < struct "Square",
2  [
3     "Dimension",    :dimension,
4     "Point",        :offset,
5  ]
6  end
```

In this example, we use two non-simple types, the first is one that we've already seen before: The "`Point`". However, the second type is one that we have not seen before: The "`Dimension`". (Note: The fact that we use only non-simple types, is not the issue here, we may as well have used some simple types in this example as well.) The real problem is that, because *DLX*'s memory layout reconstruction algorithm does not know either the layout or the size of the memory that is occupied by "`Dimension`", it also cannot say anything about the *size* and layout of the memory that is occupied by "`Square`". For example, if it were a *type alias* to a pointer, it's size would have been `sizeof(void*)`, however if it would have been a type alias to a structure (like the definition for "`Point`"), then the offset would have likely been something like twice the `sizeof(int)`[29]

And there we are at the root of the problem:

If *DLX* cannot perfectly determine all types at the time they are referenced then it is said that it cannot perform a formal type closure, that is, there are still some types referenced that have not been previously defined. And because such incomplete types will interfere with the correctness of our memory layout reconstruction algorithm (possibly causing some very bad unexplainable segmentation faults), *DLX* prevents further execution of the program and terminates it by raising an appropriate error.

Forward Declarations

There is one exception to the strictness of the order in which types are to be introduced into the database.

If a certain structure or union *A* contains a reference to another structure or union *B*, and if *B*, in turn, contains a reference to *A*, this would result in a typical chicken and egg problem.

In C, where the problem occassionally arises as well, there is a solution to this: The *forward declaration*. A *forward declaration* allows one type (*A*) to be declared upfront. This type declaration, results in an incomplete type definition. After the first type has been declared, type *B* is defined, referencing *incomplete* type *A*. Then the previously declared type *A* is defined, leaving no incomplete types.

DLX also supports forward declarations, by special use of the `typealias` command: Whenever a type alias is made to a type of `void*`, *DLX* allows the type to be specified later. When –at some later point in the specification– the type becomes fully specified, *DLX* will reflect the change back to all earlier occurrences of the particular type.

[29] We intentional leave out any alignment difficulties here, that also play a role.

```
1  typealias( "stuct ForwardDeclarated*", "void*" );
```

Figure 6.20: Forward declarations in *DLX*

DLX forward declarations, however, are slightly different from C. C allows for a struct or a union to be forwardly declared, whereas *DLX* only allows a *pointer* to a struct or union to be forwardly declared. This is intentional, because it is the only way to guarantee *Type Database* consistency at all time[30]. These differences, however, do not appear to impact the effectiveness of *DLX* as will be shown in the next chapter.

Defining Only What You Need

Apart from making a *forward declaration*, there is another use for making a *typealias* to `void*`. Making a *typealias* to `void*`, allows for –as we would like to call it– an *opt out* from having to specify a complete API using *DLX*, when only a small portion is required. The *opt out* allows one to define only those portions that are needed. This is very useful in early stages of development or binding creation, as it allows one to add functions or types on a when-needed basis only. Because the specification is fairly similar to C, it does not form much of a distraction, so that users can focus on the original problem (which is probably not: *creating the multi-language interface*).

During the specification of the API, the type closure algorithm will always make sure that the database is consistent, avoiding errors.

For every *new* type, for which a specification is required, there is an option to *not specify* this type any further. However, because it is `typealias`ed to a `void*`, it can still be used with its original name (e.g. `Circle*`), as part of the specification of other types. This means that there is no need to textually substitute the specific pointer type (e.g. `Circle*`) with an explicit `void*` in the *DLX* type specification text (i.e. inside the Ruby source file). This would contaminate the *DLX* type specification if, at a later date, the type (e.g. `Circle*`) is really added to the *DLX* type specification.

Once the type is aliased to a `void*`, it is considered to be generic *DLX* pointer (a `DLXCPtr`). As previously described in section 6.4.4, any C object can be turned into a generic pointer, and any such pointer can be stored inside a normal Ruby variable using the `DLXCPtr` wrapper object. And, like any other object, it can be passed as a parameter to function that expects that (implicit) pointer type.

6.7 Penalties

The solution that we present in this chapter with *DLX*, incurs some penalties. Some of these penalties are caused by the current implementation, while others are the result of the design choices that we have made.

[30] This is because the size of a `void*` is fixed and always known.

6.7.1 Execution Speed Penalties

First of all, because we have chosen a run-time approach in establishing a language interface, we suffer a small penalty in execution speed. Our generic library loading Ruby C extension requires a little more processing as would have been the case with a specific, tailor made solution constructed directly using the Ruby C extension interface. However, a small penalty is considered acceptable, given the fact that our interface is much better maintainable and Ruby programs requiring certain libraries can be used directly on the target platform[31] without the need of creating, compiling, porting, and distributing the extra Ruby extension library.

The current implementation also suffers an extra penalty in execution speed, caused by current limitations of the Ruby extension interface. However, this subject is handled in more detail as part of the discussion on the performance experiments, that we have conducted, in section 7.5.1.

6.7.2 Memory Penalties

DLX also suffers from memory penalties. These penalties are not so much caused by the current implementation; they are more fundamentally caused by our choice of a run-time language interface.

In C, the type information from the header files is used only at compile-time. *DLX* (and Ruby) require type information to be available at run-time. For *DLX*, this type information must be in the *Type Database* otherwise the *Ruby-to-C* and *C-to-Ruby* Mappers cannot perform the mapping. For large APIs with a lot of types and functions, the memory penalties can become quite large.

6.7.3 Startup Penalties

Because of the type information, *DLX* also suffers from startup penalties. For this, two causes can be identified:

1. All functions are bound during startup.

2. All type information is parsed, processed and kept consistent during startup.

6.7.4 Possible Solutions

We suggest two possible solutions to minimize the impact of both the memory and startup penalties.

[31] We assume here, that the original library is available for that platform, obviously.

Solution 1: Delay Binding

In Ruby it is possible to catch a failed call to a method if the called method does not exist (i.e. it is not defined). To do this, one uses the generic Ruby method `method_missing`, which is always called in this case. The arguments to this function are the name of the failed method, followed by each of the original parameters that where passed to the missing method. We use this method to introduce some form of late binding, by only parsing and binding functions when they are called. This obviously introduces a new problem: every first call to such a method will have some delay. See section 6.3.3.1 for more information about *DLX* and this form of *late binding*.

Solution 2: Profiling

Both problems, but especially the second problem, can be solved by introducing some form of profiling. This profiler would then extract all used types and methods for a particular program. Only those type definitions and methods that are actually used are parsed and kept in memory. This profiler will somehow need to automatically solve the problem of the Type Closures from section 6.6.3.

6.8 Advanced Topics

In this section we present advanced topics that we, because of their advanced and complicated nature, have not yet researched in great detail. However, we have explored some of the possibilities of our solution with respect to each of these topics, so that they may provide a starting point for future developments.

6.8.1 AutoDLX

Because the *DLX* type specification is so remarkably similar to the original C header specification, one soon starts to wonder, whether it would be possible to automate this to some extent.

We have investigated this. Unfortunately, it appears to be less straightforward than it at first glance seemed. We *did* implement a solution that, when applied to the *OpenGL API*, successfully and automatically transformed the OpenGL and related header files into a *DLX* specification. In general, this current implementation, however, is insufficient. When we applied it to other C API specifications we encountered the following difficulties:

- C APIs are note pure.
- The C grammar is context sensitive.
- Problems with to automatic renaming.

What we mean by these will be discussed in the following subsections.

C APIs are not pure

The first problem that we encountered is that: *C APIs are not pure*. By this we mean that, in general, a C API is not purely specified in C. Instead a mixture of C type and object specifications is mixed with a collection of preprocessor definitions.

The preprocessor definitions manifest themselves in various ways:

- *constants -* Constants are typically in the form of `#define MYCONSTANT 0x9761438`. The OpenGL API has many of these, but since they are easily translated into Ruby constants, they are not much of a problem.
- *macros -* Preprocessor macros are a harder problem, because these can be seen as a form of inline functions, that are literally (textually) expanded at various locations in the source code, right before compilation. Ruby *DLX* does not use any C code, nor does it require a compilation step. This means, that any macros that are part of the API will need to be automatically translated into pure Ruby, a non-trivial task.
- *platform-dependent conditionals -* Another hard problem is caused by *platform-dependent conditionals*. These are the `#ifdef __WINDOWS ... #elif defined(__LINUX) ... #endif` conditional preprocessor instructions that select the appropriate type specification for the current target platform. While it is possible to mimic these definitions by hand in Ruby using simple if-then-else conditionals, it is much harder if you want to do this automatically without prior knowledge of each and every possible preprocessor conditional. A complicating factor is that very often a new preprocessor conditional is introduced as part of the parse and expand flow of the preprocessor itself (e.g. `#ifdef __WINDOWS #define _USE_LOAD_LIBRARY #endif`).

The C Grammar is Context-Sensitive

The second problem is, that the C grammar is not a simple context-free grammar. Any lexical analyzer that is to analyze C is required to receive feedback information from the parser, as it progresses through the source code.

The problem is caused by the fact, that the lexical analyzer has to deal with ambiguous token descriptions: Both ordinary identifiers, and type names are defined by exactly the same token description.

We shall briefly explain how every C lexical analyzer has to solve this problem. First recall that, using the *typedef* facility from section 6.4.8, it is possible to introduce new type names in C. Now, when the tokenization process starts, the lexical analyzer classifies, by default, any identifier, or type name, that it encounters as just an identifier. Classified tokens are supplied to the C parser which starts parsing. Now, whenever the parser encounters the *typedef* facility, it identifies the new type name. The type name is then sent back to the lexical analyzer in a feedback construction. At this point, everytime an identifier is classified that matches this particular type name, it is classified as a *type name* instead.

To *AutoDLX* this poses a major problem. Because in order to parse just a single (portion) of a C header file, to let the parser identify possible type names, *AutoDLX* needs to parse *all* included header

files. Otherwise, the lexical analyzer that is part of *AutoDLX* does *not know* which tokens to classify as identifiers and which tokens to classify as type names.

Combine this difficulty with the *platform-dependent conditionals* from the previous section and we have just created an even more complicated problem.

Renaming strategies

The last problems that *AutoDLX* needs to deal with, are difficulties caused by automated renaming strategies. Recall that, since in Ruby a class name is constant, it must always start with a capital. C, on the other hand, is more relaxed in its definition for type names (e.g. of structures and unions). Since in *DLX* the type names of such structures or unions are mapped onto equivalent class names, this requires a sensible renaming strategy.

To illustrate this type of problem, please consider the following example:

C

```c
struct _GtkObject
{
    ...
};

