Norwegian University of Technology and Science

Faculty of Engineering Science and Technology

Department of Engineering Design and Materials

Integrating general-purpose software design tools into KBE-development

Author:
Teodor Ande Elstad
Simen Haugerud Granlund

Supervisors:
Ole Ivar Sivertsen, NTNU
Ivar Marthinusen, NTNU
Christos Kalavrytinos, NTNU
Mahsa Mehrpoor, NTNU
Geir Iversen, AkerSolutions

May 5, 2014

NTNU – Trondheim
Norwegian University of
Science and Technology

THE NORWEGIAN UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF ENGINEERING DESIGN
AND MATERIALS

**PROJECT WORK AUTUMN 2013
FOR
STUD.TECHN. TEODOR ANDRE ELSTAD AND
SIMEN HAUGERUD GRANLUND**

INTEGRATING GENERAL-PURPOSE SOFTWARE DESIGN TOOLS INTO KBE DEVELOPMENT
Integrere generiske verktøyer for programvareutvikling i KBE utvikling.

Most software tools for KBE development is today based around special purpose Modeling frameworks like AML. While this approach has been very successful, there is still room for improvement. This project aims at investigating the possible benefits of utilizing generalpurpose software development tools as a part of KBE development. The following areas of interest will be studied, with KBE development at Aker Solution KBeDesign serving as a case study. Integration of general purpose software development tools into the AML modeling framework. Usage of general purpose software development tools for KBE development alongside the AML modeling framework. Usage of general purpose Software development as standalone tools for KBEdevelopment.

The assignment includes:

1. A literature study of software development tools and KBE technology.

2. A valuation of generic software development tools with respect to KBE development

3. Specification of a set of software tools well suited for implementing KBE applications

4. Evaluate how the specified tools could support implementation in the AML framework

5. If time allow, develop an AML prototype using the specified tools.

Students are required to submit an A3 page describing the planned work three weeks after project start both as a paper version and as a pdf-file. A template for this sheet is found on IPM's web-page, using the link http://www.ntnu.no/ipm/prosjekt.

Performing a risk assessment of the planned work is obligatory. Known main activities must be risk assessed before they start, and the form must be handed in within 3 weeks of receiving the problem text. The form must be signed by your supervisor. All projects are to be assessed, even theoretical and virtual. Risk assessment is a running activity, and must be carried out before starting any activity that might lead to injury to humans or damage to materials/equipment or the external environment. Copies of signed risk assessments should also be included as an appendix of the finished project report.

No later than 1 week before the deadline of the final project report, an A3 page summarizing and illustrating results obtained in the project work, is required to be submitted (paper and pdf version).

The projects are presented orally by the students on 18 October. Participation during the presentation is compulsory for all project students.

Official deadline for the delivery of the report is 19 December (by 15:00).The report should be delivered in two paper copies and one electronic version via email to Jorunn.hvalby@ntnu.no.

When evaluating the project, we emphasize how clearly the problem is presented, the thoroughness of the report, and that the student gives an independent presentation of the topic using their own assessments.

The report must include the signed problem text, and be written as a scientific report with summary of important findings, conclusion, literature references, table of contents, etc. The preface should specify the modules (fordypningsemne) chosen within the student's specialization. Specific problems to be addressed in the project are to be stated in the beginning of the report and briefly discussed. The report should not exceed thirty pages including illustrations and sketches.

Additional tables, drawings, detailed sketches, photographs, etc. can be included in an appendix at the end of the thirty page report. References to the appendix must be specified. The report should be presented so that it can be fully understood without referencing the Appendix. Figures and tables must be presented with explanations. Literature references should be indicated by means of a number in brackets in the text, and each reference should be further specified at the end of the report in a reference list. References should be specified with name of author(s) and book, title and year of publication, and page number.

Contact persons:
From the industry :
Geir Iversen, Aker solutions

From NTNU :
Ivar Marthinusen
Christos Kalavrytinos
Mahsa Mehrpoor

Ole Ivar Sivertsen
Supervisor

Integrating general-purpose software design tools into KBE-development

Project assignment TDT4560

Stud.techn. Teodor Ande Elstad, Stud.techn. Simen Haugerud Granlund
Department of Engineering Design and Materials
Faculty of Engineering Science and Technology
Norwegian University of Science and Technology

Abstract

The goal of this report, is to investigate the possible benefits of using general purpose software tools as a part of KBE development.

The report focuses mainly on two cases. The first case, is the possibility of extending the AML modeling framework, by interfacing with another general purpose language. By finding strategies for doing this, we are able to build a working prototype integration of the AML framework into the Sublime Text 2 editor.

The second part, explores general software tools more freely, as the possibility of building KBE models from the general purpose programming language Python is explored. This strategy gives easy integration with general software tools, and we are able to explore their benefits without the need to integrate the tool into an existing KBE development system.

Finally, the authors conclude that extending a KBE systems with modules form general software tools is beneficial, but enabling the KBE developer to extend the AML modeling framework, would be more efficient if some changes where made to AML.

Preface

This report marks the end of the project assignment integrating general-purpose software design tools into KBE development, TDT4560, given as part of the Engineering Software Laboratory, at department of Engineering Design and Materials, Norwegian University of Science and Technology.

The project was conducted by Stud.techn. Teodor Ande Elstad, Simen Haugerud Granlund. While undertaking the project, the authors of this document where enrolled in the specialization courses TMM1 Product modeling And TMM2 Product simulation.

The authors would like to thank Professor Ole Ivar Sivertsen and academic fellow Ivar Marthinusen at NTNU, as well as Geir Iversen and Olivier Doucet at Aker Solutions KBeDesign, for help, guidance, feedback and inspiration throughout the project.

Trondheim, May 5, 2014

Teodor Ande Elstad, Simen Haugerud Granlund

Contents

List of Figures

List of Tables

1 Background

1.1 KBE-systems

Knowledge Based Engineering (KBE) is the art of using computerized knowledge to automate engineering design. KBE systems make the engineers able to write dedicated programs that preform complex and specific engineering task efficiently. It can, for example, take engineers weeks to iterate to a final design including geometry, simulations and documentation. With a KBE application the same work could be archived within days due to reuse of knowledge and design.

A clear and strict definition of KBE has proven almost impossible to define. The term is so broad that it is difficult to embrace all aspects. One good and suitable definition is written by Chapman, see Definition 1.

Definition 1. *KBE represents an evolutionary step in Computer-Aided Engineering (CAE) and is an engineering method that represents a merging of Object-Oriented Programming (OOP), artificial intelligence (AI) and Computer-Aided Design (CAD) technologies, giving benefit to customized or variant design automation solutions [7].*

KBE is a combination of object-oriented programming and artificial intelligence with computer aided design. It is designed to allow complex rules, heuristics, artificial intelligence and integration of other suitable systems. KBE is therefore more powerful and flexible then standard CDE/CAD systems.

KBE systems is often build on an open architecture [7]. This enables the system to incorporate different software, like geometric kernels, CAD systems, FEA systems, databases, and graphical components. This makes the KBE system more suitable to be integrated in a product lifecycle process. Figure 1.1 show one possible architecture of the AML system.

1.2 Modern general software development tools

General Software Development (GSD) is a synthesis of many disciplines. From modeling, design, code generation, project management to testing, debugging and deployment. The variety makes software development a complex and often a difficult task. Through the last decade software development has come a long way in trying to minimize this complexity. General software development tools form an important component of this evolution. These tools have particularly made an

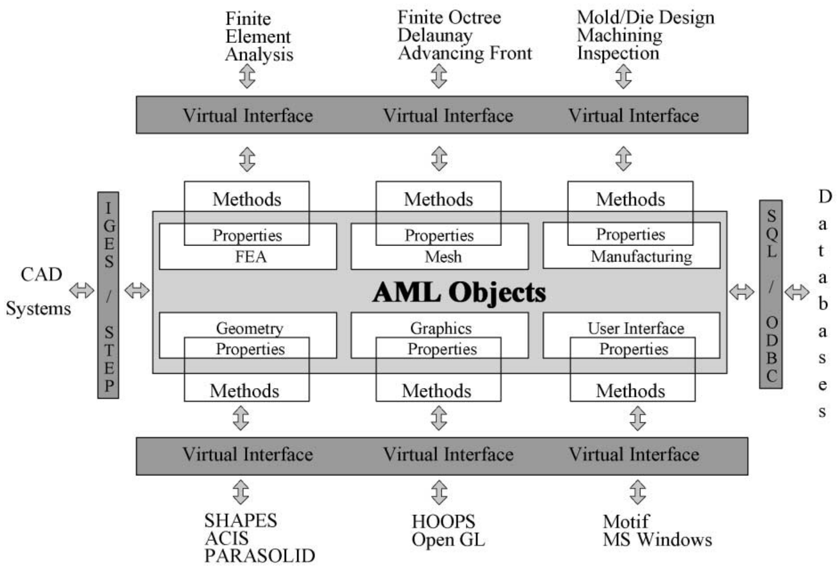


Figure 1.1: *AML virtual layer architecture [31]*

impact in the robustness, portability, re-usability and effectiveness in the software development.

GST tools is designed to help a developer with repetitive and time consuming tasks. Debugging, testing, creating and maintaining code are some of these tasks. The term is usually used about relatively simple programs that can be used in many different software development environments.

Definition 2. *General software development tool is a program that software developers use to create, debug, maintain, or otherwise support other programs and applications.*

These tools are used in modern software development environments. They are often general, which means they can be used om many different development environments. Some examples are testing frameworks, debuggers, and IDE's. .

1.2.1 Test Framework

Testing frameworks makes the developer able to test, maintain and control the quality of his code. Is is an execution environment for automated tests and is the overall system in which the tests will be automated.

Definition 3. *Testing Framework is the set of assumptions, concepts, and practices that constitute a work platform or support for automated testing.*

From definition 3 we can conclude that the testing framework is responsible for:

- Defining how the express expectation should be formated.
- Executing the tests.
- Reporting results.
- Creating a mechanism to drive the application under test.

In short Test frameworks helps teams organize their test suits in turn to help improve the efficiency of testing.

1.2.2 Debugger

Typically a debugger offers a lot of different features. Some of these features are step-by-step execution, symbol resolver, breakpoints, GUI interface and so on. It is used to examine a program for errors and it is a exceptionally practical software tool.

Definition 4. *A debugger is special program used to find errors in other programs. A debugger allows a programmer to stop a program at any point, examine the code and variables and change the values of variables.*

Debuggers are normally used to find error and bugs in the code base, but can also be used to get a better understanding of the code base. One can for example follow a variable or object through different transformations throughout the program execution.

