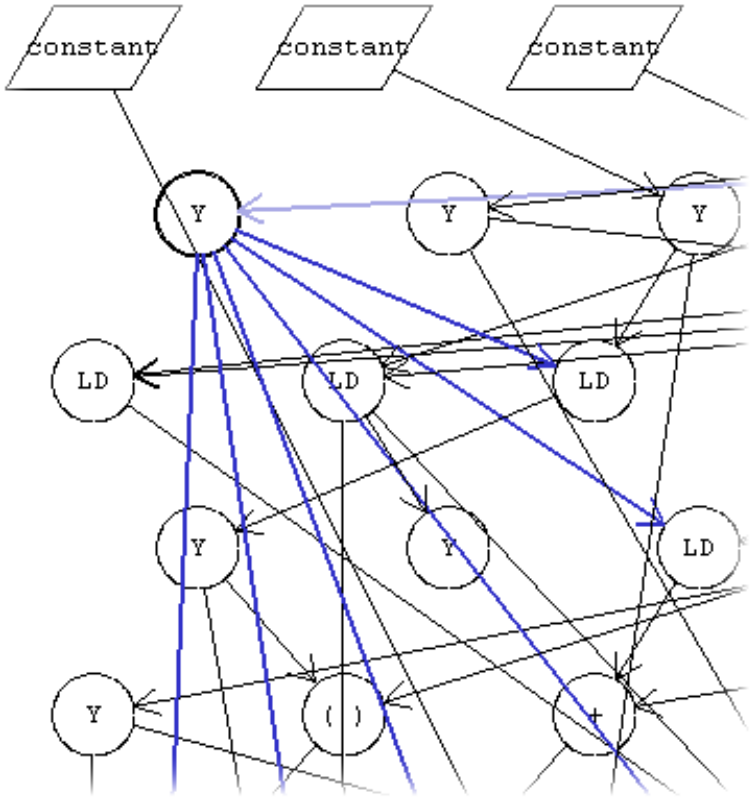
TU Delft

B.Sc. Thesis

Electrical Engineering

Development and evaluation of the HDFG Editor

An application for graph visualization and modification



Jolle Lont Maurijn Neumann

Faculty of Electrical Engineering, Mathematics and Computer Science

Preface

Jolle Lont and Maurijn Neumann are two Bachelor students Electrical Engineering at Delft, University of Technology. Currently, both of them are completing their third year. With an interest in computer science and engineering they extended their curriculum with a minor in computer science.

In the context of the course High Tech Startups a business plan was written. This business plan describes how to prepare the Delft WorkBench, a software tool chain for automatic conversion of software to hardware, for market release. Subsequently, in the context of the Final Bachelor Project a tool for the Delft WorkBench was created. This document describes the design and evaluation processes, as well as the functionality of this application.

Readers interested in reconfigurable computing, the Delft WorkBench and scheduling are referred to Chapter 3 on related research. Those with an interest in the design of the *HDFG Editor* are pointed to Chapter 2 and Chapters 4 to 7. Appendix B elaborates on the repository and readers with the desire to modify this file are referred here. The attached CD-ROM, Appendix F, contains the full source code of the application, as well as a compiled executable for Microsoft Windows.

The authors would like to express gratitude toward the following persons:

- Koen Bertels (TU Delft), for his support and enthusiasm;
- Yana Yankova (TU Delft), for her assistance and guidance throughout the project;
- Josje Kuenen (TU Delft), for reading and correcting this thesis;
- Michel Vos, for proofreading and assisting in the evaluation;
- Everyone else who has assisted or otherwise supported us.

Summary

Reconfigurable computing is a technique used to speed up computer programs by using a combination of hardware and software implementation. The Delft workbench is a tool chain for reconfigurable computing, making it possible to design software without knowledge of the underlying hardware. One function of this tool chain is to convert source code to a hardware description. The DWARV toolset performs this function. While doing so, it uses Hierarchical Data Flow Graphs (HDGFs), which contain data dependencies in the original source code.

HDGFs only exist in a binary format and there is no easy way to modify them.

This thesis describes the design process of a program, the *HDFG Editor*. This process consists of a few steps. First, the functional and non-functional requirements were determined. To prevent improper interpretation of the requirements, a set of use cases was created and prioritized. Next, an analysis of available applications and libraries took place. A concept user interface was also designed to get an idea of the visual outline of the application at an early stage.

With the decision of Irrlicht as a graphics engine the implementation could commence. Implementation of an application entails creating algorithms and analyzing code. With adaptability as a main focus point some trade-offs were made concerning run-time as scripts are sometimes preferred to code.

Concluding, the running time for the project proved to be insufficient to finish the *HDFG Editor*. Part of the functionality has been implemented but not enough to put to the program to use. However, the work done provides a solid base should the project be picked up in the future.

Table of Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Reconfigurable computing is an increasingly popular way to obtain the speed provided by hardware and still benefit from the flexibility of software. The Delft Workbench tool chain aims to make the reconfigurable hardware environment accessible to those without knowledge of hardware. In order to acquire a reconfigurable hardware implementation of software kernels, the source code must be translated to a Hardware Description Language (HDL). The DWARV toolset automates this process by translating C source code to VHDL. DWARV creates Hierarchical Data Flow Graph (HDFG) representations as intermediate format when translating the code. These HDFGs represent data dependencies within C source code.

HDFGs only exist as a binary format and there is no convenient way to modify them. An application with such functionality is clearly desired. Such a program was created, and named the *HDFG Editor*.

This thesis describes the design process and functionality of the *HDFG Editor*. It also briefly touches on the impact the product will have on the Delft Workbench.