typedef struct _GtkObject GtkObject;
```

DLX

```
class GtkObject < struct "GtkObject",
[
   ...
]
end
typealias( "GtkObject", "struct GtkObject" )
```

Figure 6.21: Choosing sensible type names.

To the left we see the type definition of a "`struct _GtkObject`", which is later `typedef`*ined* to a "`GtkObject`".

Any human immediately sees that probably the best name for the Ruby class would be "`GtkObject`" rather than "`_GtkObject`". For a computer algorithm, on the other hand, to come up for the best possible new name, this is much harder.

6.8.2 Memory Management

While it is not the primary focus of the research for this thesis, memory management, and the related problems caused by it, are known, hard to solve, issues, in any multi-language interoperability interface.

We have already explained that most memory management related problems arise because of object sharing (see section 4.7.3).

We see several approaches that look promising for handling memory management in *DLX*. To explain these approaches, first recall from sections 5.2 (where we explained our choice for selecting Ruby and C)

and 6.4.2 (where we explain Ruby types from an implementation detail perspective) that Ruby is written in C, and that Ruby objects (apart from the simple direct values) are implemented as a thin layer around the actual C objects.

In Ruby, all memory management is done automatically by a mark+sweep garbage collector[68]. This garbage collector is also made available to the Ruby Extension API, the interface that we introduced in section 5.5.2, and the interface on top of which *DLX* is built.

With the mark+sweep garbage collector interface available to *DLX*, the question arises: *"Could we use it as a basis to solve memory management issues that we listed in section 4.7.3?"*

We see two possible approaches to use the mark+sweep algorithm to track and manage C objects that have crossed the bridge. We can either do this *explicitly*, at C level, as part of the *DLX* extension library (i.e. the `dlx.so`); or we can do it *implicitly*, by passively waiting for Ruby to reclaim Ruby objects and use this as a signal to also clean up the C objects.

Explicit Approach

Since *DLX* uses a Ruby object to wrap, a pointer to a C object[32], whenever that C object is no longer either *directly* (e.g. by means of the wrapping Ruby object) or *indirectly* (e.g. as a structure field or as an element of a C array) accessible from Ruby, then it is likely that the object can be reclaimed. We can use this information as input to the mark+sweep functions directly via the Ruby extension API interface.

Implicit Approach

On the other hand, Ruby also has a notion of finalization. That is, whenever a Ruby object is garbage collected by the mark+sweep algorithm, a special function is invoked that allows the programmer to perform some clean up actions. In theory, whenever a *DLX* wrapper object is finalized, this should be a good idea to reclaim the memory of the C object that it wraps.

We have done some preliminary experiments with these ideas, which we discuss in the next chapter in section 7.6.

6.8.2.1 Discussion

The results seem promising, but it is too dangerous to draw conclusions from such a preliminary experiment.

For instance, what happens if some API retains a reference to an object of which we think is no longer accessible from within Ruby?

Although, it is likely that this is documented as part of the API's semantics, in which case we might be

[32] We ignore direct values here.

able to express such semantics by introducing additional *DLX* specification syntax, it remains to be seen if this is adequate.

In the end, we think that a combination of both approaches (explicit use of mark+sweep and implicit using Ruby finalization) would give enough control to effectively address the memory management problem in the future.

For now, it is clear that there is much to be researched in this department, so we will add it to our list of future work.

6.8.3 Threading

Also not a primary focus of this thesis, but deemed important enough to be mentioned here, is the topic of *threading*.

In *DLX* we distinguish three distinct threading strategies.

1. Ruby provides the threading support.
2. The C library provides the threading support.
3. Both Ruby and a C library provide threading support.

We have only experimented with the first two approaches (for instance, the *Test Buggy experiment* and the experiment with the *Gtk Tic Tac Toe widget* rely on threading provided by the C library. The third, and last, approach is a really hard problem that will need extensive research by itself.

This research must then seek answers to find effective approaches to synchronize such threads and how to do thread locking with multiple threading strategies in use at the same time.

For now, we can only give out the following set of advices:

- Avoid using two threading strategies (in multiple languages) at one single time.
- If you must use threading, and the C library does not require it, prefer Ruby's threading support over any special C threading library. This has the following benefits:
  - Ruby thread support is lightweight and portable.
  - All Ruby threads can access any wrapped C object.
  - C objects are already wrapped inside Ruby objects, so this can be used to facilitate thread locking, even on C objects.
- If the C library requires threading, by all means, use it (it appears to be working), but then also rely on the mechanisms that it provides for thread locking. Do not try to do self-implemented thread locking with pure Ruby objects (this is also in accordance with our first point).

Chapter 7

Experiments

In this chapter, we describe and discuss the various experiments that we have conducted to verify and validate our solution, *DLX*. The experiments are described in roughly the order of importance to us at this stage. That is, *correctness* is more important than *effectiveness* which is more important than *performance* and so on.

7.1 Correctness Experiments: Lab Tests

In this section, we describe two correctness experiments. These experiments can be considered: Experiments, conducted under "laboratory" conditions. They are conducted with libraries that have been especially created[1] for testing purposes only.

7.1.1 Memory Alignment Correctness

One of the most challenging parts of *DLX* is the memory layout reconstruction, of which the algorithm is described in section 6.5. It is crucial to the successful operation of *DLX* that it is performed correctly. Because results may vary due to compiler and platform differences, we have constructed an *Alignment Test Generator* that performs a statistical test for memory alignment correctness.

Alignment Test Generator

The *Alignment Test Generator* generates an arbitrary amount of memory alignment tests in the form of generated C source code. The generated source code is then compiled against the *DLX* memory layout reconstruction back end. A test is only successful if and only if the test yields no errors during dynamic field access to any of the generated test structures.

[1] They have been either crafted by hand, or generated as part of the testing process.

A *memory alignment test* can be divided into five parts.

First, a static test structure is generated. This structure is to be processed and compiled by the C compiler for the target platform.

```
1  struct Test_struct0
2  {
3     float        f0     ;
4     unsigned int bf1 :  6;
5     long         l2     ;
6     unsigned int bf3 :  6;
7     unsigned int bf4 :  7;
8  };
```

Then, a dynamic representation of the structure is generated that is used by the memory layout reconstruction algorithm.

```
1  void* struct_ary[] = { (void*)20,
2     (void*)TYPE_FLOAT   , "float"       , "f0" , NULL    ,
3     (void*)TYPE_BITFIELD, "unsigned int", "bf1", (void*)6,
4     (void*)TYPE_LONG    , "long"        , "l2" , NULL    ,
5     (void*)TYPE_BITFIELD, "unsigned int", "bf3", (void*)6,
6     (void*)TYPE_BITFIELD, "unsigned int", "bf4", (void*)7
7  };
```

This is followed by an instantiation of the `StructInfo` object that holds information of the dynamic representation. A short verification is performed that tests the *size* of the *static* struct (as computed by the C compiler) against the size of the struct as calculated by our algorithm from the `StructInfo` object.

```
1  StructInfo structinfo = struct_new( "MyDummyModule",
2                                      "Test_struct0",
3                                      struct_ary+1 );
4  if( structinfo->size != sizeof(struct Test_struct0) )
5  {
6     res += 1;
7     fprintf( stderr, "size mismatch for struct Test_struct0:\n\
8         calculated size:  %d != compiled size:  %d\n",
9              structinfo->size, sizeof(struct Test_struct0) );
10 }
```

Then, an instance of the test structure is filled statically with appropriate random values.

```
1  test_struct0.f0  = 428.238346297448;
2  test_struct0.bf1 = 7;
3  test_struct0.l2  = 427982407;
4  test_struct0.bf3 = 22;
5  test_struct0.bf4 = 83;
```

Finally, a field access test is performed for each member of the test struct. Errors are reported and accumulated for each field access.

```
1  tmp = struct_member_get( &test_struct0, structinfo, "f0" );
2  if( test_struct0.f0 != *((float*)tmp) )
3  {
4     res += 1;
5     fprintf( stderr,
6         "test_struct0.f0 returned incorrect value:  %f != %f\n",
7              test_struct0.f0, *((float*)tmp) );
8  }
```

Setup and Results

In a typical test, we use the *Alignment Test Generator* to generate 10,000 structures, varying in size and composition. As stated before, the test is successful if and only if there is not a single error.

The results are given as part of the *portability experiments* (see section 7.2) in the next section, where we conduct several alignment correctness tests for various platforms.

A Note on Statistical Evaluation

Please note, that since this test uses a statistical approach, rather than an analytical approach, all assumptions related to statistical testing apply. That is, the test can really only prove incorrectness of the memory layout reconstruction for a given target compiler and platform; qualifications of correctness can only be made up to a certain, though arbitrary, degree, by increasing the number of alignment tests.

7.1.2 Unit Test Suite

In addition to the automatically generated memory alignment tests, we also conducted a series of hand written unit tests to test each feature of *DLX* separately. These unit tests are written in Ruby (using Ruby's unit testing facilities), and make use of a synthetically created shared library, called *libcomplex*. (Please note that it has nothing to do with *complex numbers*, rather the name is chosen to contrast the examples with *simple* types that usually accompany multi-language development solutions.) The unit tests are automatically conducted as part of the build process, so that it is always clear if Ruby/*DLX* is working on the target platform.

It is not possible to go into details of each of the unit tests. We can only give a few examples. To make the examples a little bit more comprehensible, we will first give an excerpt of the *DLX* specification for *libcomplex*. For illustration purposes, the actual specification has been greatly simplified to cut down in required space.

```
1  class ComplexSubSub < struct "ComplexSubSub",
2  [
3    "int",                                          :id,
4  ]
5  end
6  typealias( "ComplexSubSub", "struct ComplexSubSub" );
7
8  class ComplexSub < struct "ComplexSub",
9  [
10   "ComplexSubSub**",                              :list,
11   "char*  (*simple_callback)(char* s)",           :simple_callback,
12   "char* (*complex_callback)(ComplexSubSub* css)", :complex_callback,
13   "int  (*complex_callback2)(ComplexSubSub* css)", :complex_callback2,
14 ]
15 end
16 typealias( "ComplexSub", "struct ComplexSub" );
17
18 class Complex < struct "Complex",
19 [
20   "struct ComplexSub*",                           :complexsub,
21 ]
22 end
23 typealias( "Complex", "struct Complex" );
24
25 extern "Complex* newComplex()", delay;
```

Figure 7.1: An excerpt from the *DLX* specification of `libcomplex`.

All unit tests are conducted individually, and are used to test individual features of *DLX* for a specific known (expected) outcome. In the example below, two unit tests are shown that test two of the four callback flows that were identified in figure 6.7 of section 6.3.6.3. (*En passant*, we also see the automatic array wrapping, from section 6.4.5, in action in lines 6, 8, 22, and 24.)

As with the memory alignment correctness experiment, the unit test suite passes if and only if all of the individual unit tests succeed.

7.1.2.1 A Note on Unit Test Evaluation

Unit tests can only be used to prove incorrectness of a particular portion of the implementation. They *can* be used to test for functional regression, but they can *never* prove correctness; if some feature of *DLX* is not properly tested (because of a faulty or missing test), the use of any unit test suite will not expose this.

```ruby
# test default callbacks (original callbacks residing in C)
def test_default_callbacks
  complexsub = ComplexLib.newComplex.complexsub;
  expect( complexsub.simple_callback( "DLX" ),
          "DLX works great in C" );
  expect( complexsub.complex_callback( complexsub.list[7] ),
          "ComplexSubSub's id in C: 7" );
  expect( complexsub.complex_callback2( complexsub.list[7] ), 7 );
end

# test changing + calling callbacks
def test_nondefault_callbacks
  complexsub = ComplexLib.newComplex.complexsub;

  complexsub.simple_callback   = self, :simpleCallback;
  complexsub.complex_callback  = self, :complexCallback;
  complexsub.complex_callback2 = self, :complexCallback2;

  expect( complexsub.simple_callback( "DLX" ),
          "DLX works great in Ruby" );

  expect( complexsub.complex_callback( complexsub.list[7] ),
          "ComplexSubSub's id in Ruby:  7" );
  expect( complexsub.complex_callback2( complexsub.list[7] ), 7+1 );
  # the callback performs the +1 above
end

def simpleCallback( s )
  return( "#{s}works great in Ruby" );
end

def complexCallback( complexsubsub )
  return( "ComplexSubSub's id in Ruby:  #{complexsubsub.id}" );
end