1.2.3 Integrated Development Environment

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to a software developer. It combine many features of different tools into one package. An IDE can for example provide a text editor, a compiler and/or interpreter, text and file search, a debugger, test runner, and other efficiency enhancing features.

1.3 Combining KBE and general software development

Todays KBE tools have proven itself to be able to produce solutions of great value. The demand for new solution and further development is increasing. This increase in demand has reviled some challenges in terms of code maintenance, code quality and make the application not only accessible for “natural born hackers” [26]. The solution can be based on a more advanced developer environment with more support tools and better methodologies.

GSD has today come a long way in terms of environment, support tools and work methodologies. Developers and project planers can pick and choose from a range of different utilities. The tools are portable and can be used by all kinds of development. This evolution started in the 80’s by the academic community. When the software industry, in the 90’s, started to acknowledge the need the evolution flourished. Through the 1990’s and 2010’s the software industry saw a boost in productivity, efficient, code quality, project control, coordination within the programming groups and speed to delivery [8].

KBE development, on the other hand, is today based around special-purpose modeling frameworks and programming languages. This makes the portable GSD tools not as easily integrable. While the KBE development has been very successful, there is steel room for improvement. There is today a lot of tools that is not implemented in the KBE environment, that in the software industry has made a deep impact.

This gap between KBE development and GSD is recognized by many KBE industries today. KBeDesign at Aker Solution has one of these companies. They have investigates many possibilities and tools, like contionous integration and control version systems. They are also trying to implement a testing framework for the AML environment. This framework is AUnit and started as a master thesis at the Norwegian University of Science and Technology (NTNU) [2].

2 Evaluation of generic software development tools

In this chapter, we will look at some of the different tools used for GSD, and specify a set of tools we will focus our following investigation on.

2.1 Currently used GSD tools

In this section we will try to list some of the tools, currently used for GSD. Due to the large amount of different tools available, we do not aim to make an exhaustive list. Instead we will focus our list of tools associated with the popular Agile method for development , as well as a section of well used traditional tools, used for software development.

There is two reasons for focusing on Agile development tools. The first is the popularity of using this method has in GSD. The second is that using agile principles with KBE, is a practice currently being developed and utilized buy AkerSolutions KbeDesign. The benefits of such tools have already been studied by both AkerSolutions and Lars Barlindhaug in his thesis [2], and the fact that tools for agile development is currently being in development, makes interesting to study the possibility of integrating ready-made solutions from the world of GSD.

Agile development has introduced a lot of new development tools. Tools like unit test frameworks, which enables the developer to run the same tests over and over again. Mock frameworks, which simplifies and standardizes the test writing. Test runners, which allows tests to be automated [25] [23]. Alongside tools for testing, agile projects often utilize tools for continuous integration (CI). CI is the process of automating the building and testing process in software development, and in this thesis Lars Barlinghaug showed the feasibility and relevance of such practices in an KBE environment [2].

Although Agile development is currently receiving a lot of attention in the GSD community, the use of support tools in GSD is not limited to the tools associated with agile development. One of the most overlooked tool is the editor or IDE the developer uses to write the code. The general practice in GSD is that several editor options exist for every programming language, all though some might be better suited than others.

Debugging tools, which allow the developer to step through the execution of the program and inspect variable instantiation, is a common support tool. The value of being able to perform such inspections on a KBE model

The use of previously written modules and libraries, is also a practice, widely used in GSD. Using previously written code, makes development faster and less error prone. This is not a new concept for KBE development, indeed AML facilitates the creation of reusable modules and libraries to a great extent. Would it be possible to also use modules and libraries from the larger GSD community? This could hypothetically extend the choices of the KBE developer, something that makes this an interesting case to study.

IDE or editor used to write the code is an essential tool. This is an interesting case to study, since XEmacs currently is the only editor currently supported by the popular AML modeling framework. Could other editors of IDEs bring new features to AML?

An often overlooked tool, that all software developers use at some point, it the tools available for learning a particular software system. This is especially relevant for a KBE system, Rocca states that: the role of KBE developer should be easily undertaken by engineers and not restricted to “natural born hackers” [26]. Good learning tools is a part of making this possible. This is of particular interest to the writers, as we will have to undergo training in AML as a part of the project.

2.2 Tools studied in this project

Due to time constraints, we will not be able to study all the possible GSD tools relevant for KBE development. In this section we will give a brief argument for why we have chosen to study certain tools.

The tools for Agile development, is of particular interest, due to the investigation made by AkerSolution and Lars Barlindhaug, and we will focus on integration of previously written tools for testing.

Using modules and libraries from a GSD language is also something that we’ll study. KBE development is a relatively smaller field than GSD, and as a result there is fewer ready-made components to choose from. Enabling the KBE developer to access a much larger amount of previously written components. This will require us to study the possibilities for integrating with AML from other programming languages.

Integrating AML with another editor is also something that we’ll study. The lack of several supported code editing tools for AML is puzzling, as a large amount of code editing options are available for GSD. Integrating AML with another code editing tool, will also serve as a test of the strength of our findings in how to interface with AML.

Finally we'll look at the possibility of using a debugger in KBE development. A debugger is a feature which is lacking from AML, but is widely used in most of GSD.

3 Extending AML

3.1 Interfacing with the AML modeling framework

In order to be able to extend the AML modeling framework, without writing every extension from the ground up, we need to understand how we can interact with AML from other applications. In this section we will look at how to control AML from other applications. We will first discuss control through the console, and then through the use of sub-processes in Python.

3.1.1 Interfacing with AML from the console

Using the console is a simple but powerful way to control a computer. Not only can the user give complex instructions in a short and concise manner, these instructions can be automated through batch-scripting, the resulting batch-files can be executed from other programs. Commands and results can even be streamed from one program to another. Although not the norm for consumer applications, a solid command line interface is the standard for GSD tools.

In this section we will compare the AML command line interface with the command line interface of comparable general purpose programming languages. We will identify similarities and features suitable for improving the AML command line experience.

We started our investigation by searching for documentation on using AML as a command line tool. This topic is not covered in the AML basic training [32] and we where unable to get hold of such documentation from Technosoft. This was unfortunate, as we had to base our investigation on previous work done by Olivier [10], reverse engineering of the existing start-up batch-files supplied with AML, as well as a good bit of experimentation.

To investigate AML as a command line tool, we choose two simple tasks, launching and interacting with the interpreter from the command line, as well as running a previously written program, and displaying the results in the command line. This would be compared with the procedures for accomplishing the same tasks with Common Lisp (in the form of the CLISP implementation) and Python.

Common Lisp and Python are good languages to compare AML against, as both features an interpreter, and relies heavily in this feature for development. Common Lisp is also interesting, because it shares the same syntax as AML. Python does not share the syntax, but is also implemented by using the C programming language.

Let's start out by looking at how the two tasks are preformed by the CLISP and Python interpreter.

Depending on the installation and underlying operating system, you start the interpreters in any directory by simply typing `clisp` or `python` into the command line. On Windows systems, you sometimes have to to specify the path to the interpreter executable file in the environment variables to make the executable reachable from any directory. This is dependent on the installer, as some might set this variable as a part of the installation process.

Interacting with the interpreter is then as simple as writing commands in the command line, and executing them with return. Examples of launch and use of Python and CLISP is shown in Listing 1 and Listing 2.

Listing 1: *Use of the Python interpreter*

```
teodoran@HAL:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more
   information.
>>> def greetProfessor(name):
...     print("Good day professor " + name)
...
>>> greetProfessor("Ole-Ivar")
Good day professor Ole-Ivar
```

Listing 2: *Use of the the Clisp interpreter*

```
teodoran@HAL:~$ clisp

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

(Copyright statements omitted)

Type :h and hit Enter for context help.

[1]> (defun greet-teacher-assistant (name)
(format t ``I promise I'll deliver the assignment soon ~s!'' name))
GREET-TEACHER-ASSISTANT
[2]> (greet-teacher-assistant ``Jens'')
I promise I'll deliver the assignment soon ``Jens''!
NIL
```

In the case of a malformed statement, both interpreters return the error message and stack trace directly into the command line. CLISP even supports injection of new code, instead of the erroneous code, through the use of restarts.

To start AML in the console, a minimal environment has to be set up. This is sometimes performed by a included batch script. A version of this script, describing the required setup, is shown in Listing 3

Listing 3: *AML5-startup-win32.bat*

```
ECHO OFF

SETLOCAL

set "AML_TRAINING_ROOT=%CD%\"
set "AML_STARTUP_DIR=%AML_TRAINING_ROOT%AML-Startups\AML-
   Startup-Win32\"
set "AML_DIR=%AML_TRAINING_ROOT%AML\AML5_Win32\"

cd %AML_STARTUP_DIR%
"%AML_DIR%AML.exe" "%AML_DIR%aml.img"

ENDLOCAL
```

The startup script initiates a few variables needed by the AML process. Then it launches AML.exe with the AML.img file.

This gives us an running AML interpreter, that functions much in the same way as the CLISP and Python interpreter. Errors also gives rise to a pop-up dialog, where the user can choose to debug or abort. Choosing the debug option, prints the stack trace to the console.

Using a pop-up dialog to display debug information might have its benefits in a GUI setting, but as a part of a command line interface, it can cause some troubles. If the user where to use the program remotely with a networking interface without graphical forwarding, like ssh without X11 forwarding, the open dialog would cause the AML interpreter to hang indefinitely.

Using a interpreter in this fashion is a quite common occurrence in the general software world. You might have a centralized testing or deployment server set up, where accessing and running code remotely would be beneficial. It also makes batch-scripting with AML more prone to fatal errors, as the script has no way to know if something went wrong with the AML process.

Over to the task of running previously written programs. With CLISP or Python, this is performed by executing the program with the program piped through standard in. This code can be piped directly from a source file, or from the output of another program. Listing 4 and Listing 5 shows how this is done for Python and CLISP respectively.

Listing 4: *Executing Python*

```
teodoran@HAL:~$ python print-ta.py
Teacher assistant
```

Listing 5: *Executing Clisp*

```
teodoran@HAL:~$ clisp print-professor.lisp
"Professor"
```

Running previously written AML-programs is a bit more laborious. You can load and compile systems on startup by editing the logical.pth file or you can load or compile you system with commands directly in the AML interpreter.

The problem with the available options for running previously written programs with AML, is that it makes it harder to automate tasks with batch scripts, since the batch-process cannot interact with the interpreter after it is launched. This makes executing AML programs seemingly unnecessary hard, and during our experimentations, we have not been able to perform this task.