def complexCallback2( complexsubsub )
  return( complexsubsub.id+1 );
end
```

Figure 7.2: Several callback unit tests. A selection of the many unit tests that have been created for *DLX*.

7.2 Portability Experiments

To test the portability of our solution, we have conducted two tests. First, we have conducted the experiments from the previous section on a variety of target platforms (i.e. the triplet of compiler, architecture, and operating system). Then we took our solution into the open field, and tested the portability of our solution using a real world example: A modest music playing application, called *DLXPlayer*, is shown running successfully on two very diverse platforms.

7.2.1 Platform Portability and Compatibility

Although there have been third-party reports stating that the current implementation of Ruby/*DLX* is working on a variety of Unix and POSIX operating systems (BSD, HPUX, etc.) and for several additional architectures (PowerPC and IA64), these could not be confirmed by us, due to lack of access to the required hardware and operating systems. We *have* been able to confirm that *DLX* runs all of the correctness experiments from the previous section successfully on the following platforms:

Compiler	Operating System	Architecture	Cross compiler
GCC[2] 3.2	GNU/Linux 2.6.5	IA32/x86	no
GCC 3.4	GNU/Linux 2.6.12	IA32/x86	no
GCC 4.1.1	GNU/Linux 2.6.19	IA32/x86	no
MSVC[3] 12.20.9615	Windows CE 4.2	ARM	yes
TCC[4] version 0.9.23	GNU/Linux 2.6.19	IA32/x86	no

Table 7.2: The platforms on which *DLX* has been confirmed to be working.

7.2.2 Portability: A Field Test

To test if *DLX* actually successfully provides the separation of concerns regarding portability in the real world, we have conducted a small field test.

For this, we took two fairly distinct platforms to run our experiment:

1. A GNU/Linux 2.6 Desktop environment running on an IA32 architecture.
2. A PDA based around an ARM processor running Windows CE 2003SE.

As an experiment, we have written a small graphical application that loads and plays music files. The application makes use of:

- *Ruby* - As the main execution platform. (The small application is written in Ruby.)
- *Simple DirectMedia Layer* (SDL) - A shared library used for the (graphical) user interface.
- *MikMod* - A shared library that can be used to play music modules.
- *DLX* - To establish the interface between the main application written in Ruby on one side, and SDL and MikMod as shared libraries on the other.

All four basic components where available for both target platforms. The shared libraries on Windows CE are called *SDL.dll* and *mikmod.dll*, while on GNU/Linux they are called *libSDL.so* and *libmikmod.so*.

Apart from the small display resolution of the PDA (320x240), we constructed the test application without any prior assumptions regarding target platform; i.e. there is no special code added to test for, or to take into account, the current execution platform. Still, using *DLX*, the same Ruby program, runs unaltered and exactly the same on both platforms.

We have included two photographs as an illustration, they can be seen in figure 7.3.

Figure 7.3: DLXPlayer in action. Left: Photo of an Ipaq 2210 running DLXPlayer. Right: A screenshot of a Linux desktop running the same program.

7.3 Experiments on Effectiveness: Field Tests

To truly test the effectiveness of *DLX*, the laboratory tests are not sufficient. In requirement 6 we put that our solution must be able to handle any API semantics. This means, that no matter in what way a shared library's API was intended to be used, *DLX* must be able to interface it to Ruby.

Previous experiences have learned that quite a few language interface solutions fail sooner or later on real world examples[5]; in C some pretty onorthodox APIs are possible, that make perfect sense afterwards.

What we would like to have is a series of tests to demonstrate the effectiveness of a multi-language interface between C and one or more other languages. Unfortunately we could not find an existing set of such tests. Therefore, we have selected two examples of shared libraries which, we know from previous experience, have an onorthodox construction in their APIs, or form an otherwise very complete effectivity test. We have used these examples to do two small case studies.

[5]Or they require extensive programming in addition to, or instead of, a specification.

7.3.1 Effectivity Experiment I: "ODE TestBuggy"

As the first experiment on the effectivity of *DLX*, we have selected the "*ODE TestBuggy*" application. This is an example application that comes as part of the ***O**pen **D**ynamics **E**ngine* (ODE) distribution, a physics simulation engine that comes as a set of shared libraries.

The example application, where a tri-wheeled vehicle –the *Test Buggy*, if you will– is moved in a virtual world, was selected for several reasons. First of all, the application uses a fair amount of all the language features that C has to offer, and it uses many of the features that *DLX* is equipped with.

Furthermore, the API contains a special construction in C. We would like to highlight this, to show how complete our mapping is, and to show how our mapping manages to keep the use of this API relatively simple, given the required API semantics.

We think, that this example also shows what we aimed for, when we suggested that computational intensive tasks be done in C, while doing the rest in Ruby[6].

ODE

To understand what actually is happening, we must first very briefly introduce ODE, and give a brief crash course in its terminology.

The Open Dynamics Engine, or ODE, is a physics simulation engine that can be used to simulate real world dynamics (e.g. gravity, motion, etc.). It consists of three parts,

1. a rigid body simulator;
2. a collision detection system; and
3. a simple 3D rendering engine (with OpenGL as back end).

To better understand the logic behind the control flow that we are about to discuss, please skim through the following terminology.

Rigid body - In physics, a *rigid body* is an idealization of a solid body, of finite size, in which deformation is neglected[11]. In ODE, a rigid body is simulated as a point mass that can be given various attributes, such as, mass, direction, rotation, and speed.

Joint - A rigid body is connected to another rigid body by a joint. A joint is used to constrain the relative freedom of movement between two or more rigid bodies. It is also used to conduct applied forces from one rigid body to another. Joints of various types exist in ODE, such as, a hinge, a ball in socket, and a rotation joint.

[6]The careful reader may notice that, since this is only a small demo application, e.g. without game logic and without much else, the 80/20 conjecture does not apply well here.

Geometric object - Since a rigid body in ODE is represented by a point mass, it has no volume or form associated with it. One or more *geometric objects* are associated with a rigid body, giving it a volume and a form. These objects are then used to detect collisions between two or more objects. Examples of geometric objects are a cube, a ball, a piramid, and a polygon.

Collision space - Collision detection is a painfully computational intensive task. To limit the number of geometric objects that can cause a collision, these objects are added to a collision space. Objects that are in the same collision space will not be tested for impending collisions.

Contact and contact joint - If the collision detector detects that a collision is impending, it creates a *contact* point that describes the location where two geometric objects are colliding. At this location, a special temporary joint can be created (a *contact joint*) between the rigid bodies that are associated with the geometric objects. This way, the collision is reflected back to the rigid body simulation.

Test Buggy Control Flow

While having only a single explicit function invocation direction may seem limited. We will show next that, by using callbacks, there is a very dynamic control flow possible between Ruby and C. *DLX* allows shared libraries to make maximal use of this.

To illustrate this tighly orchestrated *tango* between Ruby, *DLX*, and C, please consider the schematic overview of the Test Buggy demo in figure 7.4. Indicated with a number inside a white circle are the 7 major steps that are taken in the demo. We will go through each of these below[7].

1. When the application is started, it sets up the Test Buggy by creating all required rigid bodies, connecting them via the right joints, associating each body with a geometric object, which, in turn, is added to a collision space.

2. Then, a *Render Info* object is passed to the 3D render engine (and execution continues in C). This Render Info object is a C object that is created and allocated in Ruby. From a Ruby point of view, just a `.new` function is invoked, so allocation details are hidden from the end user (see section 6.4.5). The Render Info object contains:

   - A version field, that is initialized in Ruby with the version of the header file that was used to base the *DLX* specification on.
   - A set of callbacks, that will be invoked as part of the render loop (some of the callbacks point to Ruby methods and some of them point to C functions).
   - Data paths to graphics and the likes.

   Internal to the shared C library, the 3D Render Engine then:

   (a) Verifies that the API version that the main application expects to use is compatible with the shared library (see section 6.6.2.2).

   (b) Sets up the render world and enters the main render loop.

[7] The number in the list corresponds to the number in the schematic overview of figure 7.4.

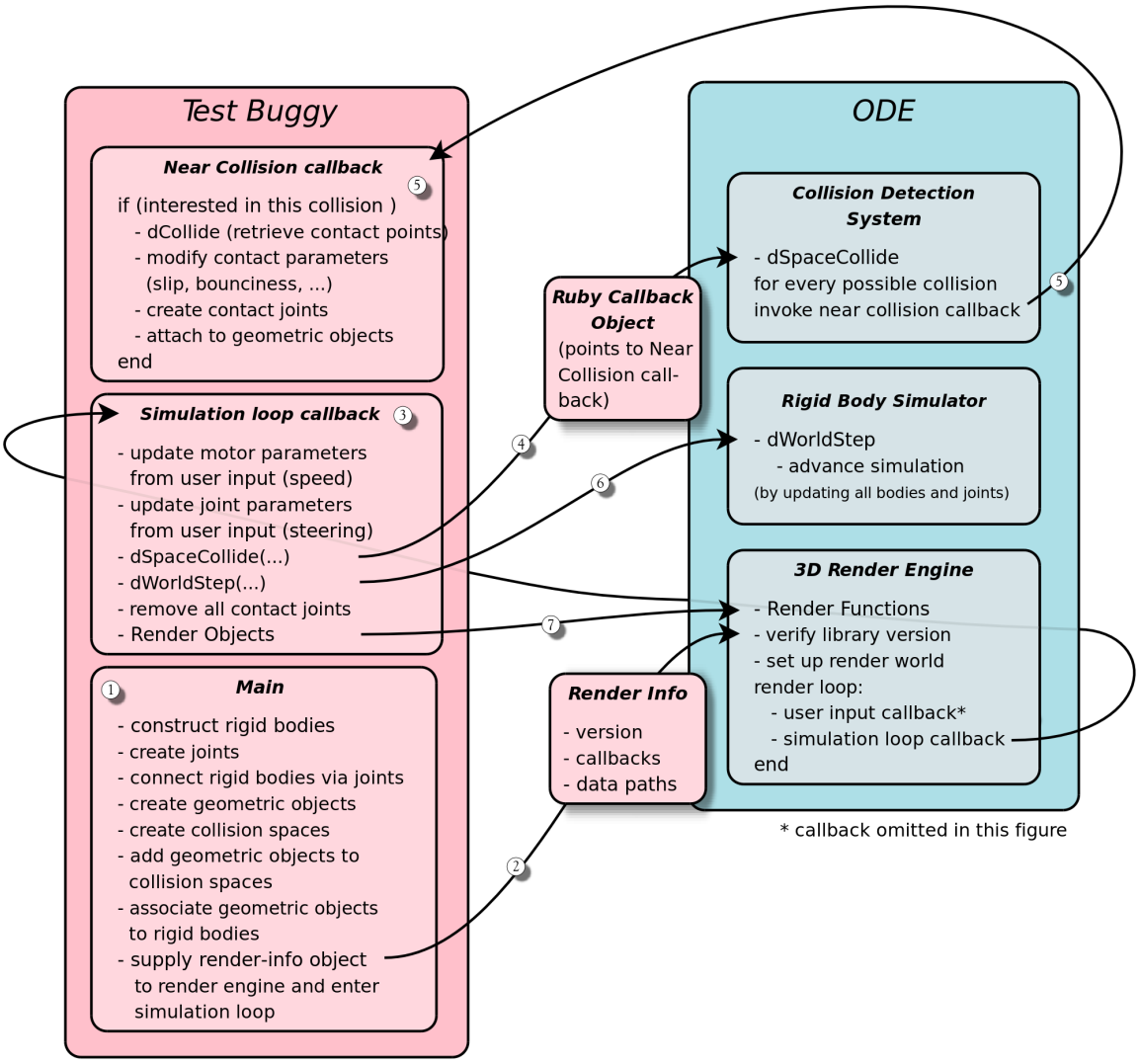

Figure 7.4: The TestBuggy's control flow between Ruby/*DLX* and C.

(c) For each iteration of the render loop it invokes several callbacks (which are taken from the Render Info object). One of the callbacks is the simulation loop callback. This callback is defined, by us, in Ruby (so execution control is given back to Ruby code).

3. Inside the simulation loop callback, user input is used to update motor and steering parameters.

4. Then `dSpaceCollide` is called to let ODE's collision detection system calculate possible collisions. By this invocation, within the same function call stack, execution control is again passed back to C.
   As a part of the call to `dSpaceCollide`, a standalone Ruby callback object is passed that points the pure Ruby *Near Collision callback*.

5. The collision detection system calculates all possible collisions, and for each of these, it invokes the pure Ruby *Near Collision callback*, passing –yet again– execution control back to Ruby.

Inside the Near Collision callback a quick test is made to see if the collision is interesting enough, if so it:

(a) Retrieves the contact points using a call to `dCollide` (more on this later).

(b) Modifies the contact parameters, such as, slipperiness and bounciness. (Field access of these structs is, of course, fully encapsulated as was described in the previous chapter in section 6.4.5. To illustrate this, see lines 14-20 in figure 7.5.)

(c) Creates contact joints, and attaches these to the geometric objects (which are, in turn, associated with rigid bodies).

6. Once the rigid bodies are temporarily connected via the contact joints, a call to `dWorldStep` is used to advance the simulation forward. This will update the location and orientation of the rigid bodies (and their associated geometric objects).

7. Finally, using the 3D render engine's functions, the geometric objects, that have been associated with the rigid bodies, are rendered[8].

We think that the dynamic control flow, that switches from Ruby to C flawlessly for several times, clearly demonstrates, that there is more to interfacing Ruby to a shared library than people at first expect. Eventhough we only allow explicit invocation of C functions from inside Ruby, this *does not mean* that the interface is simply unidirectional.

This also demonstrates, that the above shared C API is not just a simple API, that just contains a collection of functions "flat" functions. There must be a really tight interfacing for the Test Buggy application to work.

To elaborate on this a little further, we wish to highlight a small detail from the API, in which we think the limitations of "seamlessness" come into view, and how *DLX* handles it.

Collision Callback API Detail

The detail that we wish to highlight concerns the collision callback API of ODE, in particular, the `dCollide` function from step 5a. It is not so much the function call itself that is peculiar, but what it expects to be passed as parameters. Essentially, the function expects to receive an array of elements of type `struct Contact` (for a description of this structure, see figure 7.6).

A `struct Contact` has two embedded structures. Please note, these are not *pointers to* structures, but *nested* structures.

As a part of API semantics of `dCollide`, a C programmer must:

1. Supply an allocated block of memory capable of holding a certain amount of "`struct Contact`" elements. Again, please note that we say *structures*, not *pointers to structures*.

[8] It is common to use a 3D model with a higher polygon count for this step.

```
1  N = 10;  # handle at most 10 contact points per collision
2  @contact_ary = ContactArray.new!( N );
3
4  def nearCollisionCallback( data, geom1, geom2 )
5    contact = @contact_ary;
6    n = ODE.dCollide( geom1, geom2, N, contact[0].geom, Contact.size );
7    n.times do |i|
8      contact[i].surface.mode = ContactSlip1 | ContactSoftERP | ...  # etc.
9      contact[i].surface.mu = Infinity;
10     contact[i].surface.slip1 = 0.1;
11     contact[i].surface.slip2 = 0.1;
12     contact[i].surface.soft_erp = 1.5;
13     contact[i].surface.soft_cfm = 0.1;
14
15     contactjoint = ODE.dJointCreateContact( world, contactgroup, contact[i] );
16     ODE.dJointAttach( contactjoint, ODE.dGeomGetBody( contact[i].geom.g1 ),
17                                     ODE.dGeomGetBody( contact[i].geom.g2 ) );
18   end
19 end
```

Figure 7.5: Near Collision callback.

2. Supply the number of elements of type "`struct Contact`" that the block of memory can hold.
3. Supply the *memory address* of (i.e. a pointer to) the first "`struct ContactGeom`"
4. Supply the size of "`struct Contact`"

```
1  class Contact < struct "Contact",
2  [
3    "struct SurfaceParameters",  :surface,
4    "struct ContactGeom",        :geom,
5    "Vector3",                   :fdir1
6  ]
7  end
8  typealias( "Contact", "struct Contact" );
```

Figure 7.6: *DLX* type specification for Contact.

To translate this into Ruby/*DLX* semantics, we are faced with two problems:

In the C source code, this results in the use of a mixture of `.` and `->` operators. So there is challenge as to how this must be handled.

Furthermore, we *must* be able to distinguish arrays that are merely pointers to structures and real structures (like in this case).

Because in *DLX*, we hide all notions of pointers, this is a problem. Therefore, we introduced new terminology: *Contiguous* versus *non-contiguous* arrays (see section 6.4.7).

With the above knowledge, we can translate the above C API semantics into Ruby/*DLX* API semantics.

In Ruby/*DLX* the programmer must:

1. Supply a *contiguous* array of "`Contact`" elements (e.g. using the "`new!`" method instead of the more common "`new`"[9]).
2. Supply the size of the above array (either as a separate variable or as call to its `.size` instance method).
3. Simply reference the "`Contact.geom`" field. When it is supplied to a function that expects a pointer, it is automatically supplied as one.
4. Supply the "`Contact`" size (with a call to its `.size` class method).

As we have described in the previous chapter, in *DLX*: We hide the notion of pointers, and we automatically switch between the interpretation of a structure (i.e. either a *pointer to* a structure or a real structure, e.g. one *nested* inside another), depending on the context. Therefore, in *DLX*, we can always use the `.` operator, and there is no mixture of `''.''` and `''->''` operators needed. This is demonstrated in figure 7.5.

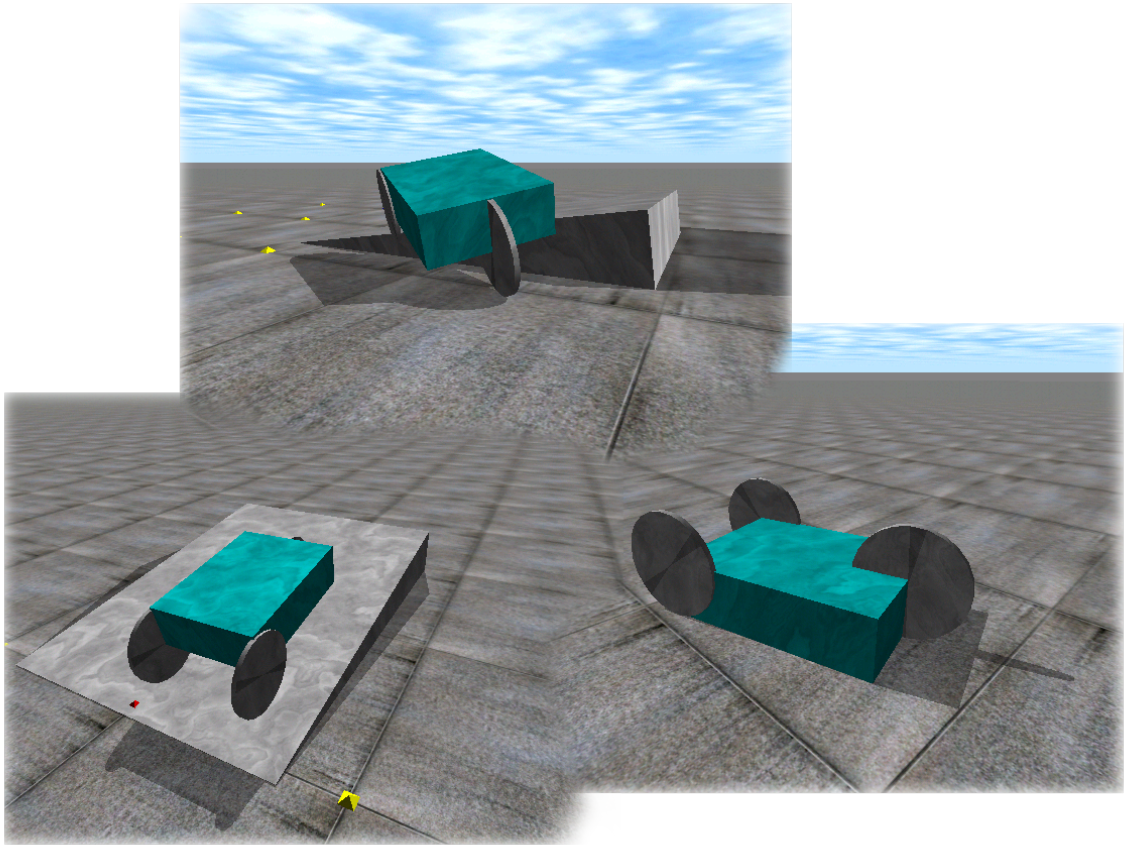

Figure 7.7: Screenshots from the ODE TestBuggy demo application.

[9]Please notice the exclamation mark in the first method name.

Final Note and Screenshots

In *DLX*, one of our goals is to supply an easy to create and maintain multi-language interface, by means of a simple specification. At the same time we want to shield end users from explicit low-level details, such as, pointers and pointer arithmetic (which are normally kept transparent to Ruby programmers).

Experiments like the Test Buggy help us to ensure the effectivity of such a multi-language interface, with even the more onorthodox API semantics. In these cases, the boundaries of what we can achieve by specification, and what we can hide from the user for the sake of seamlessness are really stretched to the extremes.

Finally, let us conclude this section with a little eye candy. A few screenshots of the running demo can be seen in figure 7.7.

7.3.2 Effectivity Experiment II: "Inheritance in C"

We think that the previous experiment already shows a great deal of the possibilities of *DLX*, however, a few features of *DLX* are *not* tested by the TestBuggy experiment[10]. These shall be tested as part of our second experiment: "Inheritance in C".

Although C is not an object-oriented language itself, by creatively using Structure Types, Union Types and function pointers, features that are traditionally attributed to object-oriented languages, such as, *inheritance*, *polymorphism*, and *reference counting* can be emulated in C.

One example of using an object-oriented approach in C is the *GTK+ Object System*[5]. It is the object-oriented system that is used as the basis for GTK+, an opensource windowing toolkit for creating graphical user interfaces.

In this experiment, we are going to create a "subclass" in the notion of the GTK+ Object System. For this, we are going to use nothing but pure Ruby and *DLX*. Normally, a subclass in the GTK+ Object System must be written in C and compiled to native code before it can be used. We think that we have a strong point, if we can *convince* GTK+, that our subclass, which will be in the form of a custom widget, is a legitimate GTK+ widget.

We are not only strengthened in our believes that we successfully achieved this if only the demonstration program appears to be working correctly; GTK+ also performs various internal consistency checks to test for errors[11] when we introduce our subclass to the GTK+ Object System.

We must stress, that we do *not* wish to advocate the use of C in this fashion and subsequently interfacing it to Ruby with *DLX*. Ruby is a pure object-oriented language and, as such, it does a better job than C in this regard. By this experiment we simply want to show the effectiveness of *DLX* when interfacing Ruby to this, or any other, shared C library.

[10] The most notable of these are: Deep nesting of structures; and bitfields.
[11] We categorized these as *external errors* in section 6.6.2.

GTK+ Object System

Inheritance in the GTK+ Object System is achieved by nesting structures within each other. So if, for instance, the `GtkButton` class inherits from `GtkWidget` (which it does), then a `GtkButton` structure must start with a nested `GtkWidget` structure. This ensures that a pointer to a `GtkButton` can be cast into a pointer to a `GtkWidget`.

There are two types of structures that are nested in this way:

1. *Instance structures*
2. *Class structures*

The structures mentioned above are called *instance structures*, as they represent instantiated objects.

Each instance is also associated with a *class structure*. This is essentially a table of pointers to functions that are the class's implementation. The function pointers in this table can be reassigned by subclasses that override particular functions. This essentially allows for polymorphism. (For example, the `GtkWidgetClass` structure includes a pointer to a `draw()` function; `GtkButtonClass` provides a specific implementation that draws buttons.)

Setup

The input for this widget is taken from the first example from *Chapter 21. Writing Your Own Widgets*[64], which is part of the GTK+ 2.0 tutorial[66]. In this experiment we left much of the original source code in tact (apart from translating it into Ruby and *DLX*). We have not bothered with giving it a more Ruby-like appearance.

The Widget

In this experiment, we shall try to create a subclass of the GTK+ widget `GtkVBox`. The inheritance path of `GtkVBox` is given below.

```
GObject
 +—GInitiallyUnowned
   +—GtkObject
     +—GtkWidget
       +—GtkContainer
         +—GtkBox
           +—GtkVBox
```

The widget is to render nine on/off buttons in three rows with three buttons each. The widget will be called `Tictactoe`, and in fact, it can be seen as a poor man's solitaire tic tac toe: Whenever three buttons in the same row, column, or in a diagonal are switched on, the widget sends out a "win" signal.

In order to create a subclass of `GtkVBox`, we must create two structures, as we explained earlier: A `Tictactoe` structure, the *instance structure* and the associated *class structure*, `TictactoeClass` (see figure 7.8). The structures are defined in Ruby and *DLX*, but do understand that, unlike the previous experiments we have conducted, these two structures do *not* represent a counterpart inside a compiled C object (i.e. a shared library); they exist purely in run-time, during the execution of the Ruby program.

```
 1  typealias( "Tictactoe*", "void*" );
 2  class TictactoeClass < struct "TictactoeClass",
 3  [
 4    "GtkVBoxClass",                       :parent_class,
 5    "void (*tictactoe) (Tictactoe* ttt)", :tictactoe,
 6  ]
 7  end
 8  typealias( "TictactoeClass", "struct TictactoeClass" );
 9
10  class Tictactoe < struct "Tictactoe",
11  [
12    "GtkVBox",                            :vbox,
13    "GtkToggleButton*[9]",                :buttons,
14  ]
15   # ...  rest of the definition ...
16  end
17  typealias( "Tictactoe", "struct Tictactoe" );
```

Figure 7.8: The two structures of the `Tictactoe` widget: A *class structure* (top) and an *instance structure* (bottom).

Clearly visible in the source code fragment is that `GtkVBox` is extended by nesting the `GtkVBox` structure (and associated class structure) as the first element of the `Tictactoe` structure (lines 12 and 4 respectively). Like we stated previously, this ensures proper inheritance in the GTK+ Object System.

Also visible in the instance structure is a `GtkToggleButton` array. This array is will hold the nine on/off (toggle) buttons that represent the tic tac toe grid.

After we have created the two structures that represent our subclass, it needs be to registered with the GTK+ Object System. To do this, we create a `GTypeInfo` object and initialize it with various important values, for example: The size of either structure, a class initialization function, and an instance initialization function[12] (see figure 7.9). Once the `GTypeInfo` object is created and initialized with the proper values, it can be used to register the new widget with the GTK+ Object System with a call to `g_type_register_*` (line 13 of figure 7.9).

In the source code fragment, two initialization functions are referenced. We do not go into detail of either of these functions. Suffice it to say, that the class initialization function is used to add the notion of a "win" signal to the widget class, that can be triggered if a win condition is encountered.

Each time an `Tictactoe` widget is instantiated, the instance initialization function –the constructor, if you will– is used to create nine toggle buttons and add them to the `GtkToggleButton` array that we mentioned earlier. The buttons are laid out in a grid pattern. Furthermore, each of the button's "toggle"

[12] In many object-oriented languages this is called the *constructor* function.