3.1.2 Running AML from another application

One way of handling programming language interoperability, is to let one language execute code in the other, by using subprocesses. A classic example is an application interfacing with a relational database, through execution of SQL-statements. This is by no means the only way to handle programming language interoperability. Another way is using languages that can be compiled into the same executable code, .Net being the famous example of this approach, and it is an approach that makes one language interface with a very different one. In this sections we will investigate the possibilities for controlling the AML framework through subprocesses in another language.

The case used for this investigation will be a REPL application (Read Eval Print Loop), written in Python, that implements the full functionality of the AML interpreter. The user should be able to launch the Python application, and use it as he or she would use the AML interpreter.

This is conceptually a simple problem. You query the user for input, evaluate the input through the AML-subprocess, and then print the result to the command line, before looping the sequence all over again. Pseudo-code for such an application is shown in Listing 6.

Listing 6: *Pseudo-code for a AML REPL application*

```
while true:
    print(evalWithAMLSubProcess(readLine()))
```

Naive translation of Listing 6 into Python code, using the Popen class to handle the subprocess, did not work, and displayed an interesting bug. The program prints out the loading statements, but hangs indefinitely when the user should be prompted for the first input. Why was this happening?

Although we cannot be absolutely certain of what has gone wrong in the previous example, since we do not have access to the AML source code, the problem shows symptoms of a known problem relating unbuffered, line buffered and fully-buffered output. In the draft for the ISO C standard N1124, we can read the following:

> When a stream is unbuffered, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is fully buffered, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is line buffered, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. [18]

At program startup, three text streams are predefined and need not be opened explicitly: standard input (for reading conventional input), standard output (for writing conventional output), and standard error (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

This applies to our problem. It is known that AML is implemented in C, and it is fair to assume that it uses the C standard IO library (stdio) to preform IO operations. This would not be a problem if the standard out stream was unbuffered or line buffered, since the results would have been sent for every character or line. The main problem is that the stdio library only sees that it is connected to a pipe in our program, and will therefore conclude that it is not connected to a interactive device, therefor going into a fully buffered mode. The result is that our Python program waits indefinitely for the result from the AML process.

To test this theory we ran the same example, using Python with an option to force line-buffered IO. This worked as intended.

One way to try to circumvent this problem, is by reading and writing to the AML process in two threads. Listing 7 shows one way op accomplishing this.

Listing 7: *Python AML REPL with two threads*

```
import subprocess
import sys
from threading import Thread

process = subprocess.Popen(['AML5-startup-Win32.bat'],
   shell=False, stdin=subprocess.PIPE, stdout=subprocess.
   PIPE)

def print_stdout():
        while process.poll() is None:
                output = process.stdout.readline()
                sys.stdout.write(output)

t = Thread(target=print_stdout, args=())
t.start()

while True:
        input = sys.stdin.readline()
        process.stdin.write(input)
        process.stdin.flush()
```

This will accomplish our task, but has the unfortunate side effect of having us manage the communication between two different threads.

Another way of accomplishing the task, is to use a pseudo-TTY. This is a program that emulates the command line environment, making the stdio library believe it is communicating with an interactive device. Developing an application that uses a pseudo-TTY requires a lot of development, or the use of a framework for pseudo-TTY work, as this is a quite complex task. Due to the difficulties in finding a suitable pseudo-TTY library for the Windows 8 operating system, the test was performed on Ubuntu 13.04, with CLISP emulating AML, by running in block-buffered mode. This test was successful.

Listing 8: *Python CLISP REPL with pseudo-TTY*

```
import pexpect

c = pexpect.spawn('clisp')
c.expect('\[\d+]>')
print c.before.strip()

def evalArg(arg):
    c.sendline(arg)
    c.expect('\[\d+]>')
    return c.before.replace(arg, '', 1).strip().replace('\
       n', '', 1)

done = False
while not done:
    expression = raw_input('> ')

    if expression == 'exit':
        done = True
        c.sendline('(exit)')
        c.expect(pexpect.EOF)
        print c.before.replace('(exit)', '', 1)
    else:
        print evalArg(expression)
```

The benefit of this approach is the possibility of getting real interaction between the AML process and our own program, but this interaction comes with an sizable overhead and increase of complexity.

3.2 Extending the development environment

3.2.1 The importance of development environment choice

When developing applications in the world of general software development, you have the choice of selecting from a wide array of tools. You'll more often than not start out by determining what kind of programming language, development framework, development methodology, project organization tools, testing framework, continuous integration tool, and so forth, is most suited for the task at hand, and maybe more important, compatible with the knowledge in your team.

An important point to consider for the programmer, is the choice of development environment. Development environment is a term with different definitions, but we will define it as the software-tools used by the programmer to develop the new application, which is not part of the code-base.

Definition 5. *Development environment: The software-tools used by the programmer to develop the new application, which is not part of the code-base.*

Software frameworks is omitted from this definition. Although software frameworks can be considered a tool for the software engineer, it becomes a part of the developed application.

The development environment forms the tools used by the software engineer to perform his or her craft. Important parts of this environment is the code editor, source control tools, project planning tools, and so on. You could even argue that the chosen operating system is a part of the development environment.

An woodworking analogy would be the tool chest. The nail and wood is not part of the development environment, as it is destined to become a part of the final product. The hammer used for hammering the nails, and the saw used for cutting the wood is a part of the development environment, and can differ from tool chest to tool chest.

A motivation for making this division, is that the choice of development environment is often partially determined by the individual programmer, and can often differ within the same development project. These choices is also a factor that'll remain constant for a programmer between projects. An simple example: a programmer can use Emacs to work on both a web-development project in Python and a email-client in C++.

One of the most important tools of the development environment is the code editor. This is the tool that enables the software engineer to write code, and although we can find no statistical study, it is safe to assume that a large amount of the software engineers workday is spent using the code editor to write code. Therefore have we chosen to focus our work on the code-editor part of the AML development environment.

Almost no other tool is at such a whim of the taste of the programmer as the code editing tool. The therm “Editor war” is well known, and so is Richard Stallman humorous “Church of Emacs” [30]. Although there has been made endless non-constructive arguments over which editor or IDE reign supreme, we will not be discussing this. Instead we will put forth some arguments for why editor diversity is beneficial for development in AML.

- Editor choice gives new KBE developers the option of choosing the editor that most closely resembles a editor they are familiar with, in effect reducing the training time.

- A choice of editor, makes AML less dependent on one editor technology. If the dominant editor style where to change in the future, AML would be more ready to support such a change. This is indeed a case of today, where programmers is moving from simple text-editors to integrated development frameworks (IDE's).
- Exposing AML to different development environments gives the AML-developer access to the different features of the different development environments associated with each editor.
- Choice is freedom for the developer. A trait that is usually cherished by today's software developers.

There is an argument to be made against supporting multiple editors for AML, in that it it steals valuable development resource from the core functions of AML. This is a big concern, because a KBE-vendor should be able to focus on delivering the best tools for KBE-development, not code-editing. But is it possible to have the best of both worlds? Can the relationship between the AML-development environment and the code editor be structured in such a way that it is easy to maintain for the vendor, yet enables the developer to use his favorite editor of choice? This is indeed the case with general software development today, where most programming systems can be used with any number of development environments. Our hypothesis will therefore be that this is possible for AML development as well.

RQ 1. *It is possible to structure the relationship between the AML-development system and the code editing tool, in such a way, that it is possible to use other editors than XEmacs, with a minimum of development and maintenance.*

We will also investigate the new features integrating another code editor gives to the AML development environment.

RQ 2. *Can another editor, give access to other development environment features, not readily available in XEmacs.*

We will investigate these questions by trying to integrate the AML development framework with the Sublime Text 2 editor, and we will briefly review the work made by Oluf Tonning [33] on integrating AML with Eclipse and Olivier Doucet [10] on integrating AML into the .Net framework.

3.2.2 Features of an AML editor

Let's start our investigation by looking at the most important features of the current XEmacs integration, and try to organize the features by importance to the programmer. We will then try to divide our findings into features that should be part of the AML integration and features that are related to the editor choice.

AML console: At the center of the XEmacs is the AML console. The console allows the developer to interact with the running AML process through a REPL interface in one of the emacs buffers. The developer is, amongst other, able to compile and run code, interact with the compiled models and send commands to the AML system.

Launch of AML GUI: This can be performed directly in the AML console by using the `(aml)` command, but GUI elements for performing this is also present.

AML Documentation: GUI elements give direct access to the AML manual by opening the manual in the default web browser.

AML syntax highlighting: The XEmacs integration features Lisp-style syntax highlighting, as well as highlighting of some AML-specific expressions.

AML code completion: Code completion can be performed through XEmacs extensive macro system, but no default completions are included. Dynamic abbreviation, completion of words previously written in any open buffer, is supported and can be accessed by pressing M - /.

AML code navigation: The XEmacs integration features ILISP-like code navigation (a well known emacs major mode for editing Lisp, [34], superseded by SLIME, the Superior Lisp Interaction Mode for Emacs). Most notable is the possibility to search for definitions of functions from anywhere in the project.

AML code formatting: Auto-formatting of sections of code in standard Lisp style is supported, although on a side note, lines with lone right parentheses is encouraged in the AML documentation. This is in direct conflict with general Lisp style guides, as documented by both the GNU Emacs Lisp coding conventions, [15] and the Google style guide for Common Lisp, [16].

We can categorize these features into two groups. The first group is the features strictly unique to the AML development framework. The second is features that is more general in nature, and could be implemented by using existing editor functionality or plug-ins from the general software development community.

Editor/IDE	Console support	Lisp support	Configurability	Beginner friendly
Visual Studio	Yes	Low	Medium	Medium
Eclipse	Yes	Medium	Medium	Medium
Emacs	Yes	High	High	Low
Vim	Yes	High	High	Low
Sublime Text	Yes	High	High	High

Table 3.1: *Editor/IDE evaluation*

The AML console, launch of the AML GUI and the AML documentation is features which are unique to AML, but the syntax highlighting, code completion, navigation and formatting are features that closely resembles general editor features for Lisp-like languages. We note that launch of the AML GUI is a sub function of the AML console. If the AML console is implemented, will launch of the AML GUI be possible.

Access to the AML documentation is of also of a lower importance. Opening a local web page is something a developer would be able to do, even if there was no explicit GUI functionality for doing so in the editor. Geir Iversen [19] could also inform us that this is the preferred way to use the AML manual at Aker Solutions KBeDesign.

This implies that a basic integration of AML into any editor consists of two parts:

1. Integrating the AML console into the chosen editor.
2. Configure relevant existing Lisp-editing tools and plug-ins to work with AML.

This strategy already has its merits. Oluf Tonning [33] used this strategy to successfully integrate the AML development framework into the Eclipse IDE. Although this integration never gained major use, due to the quality of the Lisp support in Eclipse [19], it remains an important proof of concept.

Based on the previous discussion, we can evaluate existing code-editing tools for AML integration suitability. We will include categories for console support, Lisp support, as well as general configurability, since this will be important in configuring the editor for a new language. We will also consider user-friendliness for beginner programmers, since "KBE development should not only be for natural born hackers" [26]. The results of this evaluation can be found in Table 3.1.

Based on the evaluation and our familiarity with the editor, we chose to base further work on Sublime Text 2 (Sublime Text version 2).

3.2.3 Configuration of existing Lisp tools in Sublime

The first step was to configure existing Lisp support tools for use with AML. This is done through the existing package support in Sublime. A new folder was made for AML, and the existing configuration for Common Lisp was included. In addition code completion snippets for AML KBE classes was made. This gave the editor satisfactory syntax highlighting and, combined with the default dynamic abbreviation in Sublime, code completion features.

The default editor provided good code navigation by default, except for the possibility of searching for function definitions. Some auto formatting features was also included by default, but these features was found to be insufficient.

Sublime features an extensive library of open source, third party plug-ins. Installation and maintenance of these features is most easily organized through Sublime Package Control [5].

To enhance auto-formatting, the package Sublime Lispindent [14], was installed. Testing revealed this plug-in to give good results by re-using the default configuration for Common Lisp as shown in Listing 9.

Listing 9: *Lispindent config*

```
"aml": {
    "detect": ".*\\.(aml)$",
    "default_indent": "function",
    "regex":        ["(catch|defvar|defclass|
        defconstant|defcustom|defparameter|
        defconst|define-condition|define-modify-macro|",
        "defsetf|defun|defgeneric|define-setf-method|
        define-self-expander|defmacro|defsubst|deftype|
        defmethod|",
        "defpackage|defstruct|dolist|dotimes|lambda|
        let|let\\*|prog1|prog2|unless|when)$"]
}
```

The indentation behavior of Lispindent can also be modified through regex parameter in the config file, although this was never desired in our tests.

In order to enable search for function definitions, the package Find Function Definition [11], was added. This enables the developer to highlight a function name, and search for the corresponding definition in the current project by pressing F8. The plug-in will display a list of files to open if multiple instances of possible definitions is found.

This plug-in also required a minimal setup to work with AML, and the setup can be extended to include other syntactic elements, like classes.

Listing 10: *Find Function Definition Config*

```
{
    "definitions": /* where $NAME$ is the name of the
       function */
    [
        "function $NAME$",
        "$NAME$: function",
        "$NAME$:function",
        "$NAME$ = function",
        "$NAME$= function",
        "$NAME$=function",
        "def $NAME$(",
        "(defun $NAMES$"
    ]
}
```

3.2.4 Integration of the AML console

Based on the discussion under Section 3.1.2, we choose to base our console integration on the multi-threaded approach. The code for this integration is found in Appendix A. A pseudo-TTY approach would be better, but the added complexity would mean that we would not be able to even remotely argue for the integration being easy to maintain and develop.

A approach based on an AML-process in unbuffered or line buffered mode would have been even better, as there exist ready made tools for integrating such interpreters into Sublime [4].

The problems with having to juggle two different threads became apparent when implementing the AMLRepl. In order for the main thread to be able to access the write out from STDOUT, the reading thread had to write the lines into a global variable. The main thread would the try to check at regular interval for new out-print in this global variable, and clearing it after writing the content to the text buffer.

This is not the most elegant solution, and regularly requires the user to force an out-print of the global variable with the command Ctrl + Alt + w.

Sending code to the AML interpreter was, on the other hand, an easy job. The open API of Sublime made it an breeze to get the wanted code from the text buffers.

Some design choices where made in regards to the functionality of the REPL. One was to try to maintain as much of the default editing capabilities of the text-buffer containing AMLRepl. This meant that we would keep the default behavior of the return key. Instead we ended up using `Ctrl + Return` as our default evaluate expression key.

Since the Lisp syntax is so easy to parse, we made the default evaluate expression function to look for the last complete Lisp expression in the current file. This behavior can be overridden, by first marking the desired statements to evaluate. Entire files can also be evaluated by pressing Ctrl + Alt + e.

3.2.5 Additional Configuration

A couple of additional features was added, once the basic editing and REPL functionality was in place. The first one to be added was functions for launching the AML GUI. This was simply done by using the existing functions to make a function that always would send (AML) to the interpreter.

Using the same approach launch of AUnit, the AML unit test framework, was integrated into Sublime. This required a bit more setup, since the logical.pth file has to include the AUnit paths, as shown in Listing 11.

Listing 11: *Aunit logical path*

```
:aunit                    "<full-path-to-sublime-text-folder>
   ...
                        ... \Data\Packages\AMLRepl\AUnit\src\"
:aunit-main-system        :aunit main\
:aunit-core-system        :aunit core\
:aunit-print-system       :aunit print\
:aunit-gui-system         :aunit gui\
```

The AUnit system can then be compiled and launched using the (compile-system ...) and (aunit) functions.

We hoped to be able to display test results from AUnit directly in another text-buffer, as this would have given a more seamless integration. The developer would then have been able to edit, execute and test the AML code, using only the Sublime interface, and the AML GUI where graphical inspection was needed. This should have been possible, due to AUnit being developed with a command line mode [2]. Unfortunately our version of AUnit had a bug in the command line functionality, and coupled with project time constraints, this was never realized.

Finally the different functions of the integrations was made available through menu entries and the command palette, as well as being available directly from shortcuts.

User configurable parameters was bundled together into suitable JSON configuration files, accessible from the settings menu. Readme-files, describing basic use and installation was also added.

3.2.6 Notable differences between Sublime and XEmacs

As stated in the introduction, we will not argue for one editor over another, as we believe that preference of one editor over another is largely due to taste and familiarity. Although the main benefit of integrating AML into Sublime is to give the developer several options to choose from, some notable features of Sublime should be mentioned.

- AML console: At the center of the XEmacs is the AML console. The console allows the developer to interact with the running AML process through a REPL interface in one of the emacs buffers. The developer is, amongst other, able to compile and run code, interact with the compiled models and send commands to the AML system.
- Sublime has a more modern GUI, with a tab system that closely matches modern tabbed applications, like web browsers. This is a interface that younger developers is more familiar with, and might make the editor learning process easier.
- Sublime is customized by using the widely used general programming language Python. This will make customization of the editor easier for developers familiar with Python, as they do not have to learn the lesser known Emacs Lisp language.
- The command palette, gives the developer access to all the options and commands available in Sublime, by utilizing fuzzy search. This makes it easy to find features and commands for the beginner, as well as lesser used features and commands for the experienced user.
- Sublime has a more modern default key-binding, familiar from most other applications. One example is Ctrl-c for copy and Ctrl-v for paste.

- Browsing and editing the project files and folders can easily be done in the left side bar of the editor.
- Sublime has a large and active plug-in community. This gives the developer access to , easily installed to the Sublime package manager.
- The minimap on the right side of the editor, gives the developer an instant overview of the open file. This is especially useful when editing large files.

3.3 Learning material

The learning material is often the first part of a programming system the developer experiences. For complex tools, like AML, it plays a large role in making the tool easy to use.

When debugging an error, you do not have a good reference to the error messages. For most general software development tools, you can simply search for the error text to get a better explanation. This is also something that could be covered in the training documents.

We have, in retrospect, evaluated our learning experience with AML. Technosoft delivers the AML Basic Training Manual [32]. This manual is meant as a complete, beginners course to AML. The manual tries to cover the most important features of AML, mostly by following a case, concerning the aerodynamic surfaces of a missile. In addition, Technosoft delivers AML with a manual [31]. The manual is meant as an documentation of all the features available in AML, and although examples and explanation to the different language constructs are given, it's focus is on being a reference manual, not a training text.

Our experience with the reference manual was that it has two major strong-points. The first is it being a web-page with an exhaustive index. Chapters on different subjects can be opened in different browser tabs, and terms can be searched for by using the browser command find on the index page. The organization of the different terms also feels natural, and makes it easy to find related terms to the terms you already know. The second strong-point is the examples. As well as efficiently displaying the usage of the language term, it is a great starting point for further exploration in the AML interpreter. In our opinion even more examples would have been a useful addition to the reference manual.

The basic training manual left us with the impression that in the effort of trying to cover all aspects of AML, it does not manage to cover each section with enough depth. More intermediate and advances examples are not given not given enough space, and as a result, developing real world KBE-applications is a big step after finishing the manual. This we felt, was especially a problem when covering the topics relating to general use of a programming language with Lisp syntax.

The most dominating programming languages today, Java, C#, C++, and so on, uses C-derived syntax, quite different from the Lisp syntax of AML. Concepts as cons cells as the underlying data structure, functions as data, immutable variables, dynamic typing and macro programming is fundamental to Lisp programming, but does not play a big part in the C-derived languages. To gain a bigger understanding of these concepts, we turned to learning material for Common Lisp.

Another factor in efficient training, is the possibility of getting several different explanations of the same topic. While you might not understand a given language concept with one explanation, several explanations from different point of view might help you to form an understanding. We used the following resources when learning to use AML.

- Land on Lisp by Conrad Barski [3].
- Practical Common Lisp by Peter Seibel [29].
- On Lisp by Paul Graham [17].
- Common Lisp hyperspec [1].

Our experience was that the lessons learned form working with Common Lisp, translated well into working with AML. The basic syntax was very similar, and most basic examples could be compiled as AML. Although small syntactic differences was still an annoyance when they arose.

Looking at the learning material for general programming languages, we can find a lot of interactive tutorials. Some good examples are Codecademy [9], Try Clojure [12] and Learn Knockout.js [21]. Common for all the learning resources is that they bundle code editing, compilation and a step by step tutorial text into the same interface. The user is guided step by step through different tasks, first being explained the concept and then advancing when they supply the correct code to the given problem, or choose to be shown the solution.

Such an interactive learning experience might be interesting to explore with AML, and should be able to implement in both the Sublime and Emacs editor.

The most popular general software programming languages usually have active on-line communities. The communities discuss use of the language, improvements, solutions to common problems and is often a place for beginners to get answers and guidance. Some communities use on-line forums or maintain sites for questions and answers (QA sites). A very successful on-line community for programming language QA is the site Stack Overflow [20]. Technosoft curating such a forum or QA site, might be beneficial.