```
ttt_info = GTypeInfo.new <=
[
  TictactoeClass.size,
  nil,
  nil,
  GClassInitFunc.new( self, :tictactoe_class_init ),
  nil,
  nil,
  Tictactoe.size,
  0,
  GInstanceInitFunc.new( self, :tictactoe_init ),
];
Gtk.g_type_register_static( Gtk.GTK_TYPE_VBOX,
                            "Tictactoe", ttt_info, 0 );
```

Figure 7.9: Registering the new widget with the GTK+ Object System.

signal is connected via a callback to a Ruby method that determines if a win condition is encountered, and, if so, triggers the widget's "win" signal.

Note: In the API of GTK+, bitfields play a large role, however, they are not very visible in the above experiment. However, with faulty bitfield support, the experiment would have certainly failed for two reasons:

1. Testing to see if a toggle button is pressed is done via a bitfield.
2. Many of the structures in GTK+ use bitfields. If either the size or alignment of these bitfields would have been improperly reconstructed then, because of the many nested structures in GTK+, errors would have easily propagated to the point that the experiment would have failed miserably.

Result

The outcome of the experiment, is as we expected: When the Ruby program is started, it shows us the window containing our `Tictactoe` widget. This means that GTK+ has accepted our pure run-time subclass and used it to instantiate a new widget.

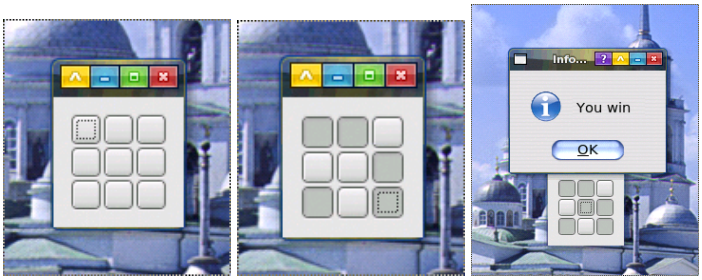

Figure 7.10: Screenshots of the working `Tictactoe` widget.

The displayed `Tictactoe` widget's buttons can be toggled on or off, and once three buttons in a single row or column or diagonally are toggled on, the win condition is triggered. As an illustration we have added a few screenshots (see figure 7.10).

7.4 Dynamic Boundaries

In this experiment we show how dynamic boundaries (*R4*) are handled in *DLX*. For this experiment, we reuse the *Tictactoe* example from the previous section. We are going to promote the `buttons` field from C to Ruby.

In *DLX*, this is very easy and straightforward. Because the *DLX* specification is part of an ordinary Ruby class specification, the dynamic boundary becomes visible quite literally at the bottom of the *DLX* struct (or union) specification, and right before the Ruby class body (lines 5 and 4 in figure 7.11). *Promoting* (or *demoting*) a struct field to a Ruby instance variable in *DLX* involves little more than a cut-and-paste action; cutting the appropriate line from the *DLX* specification and subsequently pasting it a few lines below (or vice versa for demoting). Just a small syntactical translation is required.

Before

```
1  class Tictactoe < struct
   "Tictactoe",
2  [
3     "GtkVBox", :vbox,
4     "GtkToggleButton*[9]", :buttons,
5  ]
6     # rest of the class ...
7  end
8
9
10
11
```

After

```
1  class Tictactoe < struct
   "Tictactoe",
2  [
3     "GtkVBox", :vbox,
4  ]
5
6     def buttons
7         @buttons ||= []
8     end
9
10    # rest of the class ...
11 end
```

Figure 7.11: Promoting a C structure field to a Ruby attribute.

At this point, the Tictactoe object's attribute has been promoted from being a C struct field to a Ruby instance variable. Because in Ruby/*DLX* an instance variable and a C struct field are both accessed via an ordinary Ruby method, *nothing* of the rest of the file has to be changed to make this promotion work (i.e. with the changes of figure 7.11 applied, we still have a working widget).

Of course, we assume that the promotion or demotion is performed for a reason, otherwise it would be useless to promote or demote an attribute.

A reason to *demote* an attribute from Ruby to C would be, if some portion of a computational intensive algorithm –which, of course, should be written in C– requires access to it. To the Ruby side, using Ruby/*DLX*, the demotion should yield little effect, since the attribute will still be accessible in the same way it was *prior* to the demotion.

```
def self.tictactoe_toggle( widget, ttt )

  ttt = Tictactoe.wrap( ttt );

  rwins = [ [ 0, 0, 0 ], [ 1, 1, 1 ], [ 2, 2, 2 ],
            [ 0, 1, 2 ], [ 0, 1, 2 ], [ 0, 1, 2 ],
            [ 0, 1, 2 ], [ 0, 1, 2 ] ];
  cwins = [ [ 0, 1, 2 ], [ 0, 1, 2 ], [ 0, 1, 2 ],
            [ 0, 0, 0 ], [ 1, 1, 1 ], [ 2, 2, 2 ],
            [ 0, 1, 2 ], [ 2, 1, 0 ] ];

  rwins.size.times { |k|
    success = true;
    found = false;

    rwins[0].size.times { |i|
      success = success && (ttt.buttons[rwins[k][i]*3+cwins[k][i]].active == 1);
      found = found || ttt.buttons[rwins[k][i]*3+cwins[k][i]] == widget;
    }

    if( success && found )
      Gtk.g_signal_emit( ttt, @@tictactoe_signals[TICTACTOE_SIGNAL], 0 );
      break;
    end
  }
end
```

Figure 7.12: Calculation of "*win*" conditions C-style.

A reason to *promote* an attribute from C to Ruby would be if, after some development time, code becomes more or less stable and there is no real need for an attribute to be accessible from C. It is likely that this opens the opportunity to refactor portions of the code to make it more readable or compact, as Ruby (in contrast to C) allows for this.

Our reasons for promoting the `buttons` attribute from C to Ruby fit exactly these criteria. There was no real reason the `buttons` attribute was accessible from C (hence, the whole point of the original Tictactoe example was to show that we could create an all Ruby subclassed `Widget`). And we wanted to simplify the code that is associated with the `buttons` attribute.

To illustrate this, we rewrote the original win condition code, which can be seen in figure 7.12 (which is based on the original C code, as we stated in section 7.3.2).

Apart from the fact that the C-like code looks bulky, one needs to pay real close attention to understand what is actually going on. At the moment of writing it may be clear to the original programmer, but the code is probably pretty hard to maintain by either the same programmer or someone else if, at some point, we change the rules for our Tictactoe game.

We replaced the code from figure 7.12 with following Ruby-nized implementation, which has exactly the same functionality (namely, trigger the appropriate signal if a win condition is encountered).

```
1  def self.tictactoe_toggle( widget, ttt )
2
3    ttt = Tictactoe.wrap( ttt );
4
5    if( ttt.buttons.collect{ |button| button.active }.join.match(
6     /^111......|...111...|......111|1..1..1..|.1..1..1.|..1..1..1|..1.1.1..|1...1...1$/ ) )
7          # h o r i z o n t a l          v e r t i c a l          d i a g o n a l
8      Gtk.g_signal_emit( ttt, @@tictactoe_signals[TICTACTOE_SIGNAL], 0 );
9    end
10 end
```

Figure 7.13: Calculation of "*win*" conditions Ruby-style.

This code is exactly what one wants it to be: compact, concise and clear.

7.5 Performance Experiments

7.5.1 Execution Speed Performance

At the current stage of development, execution speed performance is not a priority. However, to assess the current status, we have conducted execution speed performance tests.

There are many things of which we can test the execution speed. We have tested the execution performance of the most prominent of these: *Function calls*.

Setup

To test the execution performance of the function calls, we have created a source code generator that generates test functions. The test functions are called in a structured manner to reliably assess the performance.

Ideally, one would like to assess the performance in general, for any function prototype, but this is unfeasible: There are simply too many combinations possible. We therefore selected two sets of functions. Each set contains functions, that are according to a template function, with fixed parameter and return types. The contained functions increase in arity.

We selected the following two sets:

1. The "void" set, with prototypes:
   - `void void_func0();`
   - `void void_func1( int i1 );`
   - ...
   - `void void_funcn( int i1, ..., int in );`
2. The "dummy" set, with prototypes:
   - `PerformanceDummy* PerformanceDummy_func0()`
   - `PerformanceDummy* PerformanceDummy_func1( PerformanceDummy* p1 )`
   - ...
   - `PerformanceDummy* PerformanceDummy_funcn( PerformanceDummy* p1, ..., PerformanceDummy* pn )`.

The `PerformanceDummy` is a dummy structure that is used to represent a non-generic pointer. In *DLX*, the corresponding class must be searched for in a lookup table (part of the *DLX Type Database*), this incurs a small penalty. We are interested in the performance impact of this.

To have something to compare the results with, our test code generator generates source code for *DLX* as well as a few related solutions:

1. Pure C-to-C function invocation (c).
2. Pure Ruby-to-Ruby method invocation (ruby).
3. Ruby-to-C function invocation via Ruby's extension interface (ext).
4. Ruby-to-C function invocation via Ruby/*DLX* (dlx).
5. Ruby-to-C function invocation via the obsolete Ruby/DL[13] (dl).

Included in this list are Pure C-to-C and Pure Ruby-to-Ruby invocation, to provide insight in the performance of single-language function invocation (as opposed to multi-language function invocation that "*crosses the boundary*").

We conducted the tests by doing 2,500 consecutive function invocations. Then we repeated these tests for the functions with increasing arity.

Early results showed that a *pure Ruby-to-Ruby* method invocation is about 250-300 times slower than a *pure C-to-C* function invocation. Therefore we had to scale the pure C tests until the results were

[13] See section 5.5.3.

approximately in the same range as the other results. We achieved this by increasing the workload for the pure C-to-C invocation by a factor of 400 (see[14]). Thus making the C-to-C invocation workload a total of 1,000,000 consecutive invocations.

We did not test member field access explicitly, but since these are internally handled similar to functions in *DLX*, it is expected that any function invocation with arity 0 gives a good impression of the current performance status for these kind of instructions. Member field access in pure C is considered instantaneous, especially when compared to Ruby and Ruby/*DLX*.

Hardware and Software

All tests were conducted on a notebook with an Intel Pentium IV 2.0Ghz processor and 768MB RAM. The notebook was running a GNU/Linux 2.6.19.2 SMP kernel. All files were compiled with GCC 4.1.1 20060724 (prerelease) and level 2 optimization (-O2) options. We used Ruby 1.8.5 (2006-08-25) to conduct the tests.

Results

The results of the tests are shown in figure 7.14. The data points have been connected via lines to visually improve the graphs (obviously, since a function with an arity of 2.4 bares little meaning). Results for *DLX* are given in bold, using a solid line.

Clearly visible in the results for both sets, and for all contesters, is that an invocation with zero arity is considerably faster. This is likely caused by the overhead for creating a argument stack, which is not required in this case.

Furthermore, we see that Ruby/*DLX* is quite a bit slower than a hard-coded extension written in the Ruby extension API, about a factor two in the case of the "void" set up to a factor of about four in the case of the "dummy" set and for increased arity.

Also clearly visible from both graphs is that on the one hand, integer parameters in the "void" test set are converted from and to Ruby in more or less constant time. While on the other hand, the automatic (un)wrapping of the `PerformanceDummy` parameter and return type objects, that is required for the "dummy" test set, are in linear time. The *ext* version doesn't suffer as much from this, because the wrapper classes are created hard-coded, while in *DLX* they have to be looked up in the *DLX Type Database*.

On average, pure Ruby function invocation is about 30-50% slower than function invocation of aa Ruby extension function. Furthermore, as expected, the –in our opinion– obsoleted Ruby/DL interface is by far the slowest. Ruby/DL also does not really support the "dummy" test set; it can only return a generic pointer (see 5.5.3): There is no support to handle the `PerformanceDummy*` return type of the functions in the "dummy" test set.

In section 6.7.1 we already stated that we are willing to trade in a little performance in favor of the ease of construction and maintainability that our interface offers. Furthermore, we already assume that when

[14] This value was chosen to avoid interference with the other plots in the graph.

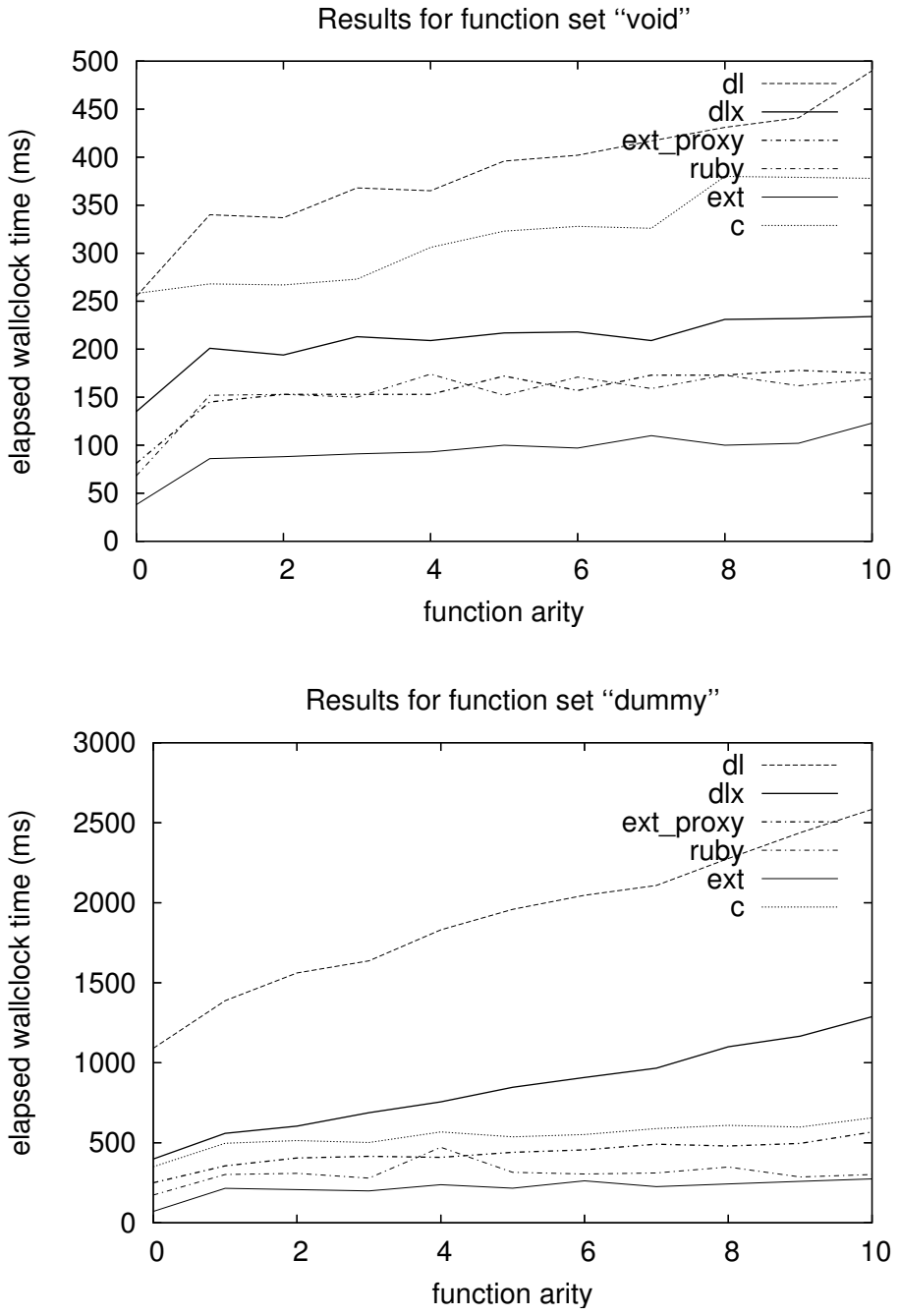

Figure 7.14: Wall clock execution times for function calls of increasing arity. Tests conducted for pure C, pure Ruby, Ruby extension API, Ruby extension API with pure Ruby proxy function, Ruby/*DLX*, and Ruby/DL. Times are for 2,500 consecutive invocations, except for pure C, which was scaled by a factor 400 (1,000,000 invocations). Top: Results for function set "void". Bottom: Results for function set "dummy"

performance is of the up-most importance, the implementation should be done mostly in C. Still, the ultimate goal would be to offer a performance that is comparable to the Ruby extension API (also recall goal 6 from section 4.7.2). Because of this, it is important to investigate a little as to what is causing the performance gap between *DLX* and the Ruby extension interface.

Proxy Functions

One of the reasons *DLX* performs considerably worse than a hard-coded binding, written directly in Ruby's extension API, may be because of an extra pure Ruby proxy function that is used by *DLX*. This pure Ruby proxy function is created dynamically by the *Symbol Binder* (see section 6.3.3). It sits between every invocation and the C proxy function (from the function pool) that we discussed in the previous chapter.

This Ruby proxy function is necessary, because of a current limitation in the Ruby extension interface:

In Ruby, a method can be declared with an optional default value for each parameter. Should such a parameter not be explicitly passed to the function, the default value is taken.

Via the Ruby extension interface one can add a C function[15] as a Ruby method to a Ruby class.

However, the current version of the Ruby extension interface *does not allow* specifying default parameters for Ruby functions that are added this way. Investigation of the Ruby source code did not reveal any theoretical constraints for this not to be possible, so we assume it is merely an omission, because nobody ever needed it before.

Unfortunately, to *DLX* it *is* important to be able to specify a default value, because we could use this default parameter to select the correct C proxy function from the function pool. Because it is not possible, the current implementation of the *Symbol Binder* is forced to dynamically create the pure Ruby proxy function: It does nothing more than invoking the *Function Caller* with the correct parameters such that the right C proxy function from the function pool is used. Had it been possible to specify default parameters via the Ruby extension interface, we would have been able to bind the *Function Caller* to each function directly. This would have spared us the penalty of a call to a pure Ruby function, and the penalties caused by the Ruby interpretion (which are considerable when compared to C).

To illustrate the impact of such an improvement, we have added an extra test for both function sets:

7. Ruby-to-C function invocation via the Ruby extension interface, but with a deliberate pure Ruby proxy function in place (ext_proxy).

The negative impact of such a wasteful pure Ruby proxy function is clearly visible in both graphs from figure 7.14. *DLX* is still slower compared to the slowed-down Ruby extension interface version, but only by 30% ("void" set) up to a factor of, rougly, two ("dummy" set).

[15] For this, the function must have a certain prototype, dictated by the extension API.

Hard-coded Wrapping?

Although the Ruby extension version uses hard-coded object wrapping, as opposed to Ruby/*DLX*, that needs to use a lookup table (via the *DLX Type Database*) to find the correct wrapper class dynamically, we are not convinced that this causes a performance loss of a factor two. So causes for the remaining difference in performance are still a bit unclear.

Discussion

To be honest, we are a bittle startled by the large difference in performance between pure C and pure Ruby function invocation.

While, on one hand, it leads us to believe that, in theory, there should be enough room to do many of the complicating type mapping logic in C, without generally affecting the performance of Ruby invocations, whether they be pure Ruby or Ruby-to-C invocations.

Still, on the other hand, the performance gap may be larger than we anticipated. Luckily, efforts are being made to greatly improve Ruby's performance in the future [59, 69]. This gives us confidence that our choice for Ruby will uphold in the future. Of course, speeding up Ruby decreases the room for doing the complicated type mapping logic, but we think that things will balance out nicely in the end.

7.5.2 Memory Footprint and Startup Performance

We have not yet been able to reliably assess memory footprint or start up performance, as this is a lot less straightforward than the execution speed performance test. Things are complicated because of two main reasons:

1. Ruby and Ruby/*DLX* use an automatic garbage collection scheme. This garbage collector is only guaranteed to reclaim unused memory if system memory is exhausted, clouding actual memory usage of the system at any given point in time.
2. The memory footprint and start up performance are heavily influenced by the API size of a particular shared library (see section 6.7).

The first reason is pretty straightforward and significant in itself. To understand the implications of the second reason, we may need to elaborate a little.

If we want to acquire sensible results, there are two options. We can either

1. compare measured memory footprint or start up performance against a measure of the API size; or we can
2. compare measured memory footprint or start up performance against a *ground truth* coupling between Ruby and C for a set of shared libraries.

To perform the first test we will need to derive a reliable size measure for any API and we will need a reliable size measure for the portion of the API that a particular *test* application is using. We do not yet have either of these.

To perform the second test, we will need a ground truth implementation of an Ruby/C interface for a particular library. Candidates for this comparison can be found in existing bindings for Ruby that make use of the Ruby extension API. However, as it seems, many of these Ruby extension *bindings* are simplified versions, or they are heavily adapted to provide a more Ruby programming style-like API. Really, this would result in comparing apples to oranges. To reliably perform this experiment, we will need to create a set of bindings using the Ruby extension interface that provide exactly the same interface as their counterparts interfaced using *DLX*. We have not yet created such a set of bindings.

Eventhough we have not yet been able to quantify memory footprint and/or start up performance, qualitatively, we think it is safe to say, that:

1. The memory footprint of running *DLX* is currently steep (as this has not been optimized at all (see section 6.7.4)); but, on the bright side:
2. The start up performance feels pretty good (typically <2 seconds), which is probably also because it has seen a little optimization introduced by the *late binding* option of the *DLX Symbol Binder* (see section 6.3.3.1).

7.6 Memory Management Experiments

Considering the discussion in section 6.8.2.1, there is still alot to be researched on memory management strategies in *DLX*. We *did*, however, perform some preliminary research to asses the feasibility of a possible approach: What if we could use the Ruby garbage collector to keep track of and reclaim unused memory? It is made possible by the fact that we wrap C objects inside Ruby objects. If the Ruby object is garbage collected, then why not garbage collect the C object as well? (More details on this approach can be found in section 6.8.2.)

Since not every structure in C can simply be reclaimed by the generic C function `free` (e.g. due to nested structure pointers, etc.), in order to connect the Ruby garbage collector to C objects using *DLX*, we must first introduce additional *DLX* specification syntax. This syntax allows us to specify which specific finalization code must be run for a particular type of structure. Therefore, for each `struct`, *DLX* automatically adds a `define_finalizer` method, which allows just this. Consider the source code fragment in figure 7.15, which is taken from our *SDL* binding.

In line 14, we bind a function that loads an image from disk, which is returned as an `SDL_Surface*`. In C, once an `SDL_Surface*` is no longer required, its memory can be reclaimed by invoking the `SDL_FreeSurface` function (line 15). To connect the garbage collector for objects of class `SDL_Surface`, a finalizer is defined in line 18.

To actually achieve the finalization, we used Ruby's dynamic programming features to create new (or extend existing) methods to establish the actual finalization (see figure 7.16).

```
1  module SDL
2    extend DLXImport
3
4    dlload( "SDL" );
5    dlload( "SDL_image" );
6
7    class SDL_Surface < struct "SDL_Surface",
8    [
9      # .  .  .
10   ]
11   end
12   typealias( "SDL_Surface", "struct SDL_Surface" );
13
14   extern( "SDL_Surface* IMG_Load( char* filename )" );
15   extern( "void SDL_FreeSurface( SDL_Surface* surface )",
16           "freeSurface" );
17
18   SDL_Surface.define_finalizer( SDL, :freeSurface );
19 end
```

Figure 7.15: Defining finalizers for structure objects.

```
1  # automatic memory management using finalizers
2  def self.define_finalizer( finalizer_object, finalizer_call )
3
4    $stderr.puts( "Defining Finalizer in #{finalizer_object}
5                     for type #{self}:  #{finalizer_call}" );
6
7    self.class_eval {
8
9      @@finalizer_object = finalizer_object
10     @@finalizer_call = finalizer_call
11
12     def self.wrap( *args )
13       res = super;
14       ObjectSpace.define_finalizer( res,
15                     self.dlx_finalize( res.to_ptr.to_i ) );
16
17       return( res );
18     end
19
20     def self.dlx_finalize( ptr )
21       proc { |id|
22         $stderr.puts( "Finalizer on #{0xffffffff&id}:  #{ptr}" )
23         @@finalizer_object.send( "#{@@finalizer_call}", ptr );
24       }
25     end
26   }
27 end
```

Figure 7.16: Finalization in *DLX*.

In contrast to the mark+sweep algorithm discussed in 6.8.2 (which is used under the hood), here we use Ruby's public interface for defining finalizers: `ObjectSpace.define_finalizer` in line 15.

As an experiment, we have written a small test program that uses our *SDL* binding to load a ~200kB bitmap image from disk for a consecutive number of times. The test machine was equipped with 786MB of RAM, which is not sufficient to hold 10,000 copies of a 200kB image in memory.

```
1  require "sdl/SDL"
2
3  10000.times do |i|
4     surface = SDL.IMG_Load( "pics/tuxmexico.bmp" )
5     puts( i )
6  end
```

Figure 7.17: The test program that loads a ~200kB bitmap image from disk for 10,000 times.

In the fragment above, it is clear that no reference to the Ruby objects is kept, causing the Ruby object to be reclaimed by the garbage collector. Once the Ruby object is reclaimed, the C object may no longer be accessible from within Ruby, possibly introducing a memory leak.

To assess the difference in behaviour, we have run the experiment twice. First, we ran the experiment without the actual call to the `SDL_FreeSurface` function (line 24 in fig. 7.16). This caused both our test application and test machine to come to grinding halt (we had to pull the plugs): Due to insufficient memory, the machine started swapping and became unresponsive.

In the second test we *did* call the finalizer and the results were as expected:

```
Defining Finalizer in SDL for type SDL::SDL_Surface: freeSurface
.
.
223
224
225
Finalizer on 3688985988: 135001376
Finalizer on 3688922048: 135370136
Finalizer on 3688921148: 134911680
.
.
```

Figure 7.18: Excerpt from the captured terminal output from the test program from figure 7.17.

Here, we see the confirmation that the finalizer is defined for type `SDL_Surface`. Then, we observe alternatingly: 1. A number of images loaded from disk (~100, denoted by the integer, that acts as a serial number). 2. This is followed by a number of finalizers that are called to destroy the `SDL_Surface` objects in C, once the wrapping Ruby objects have been reclaimed by the Ruby garbage collector.

7.6.1 Discussion

Please note, that the experiment above is by no means complete. It assumes that all wrapped instances of a structure must be reclaimed once all references to the Ruby object are lost. In reality, there may still be references available indirectly through another wrapped C object (e.g. as an element of a C array or as a field of another structure).

This example shows the feasibility of using the Ruby garbage collector to aid in the memory management of the referenced C objects. For it to be useful in production environments, the above approach must be made to cooperate with the under-the-hood implementation of *DLX* (i.e. which objects are wrapped, how they are wrapped and the implications for tracking memory). Wrapping Ruby objects must ensure that C objects are *really* no longer accessible from Ruby before they can actually be reclaimed. For example, the *DLX* array class should use the mark+sweep function calls for each object that it references (see also section 6.8.2).

Final Note

Due to the dangers of the finalizers at this stage of development, the above functionality has been deliberately turned off in the current (and coming) public release of *DLX*.

Chapter 8

Conclusion and future work

8.1 Conclusion

In this thesis we have attempted to find a solution to a multi-lingual interoperability problem in which a software component written in a lower-level programming language was to be interfaced to a software component written in a higher-level programming language. After a selection process we chose to interface *Ruby* and C.

The choice for interfacing two languages of such nature was inspired by three scenarios of which we think could benefit from multi-language development under the right circumstances.

From these three scenarios, together with information found in existing literature we derived the goals and requirements for our solution.

The summarization of the most important requirements can be seen as the main hypothesis of this thesis:

Main thesis:

> "Is multi-language software development in Ruby and C possible using a run-time language interface under the condition that the interface is easy to establish, maintainable, efficient in use and effective in all required circumstances?"

To answer this hypothesis in a structured manner, however, we must fall back to the original goals and requirements.

Easy Creation and Maintainability

Goal 1: *Present a solution that keeps it simple to construct the multi-language interface.*
Goal 2: *Present a solution that minimizes the risk of: a.* synchronization errors, *b.* heterogenous mapping, *and c.* black boxing.

We think, that by choosing for a specification (like *DllImport*), rather than a programmatical solution (like *JNI*), we have kept it fairly easy to establish an interface between a shared C library and Ruby. The user is aided by our type closures because they help to detect specification errors, and to prevent subsequent run-time errors, especially during initial specification of the interface. Because of the allowed opt-out, a user can choose to only specify those portions of an API that he really needs.

In contrast to having an ad-hoc programmed solution for every shared library that is interfaced to Ruby, *DLX* is a separate solution, with its own maintainers. It allows users to establish a multi-language interface between Ruby and an arbitrary shared C library by using a simple specification. This is makes us confident that it definately prevents *black boxing*.

Our specification is kept very similar to the original C specification, making it more or less trivial for any person to make the translation from C specification to *DLX* specification. Again, because we use a specification, there is also not much freedom to change the mapping process. We think that this adequately prevents a *heterogenous mapping*, even when more than one person is responsible for the particular multi-language interface.

Because the specification is relatively simple, we also feel that it helps preventing *synchronization errors*, because reflecting API chances back to the *DLX* specification is not a burden. The specification allows the user to keep a clear overview of the specification. However, this is not enough to prevent all forms of synchronization errors. Two forms of synchronization errors persist which are not prevented by choosing for a specification alone.

Shared library version mismatch is the most likely error to occur. This means that at run-time a shared library is used with a different version than the version to which the specification belongs. *DLX* cannot prevent this in general. However, as we have explained in section 6.6.2.2, and demonstrated in section 7.3, such errors can be prevented by the API. Because this sort of synchronization errors is not unique to a solution, such as *DLX*, many of the free software[1] C APIs use these prevention techniques.

To eliminate all forms of synchronization errors, however, we must factor out human influence in the specification process, especially for the trivial parts. Because we have kept the specification so similar to the original C specification, we have researched the possibility of automating the translation from C specification to *DLX* specification.

This has proven to be harder than expected. While our current release of *AutoDLX* was successfully used to automatically translate some APIs, it was insufficiently equipped to automatically translate from C API specification to *DLX* specification in general.

Effectivity

Requirement 2: *Present a solution that targets both interoperability between two original software components as well as interoperability between an original and an existing software component.*
Requirement 3: *Present a solution that makes it possible to establish the interface from just one side.*
Requirement 6: *Provide a solution that always allows an effective interfacing between two software components A and B, regardless of any simplifications added for the sake of maintainability or seamlessness.*

[1] GPL'd, open source, etc.

Because of requirements 2 and 3, we are required to interface with any existing shared C library and its API regardless of the way the API is intended to be used. This is especially important when choosing for a specification rather than a programmatical interface construction approach; a specification not only simplifies the interface construction, it also limits the freedom for "*creative*" possibilities when having to interface with an awkward C API.

Previous experiences have learned that quite a few language interface solutions fail sooner or later on real world examples because of these requirements; in C some pretty onorthodox APIs are possible that make perfect sense afterwards.

We have gone to great lengths in covering as much of the original C specification as possible and presented sensible mappings for each type (section 6.4).

Unfortunately, we could not find a series of tests to demonstrate the effectiveness of a multi-language interoperability between C and one or more other languages. Therefore, in section 7.3, we submitted *DLX* to two experiments of which we knew from previous experience provide a fairly complete test. The results of these experiments were positive and we were able to establish an effective interface from just Ruby without any required additional C source code.

While experiments such as above contribute to our confidence that our solution is effective in most circumstances, we cannot *prove* this in general. Only as more and more existing shared C APIs are interfaced to Ruby using *DLX* our confidence will increase.

Effiency in Use

Because the term "*Efficiency in use*" is rather abstract, we made it more concrete, by selecting three subtopics, each of which we think it can increase the efficiency in use. These are:

1. Seamlessness
2. Dynamic boundaries
3. Portability

Seamlessness

Requirement 4: *Present a solution that provides at least the same level of encapsulation to the higher-level programming language as the lower-level programming language does.*
Goal 3: *Present a solution that hides unnecessary implementation details of the lower-level programming language from the higher-level programming language, as much as possible.*

While we are confident that our solution is effective in establishing an interface with even the most challenging API semantics, *seamlessness* through *encapsulation* and *information hiding* is still a work in progress. We are constantly thinking of new and better ways to improve seamlessness (where applicable, some of these latest ideas have been mentioned in the various discussion sections of section 6.4).

We do achieve the same level of encapsulation as C by, for instance, mapping structure and union member field access to Ruby method calls. We are greatly helped here by the fact that, in Ruby, method invocation and public attribute access is exactly the same from a usage perspective.

We also hide all notions of pointers and, where applicable, we automatically switch interpretation of pointers versus, for instance, nested structures, when accessing such member fields (see section 6.4.5). This way, we avoid the dreaded mixing of `->` and `.` operators. In Ruby, there is only one such operator (the `.` operator) and it is used in all similar circumstances. So we automatically map both C operators to the one Ruby operator, which allows use to always use the `.`[2].

This, however, brings us to the limitations of achieving seamlessness through information hiding. Some things simply cannot be hidden, or they become somewhat dangerous if they are.

For instance, in C, there is a difference between:

- `struct Triangle ary[10];` and
- `struct Triangle* ary[10];`.

This sort of difference cannot automatically be resolved by *DLX*. Therefore, it cannot be kept hidden from the user.

In *DLX* there is no notion of pointers, but because we see no other parts of our type mapping having to deal with a similar problem, we simply introduced new terminology to make the end users aware of these differences. We coined the former array a *contiguous array* and the latter array *non-contiguous*. It's a compromise between fully exposing the notion of pointers, and, making end users aware of the fact that sometimes life just isn't that beautiful.

Other things can become very dangerous. For instance, assigning a wrapped C structure object to a nested structure field results in a duplication of information (see section 6.4.5). This is explicit in C, but currently implicit in *DLX*. It may lead to strange situations if mutations are made to either object later. While we suggest a possible way of dealing with it, we are unsure if and how such solutions may undermine the effectivity of *DLX*.

Dynamic Boundaries

Goal 4: *Provide a solution that makes it as easy as possible for adapting the boundary between the software components, making it easy to change what goes to which side of the fence.*

In section 7.4, we showed an experiment in which we tested how easy it was to adapt the boundary using *DLX*. In our experience, this was a fairly easy thing to achieve. *DLX* made it very clear, *where* the current boundary was, and shifting attributes from Ruby to C and vice versa was simply a matter of *moving* the specification up or down a few lines in the source code, while doing a little translation in the process.

[2] This is demonstrated in section 7.3.1, figure 7.5.

When we moved the attributes from C to Ruby, it was remarkable that we did not have to change any other line of code to get the demonstration application working again[3].

Of course, our experiment was a more simple test case, where only a single independent attribute was switched sides. In real world examples it is expected that, because of dependencies, a little more effort is required, but this is unavoidable.

Portability

Requirement 5: *A solution must be presented that allows for the separation of concerns regarding portability related issues.*

We feel, that our solution works well, when it comes to portability. First of all, the implementation itself seems fairly portable, despite its run-time memory layout reconstruction. We have confirmed it to be working on a variety of platforms in table 2, section 7.2.

Secondly, we have hidden many of the platform dependent details from the user, such as, differences in library naming conventions, or differences in byte sizes of various types (`int`,`long`,...).

To further test the portability, in section 7.2, we submitted *DLX* to a portability field test, where we wrote a small application in Ruby, and ran it on two very distinct computer systems. Apart from the obvious difference in screen size, the application ran exactly the same on both platforms, without having to alter any of the Ruby code. This field test increased our confidence that we succeeded in writing a portable solution.

Additional Goals and Requirements

In chapter 4 we also listed a number of goals and requirements that we derived from existing literature. These embody known issues with multi-language development in general. While they are important topics for a complete solution, which we obviously strive for with *DLX*, they have not been the focus point of this work.

Documentation

Goal 5: *Strive for a solution that requires less (detailed) documentation.*

By choosing for a specification that is so similar to the original specification, we hoped that this would lead to a solution that required less documentation. While we feel that our solution *does* require *less* documentation than it would have, were it a programmatical solution such as JNI, it still requires a lot of documentation, as can be witnessed from chapter 6 (even if the information presented there is not specifically end user documentation).

[3] We can possibly attribute this to the seamlessness of the interface with *DLX*.

Performance

Goal 6: *Provide a solution with a* computational, start up *and* perceived *performance that strives to be comparable to the performance of the higher-level programming language as much as possible.*

Performance has not been a focus point in this thesis. Still we have taken some performance experiments to estimate the current status. From these results, it is clear that a pure C to C function call is many times faster than a pure Ruby to Ruby function call. This gives us room to execute many of the instructions needed for interface mapping in C, without affecting the performance of Ruby too much.

Our current implementation is still slower than our ground truth comparison: A hard-coded multi-language interface, written directly in Ruby's own C extension interface. Part of the difference in performance could be explained, while substantial part must still come from future performance improvements.

Still, if computational performance is important, then it is best to isolate these algorithms and loops, so that they can be performed in pure C.

Start up performance is not quantified by us, neither is memory foot print performance. We have given reasons for this and explained why this is harder to quantify than execution speed performance. Part of the problem was caused by the fact that Ruby/*DLX* holds all API related type information in memory, even when just a small portion of the API is actually used.

For this we suggested two solutions, *late binding* and *profiling*, of which the former was actually implemented. While not formally quantified, we experienced a significant improvement on the perceived start up performance because of this.

Overall perceived startup performance was qualified as reasonable to fast (altough, as mentioned, depending on the size of the API). Perceived memory footprint performance remains reasonably bad.

Memory Management

Requirement 7: *Provide starting points for addressing memory management using the found solution.*

One important issue that is encountered when doing multi-language software development is memory management. Often the two interfaced components have different memory management strategies.