4 Turning the problem around

There are several ways to integrate general software tools in KBE development. One way, is to make the KBE tools adapt to the rest of software development environment. This way, all ready-made and future-make general software tools, can be used in KBE development without major adapting work. The KBE environment will naturally follow the evolution in software development, and can easily adapt to new changes. The KBE system vendor can focus on their core operation, and not spend time trying to adapt to the rest of the software evolution.

This idea is based one one key assumption, that it is easy to integrate general software tools in a KBE-system that is based on a general programming language. We will formulate this assumption as the following research question:

RQ 3. *Is it easy to integrate general software tools, if a KBE-system is based on a widely used programming language?*

4.1 How to test the hypothesis?

In order to begin to answer this question, we have to overcome one notable obstacle. As of today, there is no KBE-system that follows the general software evolution. The major KBE-systems use proprietary languages like GDL, IDL and AML. These languages are highly adapted for KBE development, but lack the great support that widely used languages have. To investigate this question, we therefore have to make a minimal KBE-system based on a general software language. Only then we can test our hypothesis.

We will therefore make a minimal KBE-system based on Python, Open CASCADE and FreeCAD. Then we will investigate if it is easy to integrate some of the general software tools that Python supports.

4.2 The minimal KBE system

In order to make a minimal KBE-system, we must understand the core features of a KBE system, and how this can be implemented. Rocca et al. investigates this as a part of the paper: "Knowledge based engineering: Between {AI} and CAD. Review of a language based technology to support engineering design'" [26]. Rocca argues that a basic KBE-system contains three essential parts, a viewer, geometry kernel and a programming interface, as shown in figure 4.1.

Definition 6. *A minimal KBE system is build up of a geometry kernel, graphical viewer and a programming interface.*

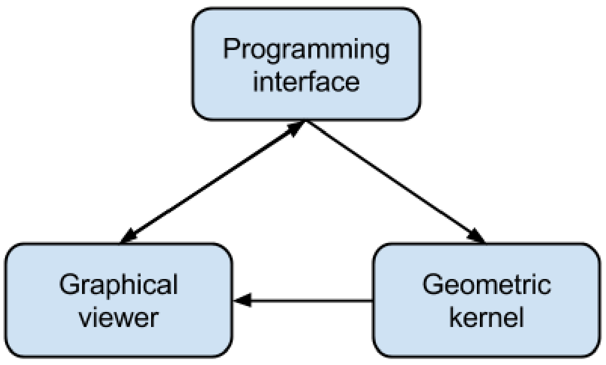


Figure 4.1: *The tree essential parts in a KBE-system and their relation*

The programming interface is what makes a KBE-system special. This is where the rules governing each model is stored. It also describes the object structure needed to create the geometry. The programming interface is build up of a programming language, and must have an interface to the geometric kernel. It must also communicate with the viewer, so the viewer knows about each KBE-object, its parameters and the relation between each object.

The graphical viewer is where the KBE-object and models can be queried and visualized. The are often tree main components.

The tree pane shows the object hierarchy.

The inspector pane is where the properties of a KBE-object can be viewed and manipulated.

The view-port pane is where the KBE-model is visualized.

The geometric kernel is a 3D modeling component, which feed geometric data to the viewer, based on data received by the programming interface module.

The main feature of a KBE system is the programming interface. The programming interface governs the object structure, logic and control that can be implemented in the KBE model. The two other components, the graphical viewer and geometric kernel, is used in many engineering systems today and is not a unique KBE part. Many CAD systems, like NX UGS, FreeCAD and AutoCAD, use the same combination, but the KBE viewer differs in how it visualize a models meta data, and the control components. The viewer system of a KBE system and a CAD system is still comparable.

4.3 Programming interface

To make a minimal KBE-system, we must first choose a suitable programming language. Today almost all KBE languages are object oriented and have some main declaration types, as shown in Table 4.1 from [26]. Possibly the most relevant operator is the `define-class`operator, which declares a class definition. It allows defining classes, superclasses, objects and relationships of inheritance. We can organize the rest of the expressions, as ether attribute function or subclass declarations. The attributes can again be divided in default, descendant, editable or normal attributes declarations. These declarations are similar to the modern object-oriented programming paradigm, which is supported by many general purpose programming languages today. The basic concept for this paradigm, is to represent the consent of objects with attributes and methods and the relationship between these objects [28]s.

4.3.1 Essential features

We see that the basic concepts of a KBE language is supported in the object-oriented paradigm [26] with one exception. The specific attribute declaration, which describe how the attribute should be handled throughout the instantiation and inheritance procedure, is not a standard feature of most general purpose object-oriented languages. This features is handled by native declarations in most KBE languages, but this might not be a necessary feature. Controlling instantiation and inheritance procedure can be handled by creating a basic object structure. The basic object structure can then control relations like superclasses and children in the object hierarchy. It can also make the different types of attributes.

Other essential part of a KBE language, is its ability to give commands to a geometric kernel. To make this possible, an interface have to exist between the language and the geometric kernel.

4.3.2 Non-essential features

Some features make the KBE language more more flexible and controllable, but is not essential. Dynamic typing and multiple inheritance are two examples. Lan-

GDL	**IDL**	**Knowledge Fusion**	**AML**
define-object	defpart	defclass	Define-class
input-slot	input	Any data type a followed by the behavioral flag b parameter (plus optional default value)	
Input-slot:settable (or:defaulting)	Default-inputs		
computed imputs	attributes	Specification of several data types, plus an optional behavioral flag(e.g., lookup, uncached and parameter)	
computed-slot:settable	Modifiable-attributes	A data type followed by the behavioral flag modifiable	
Trickle-down-slots	Descendant-attributes	All attributes descendant by default	
Type	Type	Class	
Objects	Parts	Child	Sub-objects
Hidden-objects	Pseudo-parts	Class name starts with	
Functions	Methods	Functions	

Table 4.1: *KBE language operators to define classes and objects hierarchies*

guages without these features are more strict, and don't support mixins, which, for example, make implementing materials and coordinate systems easier. Knowledge Fusion is one kind of KBE language that don't support these features [26].

Models also tend to be very large when making KBE application, which make calculation and rendering of the model time consuming. This is solved by many KBE-systems today with demand driven evaluation. Demand driven evaluation, also known as lazy evaluation, is an evaluation strategy, which delays the evaluation of an expression until its value is needed. This gives the KBE system the ability to just recalculate and render the parts needed in the model update. There are more solutions to this problem than just a native support for lazy evaluation. One solution is to implement a pattern called observable pattern. This pattern allows a one-to-one or one-to-many dependency between objects, so that when the observed object is changed in some way, all the observers can be notified [27].

Other notable non-essential features, are syntactic overhead and open source. Syntactic overhead is a programming languages ability to eliminate syntactic elements, unnecessary in a given problem domain. The "Hello World" example in Java exemplifies syntactic overhead. The developer has to contain the print statement in an main function and a container class, none of which serves the overarching goal of printing a string to the console. For this problem domain, printing a string to the console, Java would be considered to have high syntactic overhead. This is not always a bad feature, but using a programming languages with low syntactic overhead can mean less writing and more focus on the essential parts of the model. The benefit of open source programming languages, is that widely used open source languages often get a lot of constructive feedback from the open source community regarding bug reports and fixes, support, development of new software tools and libraries, as well as a collaborative culture. This often leads to programming languages with high quality, good documentation and a wide variety of support tools.

Definition 7. *Syntactic overhead is a language ability to eliminate unnecessary syntactic elements for a given problem domain.*

4.3.3 Choosing a programming language

We can review some key features from the discussion above, and try to evaluate what kind of programming languages that are suitable for KBE development. The result of this evaluation is shown in Table 4.2. We have chosen to spit the features in essential and non-essential features. The essential features are that we regard as a "must have" feature, while non-essential features are "nice to have". We also added one other category, which try to describe what kind of general software tools that is supported in the programming language.

From Table 4.2 we can draw some conclusions. First we see that all languages is a suitable KBE language, because they all support the essential features. The specific KBE object structure and hierarchy is something that all the languages can imitate. When we look at the non-essential section, we see that some languages are more suitable than others. Common Lisp, AML and Python stand out as the three most suitable languages. They have all the features that enhance flexibility and control, like demand driven evaluation, dynamic typing and multiple inheritance. They also have a low syntactic overhead. Another ting that differentiates them is that they are open source.

Investigating the feature most relevant to our RQ 3, the GSD tool support, we observe that Python, C++, C# and Java has the most support. All languages have a good and varied selection of compatible general software tools, and this makes them highly selectable for our test. The loser in this category is AML, probably due to this being a lesser used, proprietary programming language. A

Feature	AML	Python	CLisp	C++	C#	Java
Essential features						
Object-oriented	Yes	Yes	Yes	Yes	Yes	Yes
Interface to geometric kernel	Yes	Yes	Yes	Yes	Yes	Yes
Non-essential features						
Native demand driven evaluation	Yes	Yes	Yes	No	Yes	No
Dynamic typing	Yes	Yes	Yes	No	No	No
Multiple inheritance	Yes	Yes	Yes	No	No	No
Open source	N	Yes	Yes	Yes	No	Yes
Syntactic overhead	Low	Low	Low	High	Medium	Medium
General software tools						
Debugging tool	No	Yes	Yes	Yes	Yes	Yes
Test framework support	Low	High	Medium	High	High	High
IDE support	Low	High	Medium	High	High	High
Community	Low	High	Medium	High	High	High

Table 4.2: *A comparison table between KBE language features and some programming languages.*

honorable mention is the AUnit test framework for AML, but this tool is still in development.

Based on this evaluation we choose to use Python as our base language, because it has all the features needed to become a KBE language, and has good support for general software tools, which is crucial in order to explore our research question.

4.4 Geometric kernel and graphical viewer

A programming language alone does not make a KBE system. We still need to find a geographical viewer and a geometric kernel. Available modeling kernels compatible with Python are:

- **Parasolid** by ShapeData, now owned by Siemens.
- **ACIS** by Spatial Corporation.
- **ShapeManager** by AutoDesk.
- **RGK** by Russian Federation's Ministry of Industry and Trade.
- **Open CASCADE** - by open source community.

The two most used kernels are Parasolid and ACIS, which are use by NX, AML, AutoCAD and SpaceClaim. RGK is a newly developed kernel, with a lot of positive feedback [22], but is still in development. The only major geometric kernel that is both proven in other applications and based on open source, is Open CASCADE and is therefore the only one economic scope of this project. It includes C++ components for 3D surface and solid modeling, visualization and data exchange.

The geometric kernel and graphical viewer has to be connected to each other and the programming language. This give some restriction in choosing the viewer component of our KBE system, since we have chosen Python as the base programming language. We are also restricted regard to development time, so a ready-make viewer and geometric kernel combination is preferable. Some open source applications satisfies our requirements:

- The first option is PythonOCC [24]. PythonOCC is an 3D CAD/CAE/-PLM development framework for Python. There is no GUI component, just a Python interface to the underlying OpenCascade kernel and libraries to prototype a viewer.
- The second option is FreeCAD [13] and its Python objects called PytonFeatures. PythonFeatures are 100% Python objects, and contains a visualization and geometry part. FreeCAD is a parametric 3D modeler. This makes it more similar to traditional CAD applications than PythonOCC. FreeCAD includes a built-in Python interpreter and an open API, that covers almost all features of the application. With this scripting feature, a developer is able to create an object structure, on top of the viewer and kernel, to create a KBE model.

Both these alternatives are able to give us the base of our testing platform. Both have a viewer and a geometric kernel, that is controlled by a Python API. All that is left, in order to create a KBE-model, is to create the programmable interface. Out of these two alternatives we choose FreeCAD. The main reason for this was development time, and since we only need a minimal KBE system to test our resource question (RQ 3), FreeCAD was the easiest application to use.

4.5 Creating the testing platform

Now we have the basic tools to create our testing platform. Before we start implementing, we need to set some restrictions on our minimal KBE system. There is no need to create a complete system to test our hypothesis, just the basic functionality.

- Performance is not an issue, because we only need to make small models. Therefore we do not need to think about demand driven evaluation or other performance enhancing features.
- Only focus on geometry, not meshing or FEM analysis.
- Create a basic object structure, that support multiple inheritance and children.

4.5.1 The object structure

To control instantiation and relationship between each object, we have created a superclass, called KbePart (see Appendix B.1). This object has to be inherited from all KBE classes. It contains a set of functions that can be used by the inherited classes, as described below.

make_properties() can be extended to add properties to the model.

make_property(name, property_type) can be used to make a editable and descendant property. It returns a object with all editable and descendant properties.

add_children() is used to add and manipulate children.

make_child() is used to add a child.

get_properties_list() return a list of all editable and descendant properties.

print_msg() can be used to print messages to the viewer's console pane.

4.5.2 The test model

Based on this object structure, we have created a simple missile model (see Appendix B.2), based on the case from the AML Basic Training Manual [32]. This model is going to be used to test-environment for our hypothesis. The missile is build out of objects inherited from the KbePart. From figure 4.2, we can see that the missile class is based on a set of other subclasses, missile_nose, missile_aft_body and rotation_fins.

From figure 4.3, we can see the graphical viewer with the three pane, editable properties pane and view-port pane. From the properties pane, we can see that all the properties have propagated from the child objects to the missile. The properties can also be edited to change the geometry of the missile.

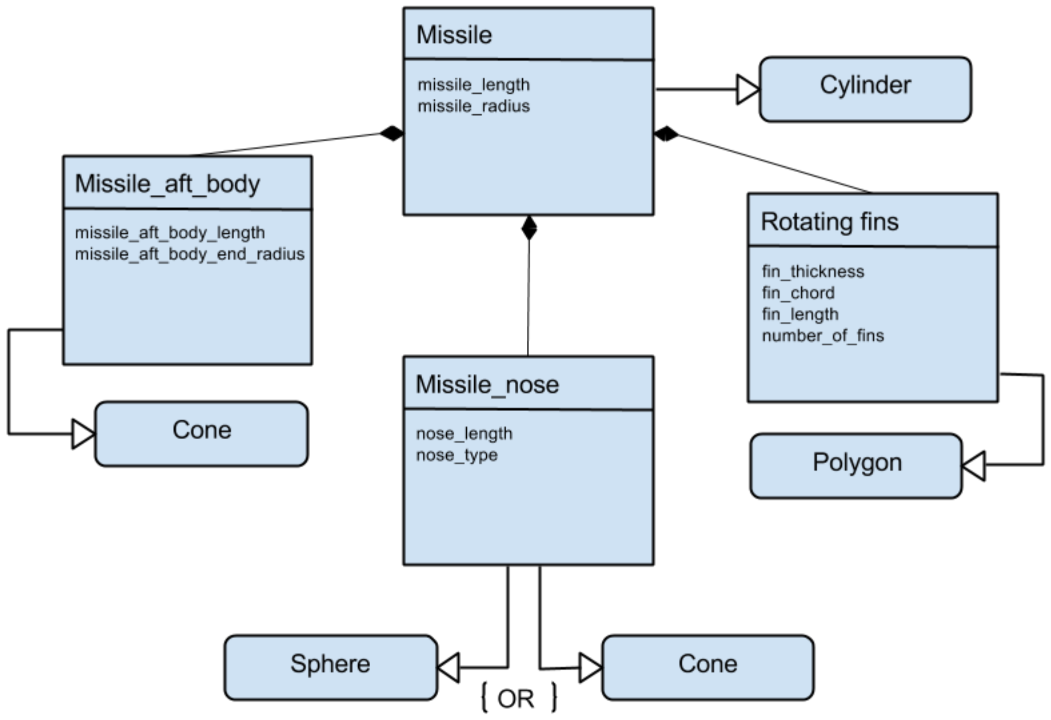


Figure 4.2: *UML Class diagram for the Missile class showing inheritance and composition links.*

s

4.6 Integrating general software tool

Now that we have a test platform we can start integrating some general software tools. Three GSD tools, that are going to be investigated in particulare is debugging tools, tools for testing and IDE/editor support.

4.6.1 Interfacing with FreeCAD and Open CASCADE

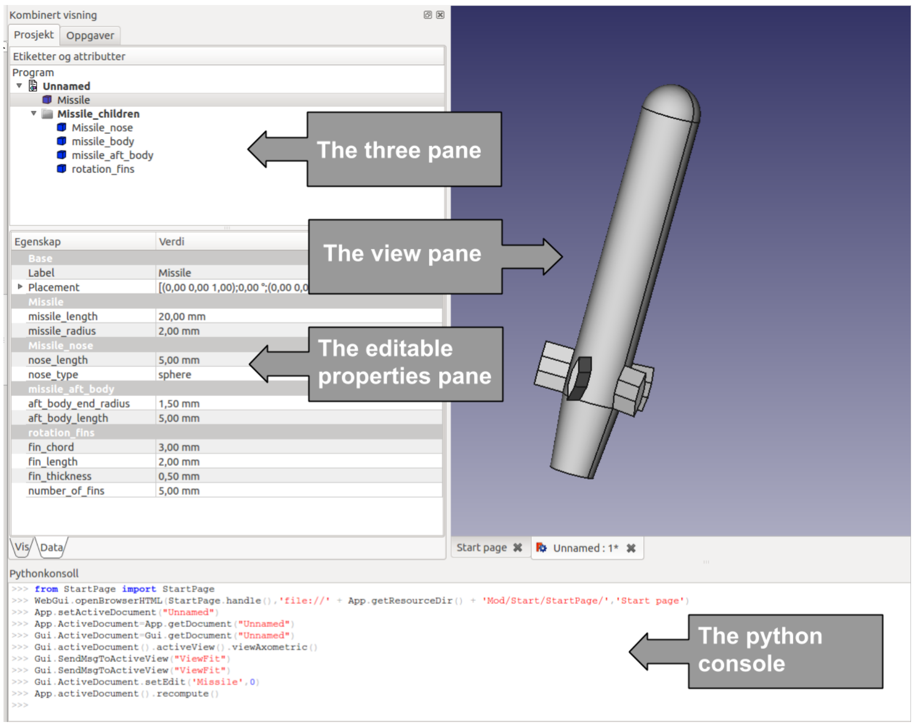
To get Pythons general software tool to work in a FreeCAD environment, an interface need to exist between them. This interface is the Python programming language. All FreeCAD features, like modeling, meshing and simulations, are accessible thought a Python API. This makes it easy to integrate different kinds of tools. FreeCAD is imported into Python as a normal module. The process for doing this in a temporary way is shown below:

```
import sys sys.path.append("path/to/FreeCAD/lib")
```

FreeCAD can now run inside other applications that use Python or from an external Python shell. We can also import our KbePart the same way:

```
import KbePart
```

This makes our minimal KBE system very modular.



Egenskap	Verdi
Base	
Label	Missile
Placement	[(0,00 0,00 1,00);0,00 °;(0,00 0,0
Missile	
missile_length	20,00 mm
missile_radius	2,00 mm
Missile_nose	
nose_length	5,00 mm
nose_type	sphere
missile_aft_body	
aft_body_end_radius	1,50 mm
aft_body_length	5,00 mm
rotation_fins	
fin_chord	3,00 mm
fin_length	2,00 mm
fin_thickness	0,50 mm
number_of_fins	5,00 mm

```
>>> from StartPage import StartPage
>>> WebGui.openBrowserHTML(StartPage.handle(),'file://' + App.getResourceDir() + 'Mod/Start/StartPage/','Start page')
>>> App.setActiveDocument("Unnamed")
>>> App.ActiveDocument=App.getDocument("Unnamed")
>>> Gui.ActiveDocument=Gui.getDocument("Unnamed")
>>> Gui.activeDocument().activeView().viewAxometric()
>>> Gui.SendMsgToActiveView("ViewFit")
>>> Gui.SendMsgToActiveView("ViewFit")
>>> Gui.ActiveDocument.setEdit('Missile',0)
>>> App.activeDocument().recompute()
>>>
```

Figure 4.3: *The freeCAD graphical viewer visualizing the missile, with the three pane, editable properties pane, view-port pane and a Python console*

4.6.2 Testing support

Testing is the practice of making objective judgments regarding the system's quality and correctness. It is a huge help for maintaining and developing a code base [25]. Testing big and complex structures, like KBE models, are not a easy task. Many problems tend to arise, because the different objects in the system have many complex dependencies between them [2]. In the software industry this is solved with good testing frameworks and mocking. Mocking is done my creating mock objects, which are injected to isolate different objects from each other during testing. Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. By using mock objects, we can assure that errors in one object does not affect the test results for another object. This makes locating errors and writing tests easier on complex systems, like KBE models.