Although not a focus point either, in this thesis we argued that we see possibilities of connecting Ruby's automatic garbagage collecting scheme to C objects that cross the boundary (i.e. we do not consider any objects that remain internal to a shared C library). We feel that this is made possible because in Ruby, at implementation level, *all* objects –and not just *DLX* wrapped objects– are merely just wrappers around the original C objects.

To briefly test the feasibility of such an approach, we submitted *DLX* to a preliminary experiment in which we automatically garbage collected C objects, when they were no longer accessible from Ruby.

The results were promising (section 7.6), but it is to early to conclude anything from such preliminary

research. There are many things that may prove to be too hard to solve adequately by this approach in the future. Therefore, this preliminary research can only act as a starting point for future research, which is everything we wanted of requirement 7 in the first place.

Threading

Requirement 8: *Provide starting points for addressing threading using the found solution.*

Due to time and size constraints, threading was not thoroughly experimented with, in this work. Some of the experiments that we have taken use native threading, while others use Ruby threading. Although threading appeared to be working in those experiments, in this thesis we only provided *suggestions* with respect to threading, as we could not provide quantified results. The suggestions we made mainly stated that it is best to prefer Ruby's threading over native threading, and that, without future research, it is best to avoid mixing Ruby threads and native threads.

8.2 Future Work

DLX is still a work in progress and there is still much work to do. In this section we shall highlight some of the most prominent things that still need to be done. Throughout chapter 6 we also give small hints for future improvements for, sometimes, very specific subjects.

Maintainability Improvements

A multi-language interface specified in *DLX* is already reasonably maintainable. However, as API specifications grow, or as larger portions of a C API specification are translated to a *DLX* specification, it becomes harder to detect slight API changes.

One way of eliminating these errors, as well as instantly boosting maintainability to a whole new level, would be to automatically translate the C API specification to a *DLX* specification. As we said earlier in this chapter, we already started to work on this with *AutoDLX*, but the current implementation is not applicable in general.

In the future we would like to redesign *AutoDLX* to better address the difficulties that we encountered while using it. Ultimately, because both specifications are so similar, we feel that it must be possible to automatically translate a C API specification to a *DLX* specification in general, with minimal input from an end user.

Seamlessness Improvements

Higher-level programming languages, such as Ruby, and lower-level programming languages, such as C, are fairly distinct in feature set. This is one of the reasons that we selected them in the first place.

However, this also means, that absolute seamlessness is very hard to achieve. We showed in this thesis that sometimes it is just not possible to map one type in C seamlessly to another type in Ruby.

However, there are things that can be improved greatly. For instance, when we consider the amount of encapsulation.

Currently our solution provides an encapsulation at the higher-level language, with at least the same level of encapsulation of the lower-level langauge. This means, for instance, that structure member fields can be accessed recursively, and that all member fields are wrapped in the corresponding types (see figure 5.7 in section 5.4.2.1).

However, in the future, we must strive to reach a level of encapsulation that provides an encapsulation that is more natural to the higher-level language. Two concrete examples come to mind:

1. Improving encapsulation of C arrays by providing extra type specification syntax (as we explained in section 6.4.7).
2. Encapsulating C functions inside corresponding Ruby classes, so that they may become instance methods instead of class methods.

This last example must be explained a little further. Recall that, in Ruby, attributes and behaviour are both encapsulated by a class. In C, attributes can be encapsulated by, for example, a structure type, while individual instructions can be encapsulated by functions. Many times, however, a function is specified that operates on an object of a structure or union type.

Collections of such functions that operate on objects of a similar type are often distinghuished from other functions by either,

- using special naming prefix; or
- by the fact that the first argument of such a function is always of the (structure) type that it operates on.

It would be very useful, if we could devise a way to automatically associate such a collection of functions with the particular structure type. This way we may be able to provide a better encapsulation on the Ruby end of the interface.

Portability Improvements

Because of the lack of access to different hardware and operating systems, we have not been able to verify correctness of *DLX* on all platforms that we would like to support.

In the future we would like to see the number of platforms on which *DLX* is successfully tested (and used) increase.

One particularly interesting platform would be, any computer system that uses a *big endian architecture* because our memory layout reconstruction algorithm is likely to require some updates if it is to work correctly on such an architecture.

Perhaps this can be combined with the wish to support MacOS X, since we have received quite a few e-mails from people who want to use *DLX* on MacOS X.

Performance Improvements

Since performance has not been a focus point in the current work, it is needless to say that there is still a lot of room for improvement in this regard.

One way of improving performance, was already suggested in this thesis in section 6.7.4: *Profiling*. By adding profiling support, we can minimize the penalty of having to load (and keep in memory) a large API specification of which only a small portion is actually used by any individual application.

Memory Management and Threading

Both memory management and threading have not been a focus point, but for memory management we did provide a nice starting point for future work.

We also still have to do some extensive research on using Ruby threading in combination with a shared library that provides native threading of its own.

Quantified Productivity Gain

Finally, we conclude our growing list of future work by looking back to scenario 1 in chapter 2. Recall that one of the main reasons for interfacing a higher-level to a lower-level language was done because of an expected productivity gain, by developing non-computational intensive (support) code in a more productive and more concise (i.e. a higher-level) programming language.

Because providing reliable quantifiable data that either confirms or rejects this assumption was well beyond the grasps of this Master's Thesis, we assumed it to be true and focussed on secondary prerequisites for providing our the solution instead.

Ultimately, however, it would be very nice if, somehow, we can reliably quantify the software productivity of a software project with single-language software development (i.e. *without* using our solution) versus software productivity of multi-language software development *with* our solution to see if, and if so, what the actual achieved performance gain turns out to be.

Bibliography

[1] URL http://www.ruby-lang.org.

[2] The dotgnu website, 2007. URL http://www.gnu.org/software/dotgnu/.

[3] The .net developers website, 2007. URL http://msdn2.microsoft.com/en-us/netframework/default.aspx,.

[4] Mozilla firefox add-ons website, July 2007. URL https://addons.mozilla.org/en-US/firefox/.

[5] The gtk+ object system, July 2007. URL http://developer.gnome.org/arch/gtk/object.html.

[6] The mono project's website, 2007. URL http://www.mono-project.com/Main_Page.

[7] Dynamic language support on the jvm, 2007. URL http://www.artima.com/lejava/articles/dynamic_languages.html.

[8] The dlr: Dynamic language runtime, 2007. URL http://www.ironpython.info/index.php/The_DLR:_Dynamic_Language_Runtime.

[9] Wikipedia on common language infrastructure, . URL http://en.wikipedia.org/w/index.php?title=Common_Language_Infrastructure&oldid=143866380.

[10] Wikipedia on representational state transfer, . URL http://en.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=150480366.

[11] Wikipedia on rigid body, . URL http://en.wikipedia.org/w/index.php?title=Rigid_body&oldid=143939292.

[12] Wikipedia on information hiding (computer science), . URL http://en.wikipedia.org/w/index.php?title=Information_hiding&oldid=141028903.

[13] Wikipedia on memory management, . URL http://en.wikipedia.org/w/index.php?title=Memory_management&oldid=140503365.

[14] Wikipedia on (software) porting, . URL http://en.wikipedia.org/w/index.php?title=Porting&oldid=138164846.

[15] Wikipedia on prosumer as producer and consumer, . URL http://en.wikipedia.org/w/index.php?title=Prosumer&oldid=146954064.

[16] Wikipedia on separation of concerns (computer science), . URL http://en.wikipedia.org/w/index.php?title=Separation_of_concerns&oldid=140914997.

[17] Wikipedia on threading (computer science), . URL http://en.wikipedia.org/w/index.php?title=Thread_%28computer_science%29&oldid=138197462.

[18] J. Gaffney A. Albrecht. Software function, source lines of code and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(11):639–648, 1983.

[19] Allan J. Albrecht. Measuring application development productivity. *Proc. IBM Application Development Symposium*, 1979.

[20] Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 147–155, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-797-9. doi: http://doi.acm.org/10.1145/239098.239123.

[21] Victor R. Basili and Marvin V. Zelkowitz. Analyzing medium-scale software development. In *ICSE*, pages 116–123, 1978.

[22] Fabrice Bellard. Tiny c compiler - website, 2007. URL http://fabrice.bellard.free.fr/tcc/.

[23] J. A. Bergstra and P. Klint. The discrete time toolbus – a software coordination architecture. *Sci. Comput. Program.*, 31(2-3):205–229, 1998. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(97)00021-X.

[24] Jan Bergstra. Philosophy of open computing: Phoocom, 2007. URL http://www.phil.uu.nl/~janb/phoocom/top.html.

[25] Antoine Beugnard. Method overloading and overriding cause encapsulation flaw: an experiment on assembly of heterogeneous components. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1424–1428, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-108-2. doi: http://doi.acm.org/10.1145/1141277.1141608.

[26] B. W. Boehm and W. L. Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference*, pages 63–82, 1992.

[27] Barry Boehm and Victor R. Basili. Gaining intellectual control of software development. *Computer*, 33(5):27–33, 2000. URL citeseer.ist.psu.edu/boehm00gaining.html.

[28] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–100, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-450-9. doi: http://doi.acm.org/10.1145/503272.503282.

[29] Weiser Boehm, Demers. A garbage collector for c and c++ - website. URL http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[30] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997. URL citeseer.ist.psu.edu/cardelli97type.html.

[31] Dave A. Chappell. *Enterprise Service Bus*. O'Reilly.

[32] Wesley J. Chun. *Core Python Programming, 2nd ed.* Prentice Hall PTR / Pearson Education, 2006. ISBN 0132269937.

[33] The Common Software Measurement International Consortium. History of functional size measurement, march 2007. URL http://www.cosmicon.com/historycs.asp.

[34] W3 Consortium. Simple object access protocol (soap) 1.1, may 2000. URL http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[35] Mozilla Corp. Mozilla firefox, July 2007. URL http://www.mozilla.com/en-US/firefox/.

[36] Xerox Corporation. Inter-language unification. URL ftp://ftp.parc.xerox.com/pub/ilu/2.0b1/manual-html/manual_1.html.

[37] Andy Hunt Dave Thomas, Chad Fowler. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, 2004. ISBN 0974514055.

[38] Ryan Davis and Eric Hodel. Ruby inline project page, 2007. URL http://rubyforge.org/projects/rubyinline/.

[39] Bruce Dawson. Game scripting in python. In *Game Developers Conference Proceedings*, 2002. URL http://www.gamasutra.com/features/20020821/dawson_pfv.htm.

[40] Hayco de Jong and Paul Klint. Toolbus: the next generation, 2003. URL http://homepages.cwi.nl/~paulk/publications/fmco02.pdf.

[41] Ruben Geerlings and Marcel Toele. Javaterms, 2002. URL http://www.dwarfhouse.org/mtoele/javaterms.pdf.

[42] GNU. Gnu compiler collection - website, 2007. URL http://gcc.gnu.org/.

[43] International Function Point User Group. Ifpug website, March 2007. URL http://www.ifpug.org/.

[44] Samuel P. Harbison III and Guy L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, pub-PH:adr, fourth edition, 2002. ISBN 0133262243. URL http://www.CAReferenceManual.com/;http://www.phptr.com/ptrbooks/ptr_013089592X.html.

[45] Samuel P. Harbison III and Guy L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, pub-PH:adr, fifth edition, 2002. ISBN 0-13-089592-X. URL http://www.CAReferenceManual.com/;http://www.phptr.com/ptrbooks/ptr_013089592X.html.

[46] Michi Henning. The rise and fall of corba. *Queue*, 4(5):28–34, 2006. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1142031.1142044.

[47] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. ISBN 9780521692694.

[48] T. Capers Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1): 39–63, 1978.

[49] T. Capers Jones. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., New York, NY, USA, 1991. ISBN 0-07-032813-7.

[50] Tim Lindholm and Frank Yellin. The java^{TM} virtual machine specification, 2007.

[51] CAI Electronic Magazine. Focus on barry boehm, software estimation pioneer. URL http://www.compaid.com/caiinternet/ezine/barryboehminterview.pdf.

[52] Microsoft. embedded visual c++ 4.0 - website, 2006. URL http://www.microsoft.com/downloads/details.aspx?FamilyID=1dacdb3d-50d1-41b2-a107-fa75ae960856&DisplayLang=en.

[53] Harlan D. Mills. *Software Productivity*. Dorset House Publishing Company, Incorporated; Reprint edition, 1988. ISBN 0-93-263310-2.

[54] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 19 April 1965. URL ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf. Director of R&D Labs at Fairchild Semiconductor http://www.intel.com/intel/annual96/bio_moor.htm.

[55] Andrew M. Phelps and David M. Parks. Fun and games: Multi-language development. *Queue*, 1 (10):46–56, 2004. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/971564.971592.

[56] Frank Piller. Glossary: Mass customization, open innovation, personalization and customer integration. URL http://www.mass-customization.de/glossary.htm.

[57] R. Pressman. *Software Engineering, a Practitioner's Approach, fourth edition*. McGraw-Hill, 1997.

[58] Bryce Ragland. Measure, metric, or indicator: What's the difference? *Crosstalk: The Journal of Defense Software Engineering*, March 1995. URL http://www.stsc.hill.af.mil/crosstalk/1995/03/Measure.asp.

[59] Koichi Sasada. Wikipedia on: Yarv: Yet another ruby vm, July 2007. URL http://en.wikipedia.org/w/index.php?title=YARV&oldid=146754628.

[60] Mary Shaw. The tyranny of transistors: What counts about software? In *Fourthe Workshop on Economics-Driven Software Engineering Research*, May 2002. URL http://www.cs.cmu.edu/~Compose/ftp/shaw-sw-measures-fin.pdf.

[61] Spronck, Pieter, Ponsen, Marc, Sprinkhuizen-Kuyper, Ida, Postma, and Eric. Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248, June 2006. ISSN 0885-6125. doi: 10.1007/s10994-006-6205-6. URL http://dx.doi.org/10.1007/s10994-006-6205-6.

[62] Jeff Kesselman Steve Wilson. *Java Platform Performance: Strategies and Tactics*. Prentice-Hall, pub-PH:adr, 1st edition edition, 2000. ISBN 0201709694. URL http://java.sun.com/docs/books/performance/;http://java.sun.com/docs/books/performance/1st_edition/html/JPPerformance.fm.html.

[63] Juha Taina, March 2007. URL http://www.cs.helsinki.fi/u/taina/ohtu/fp.html.

[64] Owen Taylor. Chapter 21. writing your own widgets, July 2007. URL http://www.gtk.org/tutorial/c2182.html.

[65] Alvin Toffler. *The Third Wave*. William Morrow and Company, New York, 1980.

[66] Ian Main Tony Gale and the GTK team. Gtk+ 2.0 tutorial.

[67] Kenneth West. Soa vs. corba: Not your father's architecture. URL http://microsites.cmp.com/documents/s=9063/ilc1086397013472/.

[68] Paul R. Wilson. Uniprocessor garbage collection techniques. In *In International Workshop on Memory Management*, September 1992. URL ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps.

[69] c.s. Yukihiro Matsumoto. Ruby 2.0: Rite.