Integrating a test framework in a KBE system, means that we have to isolate the models and their logic. In our case this implies we have to isolate the KbePart objects. They alone, hold all logic and rules governing the model and their geometry. With KbePart being a complete Python module, it is easy to integrate any test framework supported by the Python programming language.

Python have a large variety of test frameworks to choose from. Some varieties are unittest, pyUnit, py.test or doctest. They all have positive and negative qualities. We have chosen unittest, as this is the currently most used testing framework for Python.

Integrating the unittest framework was done in a normal manner. We imported the module and created a test case class.

Listing 12: *Creating a TestCase*

```
import unittest
from mock import Mock

class DescribeMissile(unittest.TestCase):

  def setUp(self):
      self.missile_doc = DocumentMock()
      self.missile = Missile(self.missile_doc)
      self.fp = FeaturePythonMock()
```

The *setUp* function is called before each test is executed. We make the missile from independent from its dependencies, by creating two different mock objects, as we can see from the code snippet above and in Appendix B.3. The document mock simulates the freeCAD document where every model is added. The FeaturePythonMock is an object that represent all properties, so we can control them in the test situation. Every KbePart have control of its own objects, making it easy to abstract the parts currently being tested. Now we can start to make some tests. The first test, shown in Listing 13, is checking if the missile has four children.

Listing 13: *Missile test 1*

```
def test_missile_should_be_init_with_4_children(self):
     children_length = len(self.missile.children)
     self.assertEqual(children_length, 4)
```

We are also able to mock out the objects/attributes/functions we want. Listing 14 shows how we have mocked out the missile's execute function.

Listing 14: *Missile test 2*

```
def test_missile_should_update_when_a_value_is_changed
   (self):
    self.fp.missile_radius = 5
    self.missile.execute = Mock(return_value=True)
    self.missile.onChanged(self.fp, 'missile_radius')
    self.assertTrue(self.missile.execute.called)
```

4.6.3 Debugging support

A lot of different debugging tools are available for the Python programming language. Some examples are WinPDB, PyDebug which are embedded in many Python IDEs. The most basic debugger is the Python debugger, which is available as a Python module called pdb. At the core this is a command line debugger, but there exists graphical interface that work with it. It also integrates with the Sublime Text 2 editor. It is integrated just by importing the pdb module. A breakpoint as created by inserting a *pdb.set_trace()*. When the code is executed, the developer is free to access all the debugger features through the command palette in Sublime, see Figure 4.4.

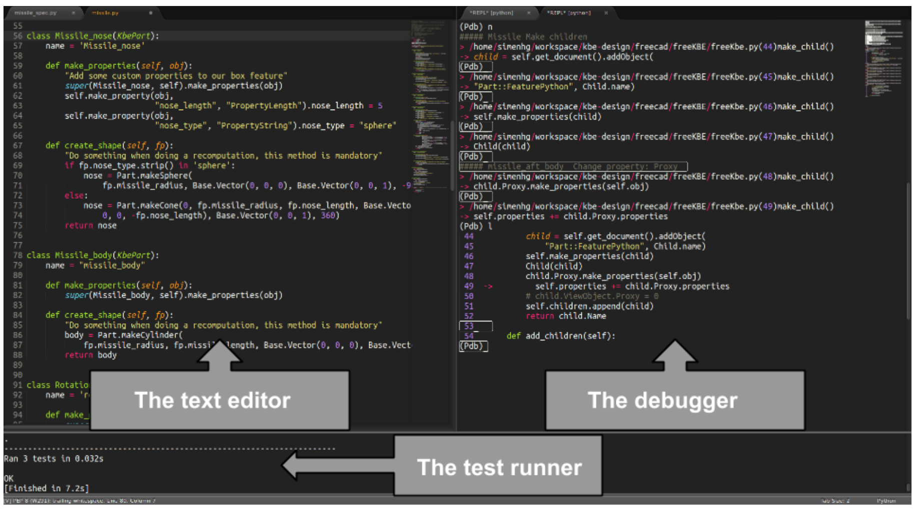
4.6.4 IDE support

Many IDE options exist for Python. Some of the most acclaimed are PyCarm, Eclipse with PyDev, Wing and Sublime Text. We chose to integrate our it in sublime text, since it is our favorite editor, has a great variety of features and makes a nice comparison to our Sublime AML integration.

To better configure Sublime to Python editing, the anaconda [6] plug-in should be used. Anaconda adds a wide spectrum of facilities, like auto-complete, goto definition, goto documentation, refactoring, linting and much more. It also adds a lot of handful snippets.

To integrate Python debugger, the plug-in SublimeREPL was used. SublimeREPL is able to run a Python interpretor in a Sublime buffer. SublimeREPL has a command to run current file in pdb-mode, which is useful for debugging.

Sublime supports unit-testing out of the box. *Ctrl+b*, compiles and runs the current test file. A little buffer at the bottom of the editor will pop up and display the test results, as shown in Figure 4.4.



```
class Missile_nose(KbePart):
    name = 'Missile_nose'

    def make_properties(self, obj):
        "Add some custom properties to our box feature"
        super(Missile_nose, self).make_properties(obj)
        self.make_property(obj,
                           "nose_length", "PropertyLength").nose_length = 5
        self.make_property(obj,
                           "nose_type", "PropertyString").nose_type = "sphere"

    def create_shape(self, fp):
        "Do something when doing a recomputation, this method is mandatory"
        if fp.nose_type.strip() in 'sphere':
            nose = Part.makeSphere(
                fp.missile_radius, Base.Vector(0, 0, 0), Base.Vector(0, 0, 1), -9
        else:
            nose = Part.makeCone(0, fp.missile_radius, fp.nose_length, Base.Vecto
                0, 0, -fp.nose_length), Base.Vector(0, 0, 1), 360)
        return nose


class Missile_body(KbePart):
    name = "missile_body"

    def make_properties(self, obj):
        super(Missile_body, self).make_properties(obj)

    def create_shape(self, fp):
        "Do something when doing a recomputation, this method is mandatory"
        body = Part.makeCylinder(
            fp.missile_radius, fp.missil length, Base.Vector(0, 0, 0), Base.Vect
        return body


class Rotatio
    name = 'r

    def make_
```

```
(Pdb) n
##### Missile Make children
> /home/simenhg/workspace/kbe-design/freecad/freeKBE/freeKbe.py(44)make_child()
-> child = self.get_document().addObject(
(Pdb)
> /home/simenhg/workspace/kbe-design/freecad/freeKBE/freeKbe.py(45)make_child()
-> "Part::FeaturePython", Child.name)
(Pdb)
> /home/simenhg/workspace/kbe-design/freecad/freeKBE/freeKbe.py(46)make_child()
-> self.make_properties(child)
(Pdb)
> /home/simenhg/workspace/kbe-design/freecad/freeKBE/freeKbe.py(47)make_child()
-> Child(child)
(Pdb)
##### missile_aft_body  Change property: Proxy
> /home/simenhg/workspace/kbe-design/freecad/freeKBE/freeKbe.py(48)make_child()
-> child.Proxy.make_properties(self.obj)
(Pdb)
> /home/simenhg/workspace/kbe-design/freecad/freeKBE/freeKbe.py(49)make_child()
-> self.properties += child.Proxy.properties
(Pdb) l
 44             child = self.get_document().addObject(
 45                 "Part::FeaturePython", Child.name)
 46             self.make_properties(child)
 47             Child(child)
 48             child.Proxy.make_properties(self.obj)
 49  ->           self.properties += child.Proxy.properties
 50             # child.ViewObject.Proxy = 0
 51             self.children.append(child)
 52             return child.Name
 53
 54         def add_children(self):
(Pdb)
```

```
.
----------------------------------------------------------------------
Ran 3 tests in 0.032s

OK
[Finished in 7.2s]
```

Figure 4.4: *Sublime text integration, displaying the test runner, the debugger and the text editor.*

4.7 Problems with switching KBE-system

In this chapter we have created a minimal KBE system and integrated a set of general software tools. In our tests, no major obstacles have been encountered. However, this is not conclusive evidence that creating a new KBE-system is an easy task.

Despite the fact that creating a new KBE System seems like a easy solution, some major problems has to be solved. One problems is continuation of knowledge. Today all knowledge and logic connected to a KBE model is interleaved in the KBE classes. This means that all knowledge is lost if one decide to switch KBE system. The knowledge can, however, be transferred to a new KBE system. It can for example be done by parsing the objects to the new system. It should be said that this is not an easy task and should be examined before attempting to create a new KBE system.

Time and quality is also factors we need to take into account. Although it is easy to integrate general software tools in a new KBE system, it may not be the best solution. The time it takes to create a good KBE system may exceed the time to integrate general software tool. The quality of a new system may also not be as good as a old and fully tested system.

5 Results and discussion

5.1 Interfacing with AML, developing a new editor and exploring new learning possibilities

Section 3.1, contained an exploration of the possibilities for interfacing with AML from the system console and from another programming language. The console interface of AML was initially hard to work with, compared to the general programming languages Python and Common Lisp. The reason was mostly because the lack of documentation and the slightly laborious setup needed to launch the AML process in the console. However, improving the documentation, could make the difference negligible.

Then the possibility of controlling AML from another programming language was addressed. Several attempts at making a prototype REPL for AML in Python was made, however only two of the attempts where successful. Reading and writing to the AML process in different threads and using a pseudo-TTY. This was doe to AML utilizing block-buffered IO when being piped to another process. An argument was made, that this behavior is caused by the way the C standard IO library behaves.

This behavior made interfacing with AML from other programming languages unnecessary hard, and it was reviled that other programming languages avoids the problem, by giving the user the option of forcing line-buffered or unbuffered IO. Making such options part of AML would be a valuable feature.

In section 3.2, the RQ 1 was studied by making a prototype integration of AML into the Sublime Text 2 editor. Using the strategy of configuring the editors existing Common Lisp support to work with AML, and then integrating the AML console into the editor, gave a working prototype. This prototype was further enhanced by some general configuration of Sublime, in the end, managing to duplicate the most important features of the AML XEmacs integration.

The main drawback of the prototype AML integration, was the problems related to controlling the AML process from Python, as described in section 3.1. If AML had the option of line-buffered or unbuffered IO, the entire integration could have been made using configuration of existing features and plug-ins of Sublime.

Even with the problems related to integrating the AML console, development of the Sublime integration was fairly straight forward. With the prototype integration only a minimum development and maintenance is needed, although both the quality and maintainability of the console integration is diminished to some degree.

The next question, RQ 2, was not covered in as great depth as the first one, mainly due to the difficulty of integrating the AML console. An argument was made for the value of choice between editors, something that in a sense can be understood as a feature not readily available in XEmacs. A brief overview of notable features in Sublime was supplied. Although some of these features can be duplicated in XEmacs, XEmacs being a highly configurable editor, the focus was to investigate the features readily available in the editor. With this premise one can argue that a Sublime AML integration offers other features than XEmacs.

In the section 3.3, the learning challenges of AML where addressed, and the learning material of AML was discussed. Some shortcomings of the current learning material for AML was addressed, and the option of using learning material designed for Common Lisp was explored with good results. Finally the benefits of having an active on-line community during the learning process was also discussed.

This section leads to the question if the syntax of AML should be made more closely like, or even compatible with, the syntax of ANSI compliant Common Lisp. This would give the developer even more options to use previously written general software tools together with AML. Our learning experience with AML indicated that even with the existing, non ANSI compliant syntax, AML training can be improved by using the learning material and community support for Common Lisp. By making the syntax ANSI compliant, further gains from this strategy is expected, and this could give the developer the option of using existing software libraries and tools written for Common Lisp.

The trade-of of altering the syntax in such a way, is compatibility with previous versions of AML, as well as the more substantial resources needed to modify the AML modeling system. These would be needed to be throughly investigated in a more if such an undertaking was to be considered, but the possible gains of making this change makes us recommend that this should be studied in further work.

5.2 Building KBE models with Python and Open CASCADE

In the introduction, another way of integrating GSD tools was introduced and the RQ 3 was submitted.

The anatomy of a KBE system and different options for replacing the individual components was discussed, before settling with Python as the programming language, and Open CASCADE and FreeCAD for the geometric kernel and graphical viewer.

A test model of the missile system, described in the AML basic training manual, was developed. To aid with developing the model, ready made solutions for IDE, testing and debugging support was utilized. Compared to the troubles faced when developing the Sublime AML integration, this indicates that integrating general software tools is easier when the KBE-system is based on a widely used programming language.

More surprisingly, using this system to develop KBE models, was not as complicated as initially estimated, much due to the large amount of available open source tools and libraries. Notable drawbacks of using such a system was also identified in Section 4.7.

6 Conclusion

The success of extending AML with parts from general software tools was in our cases largely dependent on the interfacing options to access AML from other programming languages. Documenting the console interface and allowing to force unbuffered or line buffered IO would have made the task of integrating AML into Sublime a trivial task, well within the grasp of a beginning KBE developer with some programming experience. It is our belief that this will enable developers to integrate AML into their favorite development tools, diversifying and expanding the AML system.

Documenting the console interface and allowing to force unbuffered or line buffered IO, seems to be possible to implement in AML without huge alterations to the existing framework.

Further modifications could be made to give AML a ANSI compatible syntax, giving AML developers even greater gain from using Common Lisp learning material as well as access to previously written libraries and tools for the more widely used Common Lisp language. If this where to be considered as an option, further investigation should be preformed, due to the possible complexities related to such modifications. Either way, resources used by Thecnosoft on developing learning material, might be better used on expanding the material covering KBE-specific parts of AML, and use learning material for Common Lisp to teach the basics of Lisp languages.

By experimenting with building KBE models with a widely used programming language, we instantly got access to testing, debugging and IDE/editor support. This we feel, demonstrates the advantages of having a KBE system that is compatible with general software tools. The obstacles of using such an approach is also apparent. Development time, quality, knowledge transfer from old systems, all makes this a more substantial undertaking than one might believe at first glance.

The ease of building KBE models with existing general software tools, should be seen as an incentive for Technosoft to push for better access to AML from other programming languages. Using a widely used general programming language at the core of a KBE model, instantly gives the developer access to a large amount of existing development tools, libraries and training material. Although development of KBE models with existing general software tools is not an viable alternative to AML today, this might change in the future. In such a scenario, changes would be needed to make AML an competitive option for the KBE developer.

To conclude this report, the authors would like to inform the reader that all code written as part of this project is freely available for use, re-use and sharing at `https://github.com/teodoran/TDT-4560`. An video-tutorial AMLRepl is available at `https://www.dropbox.com/s/5n2ksllro5kj72o/aml-repl-tutorial.mpg`.

References

[1] Common lisp hyperspec.

[2] Lars Barlindhaug. DEVELOPING SOFTWARE QUALITY IN KBE IMPLEMENTATIONS - AUnit. November 2011.

[3] Conrad Barski. *Land of Lisp - Learn to Program in Lisp, One Game at a Time!* No Starch Press, 2011.

[4] Wojciech Bederski. SublimeREPL. (available online at `https://github.com/wuub/SublimeREPL`), 2013.

[5] Will Bond. Sublime Package Control. (available online at `https://sublime.wbond.net/browse`), 2013.

[6] Oscar Campos. Anaconda. (available online at `https://github.com/DamnWidget/anaconda`), 2013.

[7] Craig B. Chapman and Martyn Pinfold. The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure. *Advances in Engineering Software*, 32(12):903 – 912, 2001.

[8] George Chlapoutakis. CASE Tools Versus Modern Software Development Tools: A critical analysis of the changes in Software Development Tools between 1990 and 2005. 2013.

[9] Codecademy. Codecademy. (available online at `http://www.codecademy.com/`), 2013.

[10] Olivier Doucet. private communication, 2013.

[11] Tim Douglas. Find Function Definition. (available online at `https://github.com/timdouglas/sublime-find-function-definition`), 2013.

[12] Anthony Grimes et al. Try Clojure. (available online at `http://tryclj.com/`), 2012.

[13] FreeCAD. FreeCAD. (available on github `https://github.com/FreeCAD/FreeCAD_sf_master`), 2002.

[14] Jonathan Fischer Friberg. Sublime Lispindent. (available online at `https://github.com/odyssomay/sublime-lispindent`), 2013.

[15] GNU. GNU Emacs Lisp coding conventions. (available online at http://www.gnu.org/software/emacs/manual/html_node/elisp/Coding-Conventions.html), 2013.

[16] Google. Google style guide for Common Lisp. (available online at http://google-styleguide.googlecode.com/svn/trunk/lispguide.xml?showone=Horizontal_white_space#Horizontal_white_space), 2013.

[17] Paul Graham. *On Lisp.* Prentice Hall, Englewood Cliffs, 1994.

[18] ISO/IEC. Draft of the ISO C standard N1124. (available online at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf), 2005.

[19] Geir Iversen. private communication, 2013.

[20] Joel Spolsky Jeff Atwood. Stack Overflow. (available online at http://stackoverflow.com/), 2008.

[21] knockoutjs. Learn Knockout.js. (available online at http://learn.knockoutjs.com/#/?tutorial=intro), 2013.

[22] Dmitry Semin Nikolay Snytnikov Leonid Baranov, Sergey Kozlov. Russian 3D-kernel RGK: Functionality, Advantages, and Integration. (available online at http://isicad.net/articles.php?article_num=16135), 2013.

[23] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[24] Thomas Paviot. PythonOCC. (available on github https://github.com/tpaviot/pythonocc), 2013.

[25] Jonathan Rasmusson. *The Agile Samurai: How Agile Masters Deliver Great Software.* Pragmatic Bookshelf, 1st edition, 2010.

[26] Gianfranco La Rocca. Knowledge based engineering: Between {AI} and cad. review of a language based technology to support engineering design. *Advanced Engineering Informatics*, 26(2):159 – 179, 2012. ¡ce:title¿Knowledge based engineering to support complex product design¡/ce:title¿.

[27] Andrew Rollings and Dave Morris. *Game Architecture and Design: A New Edition.* New Riders Games, 2003.

[28] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Cambridge, MA, USA, 2004.

[29] Peter Seibel. *Practical Common Lisp.* APress, 2004.

[30] Richard Stallman. St IGNUcius. (available online at `https://stallman.org/saint.html`), 2000.

[31] TechnoSoft. Adaptive Modelling Language, 2013.

[32] TechnoSoft. AML Basic Training Manual, 2013.

[33] Oluf Tonning. private communication, 2013.

[34] The Common Lisp wiki. ILISP. (available online at `http://www.cliki.net/ilisp`), 2013.

Appendix A AMLRepl

A.1 AMLRepl.py

A.2 AMLRepl.sublime-settings

A.3 Default.sublime-settings

A.4 Main.sublime-menu

A.5 Default.sublime-commands

A.6 Sublime text snippets for AML

apropos.sublime-snippet defconstant.sublime-snippet define-class.sublime-snippet defmacro.sublime-snippet defparameter.sublime-snippet defun.sublime-snippet defvar.sublime-snippet if.sublime-snippet let.sublime-snippet let1.sublime-snippet loop-from-to.sublime-snippet setf.sublime-snippet

Appendix B Python and Open CASCADE

B.1 KbePart.py

B.2 missile.py

B.3 missile_spec.py

Appendix C Risk analysis

THE NORWEGIAN UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF ENGINEERING DESIGN
AND MATERIALS

PROJECT WORK AUTUMN 2013
FOR
STUD.TECHN. TEODOR ANDRE ELSTAD AND
SIMEN HAUGERUD GRANLUND

INTEGRATING GENERAL-PURPOSE SOFTWARE DESIGN TOOLS INTO KBE DEVELOPMENT
Integrere generiske verktøyer for programvareutvikling i KBE utvikling.

Most software tools for KBE development is today based around special purpose Modeling frameworks like AML. While this approach has been very successful, there is still room for improvement. This project aims at investigating the possible benefits of utilizing generalpurpose software development tools as a part of KBE development. The following areas of interest will be studied, with KBE development at Aker Solution KBeDesign serving as a case study. Integration of general purpose software development tools into the AML modeling framework. Usage of general purpose software development tools for KBE development alongside the AML modeling framework. Usage of general purpose Software development as standalone tools for KBEdevelopment.

The assignment includes:

1. A literature study of software development tools and KBE technology.

2. A valuation of generic software development tools with respect to KBE development

3. Specification of a set of software tools well suited for implementing KBE applications

4. Evaluate how the specified tools could support implementation in the AML framework

5. If time allow, develop an AML prototype using the specified tools.

Students are required to submit an A3 page describing the planned work three weeks after project start both as a paper version and as a pdf-file. A template for this sheet is found on IPM's web-page, using the link http://www.ntnu.no/ipm/prosjekt.

Performing a risk assessment of the planned work is obligatory. Known main activities must be risk assessed before they start, and the form must be handed in within 3 weeks of receiving the problem text. The form must be signed by your supervisor. All projects are to be assessed, even theoretical and virtual. Risk assessment is a running activity, and must be carried out before starting any activity that might lead to injury to humans or damage to