The main concern while designing the *HDFG Editor* was that it should be easy to use but still provide a lot of functionality. Achieving this was attempted by communicating with potential users of the program and involving them in the evaluation process.

This thesis is structured as follows. In Chapter 2, the problem of graph visualization and manipulation in is described in greater detail and a solution, the *HDFG Editor* is proposed. In Chapter 3, the context of this thesis, namely the Delft Workbench and the DWARV toolset is expressed. In Chapter 4, the requirements for the *HDFG Editor* are documented. From the requirements, the most important design choices were made, which are described in Chapter 5. In Chapter 6, the internal implementation of the product is presented. Next, the employed verification process is portrayed in Chapter 7. Finally, in Chapter 8 conclusions on the usability and effectiveness of the implementation are drawn and recommendations for future use of the *HDFG Editor* are given.

2 Graph modification problems

HDFG representations[25] are very helpful in analyzing complex program structures. From a clear representation, data dependencies and hardware requirements can easily be deduced. To automate the creation of such a clear representation proper software is necessary.

In Section 2.1 the problem of graph modification is depicted. Section 2.2 portrays methods in use for HDFG manipulation. The final section of this chapter introduces a solution to the problem of visual graph manipulation.

2.1 Viewing and editing graphs

Although software that visualizes graphs is widespread and freely available, said software is unable to address all characteristics of HDFG representations. For example, many nodes have properties and extra information that existing software is unable to cope with. More importantly, no applications currently exist that allow visual manipulation of HDFGs. These limitations present an obstacle in the design process of software for reconfigurable computing.

Graph modification is easiest in a What You See Is What You Get (WYSIWYG) editor. Such editors allow for visual manipulation of nodes and edges. Modification based on textual commands, or file manipulation are considerably more difficult.

2.2 Currently available solutions

Currently, existing software is used to view HDFGs in order to achieve some degree of visualization. Editing HDFG representations is out of the question as saving such a file would cause the HDFG properties[1] of the graph to be lost as such software is unable to cope with that.

Because of the lack of suitable software, HDFGs are currently analyzed mainly by hand using pen and paper. This, of course, takes a lot of time and is a tedious process. This method is also very impractical for large graphs.

A solution would be not to use HDFG representations. Other ways of analyzing the structure and dependencies of software applications exist. These other formats, however, are again represented as graphs so the problem of finding an application able to modify such a format remains. Also, the HDFG format was specifically designed for DWARV and presents the optimal way of representing a program schematically. In addition, it is by far the most accessible format for other tools in the same toolset.

2.3 Solution: the Graph Viewer

The problems mentioned above can be solved simply by having a program that is able to view and edit HDFGs. Such a program would ensure that every characteristic aspect of the HDFG format is handled properly. The modification capabilities of this program would also present users with a trivial way to save small alterations to the original HDFG. With a WYSIWYG editor, the user would no longer have to change the source files with an increased risk for errors. This program is named the *HDFG Editor*.

[1] These properties are described in more detail in section 3.2

3 Related research

The *HDFG Editor* will be part of the Delft Workbench (DWB) tool chain for reconfigurable computing. One of its applications will be to gain insight in the scheduling of hardware operations. Before starting its design, some research was done about the DWB, reconfigurable computing in general, scheduling and pipelining. Also, existing graph viewing applications were examined.

Section 3.1 details on reconfigurable computing. More information on scheduling can be found in Section 3.2. Section 3.3 deals with pipelining, both the hardware and software variants. Hierarchical Data Flow Graph (HDFG) representations are portrayed in Section 3.4. The final section describes existing graph viewing applications.

3.1 Reconfigurable computing

Currently available General Purpose Processors (GPP) allow us to execute a wide variation of programs, regardless of their nature. Also, they allow us to execute said programs on various system configurations. A concern, however, is the speed limitation these GPPs induce. This speed limitation can be circumvented by using dedicated hardware instead of software but that is an inflexible and time-consuming operation.

Reconfigurable computing introduces a solution to the speed limitation by combining dedicated hardware and GPPs. This combination results in the flexibility of software in addition to a significant speedup. It works by connecting a Field Programmable Gate Array (FPGA) and a GPP. The FPGA can be programmed during run-time, and its contents can be executed much faster than if they were run on a GPP. The physical limitations that come with FPGAs like a finite number of gates are less important as that part of the code can be executed on the GPP. This parallelization of the FPGA and a GPP results in the greatly desired speedup.

3.1.1 The MOLEN hardware platform

In light of the developments in reconfigurable computing the MOLEN hardware platform was designed to improve certain aspects of the hybrid computing environments[18]. Among others, these aspects include performance. These suggestions consist of a one-time instruction set extension, a change in processor organization and the development of a back-end compiler.

An addition of eight instructions to the instruction set of an architecture (πISA) is sufficient to enable the proposed programming paradigm. These instructions include a `set` and an `execute` instruction to program the reconfigurable hardware and execute the loaded code. To allow communication with the GPP a number of dedicated registers are available, they can be accessed through the `movfx` and `movtx` commands. The remaining four commands enable higher performance and synchronization but are not required to enable the concept to work. They are beyond the scope of this thesis.

In order to distinguish between instructions intended for the FPGA and instructions intended for the GPP an arbiter is introduced. This arbiter partially decodes instructions and forwards them to their intended recipients. The data memory is shared between the GPP and the FPGA. Communication between the GPP and the FPGA is available through exchange registers (XREG).

As general compilers do not generate the newly developed MOLEN specific instructions, a back-end compiler is to be developed. Said back-end compiler should implement the extra πISA instructions to enable execution on a hybrid platform.

The MOLEN platform is currently implemented for experimentation purposes. The minimal πISA (only the `set`, `execute`, `movfx` and `movtx` instructions) is implemented using a PowerPC

405 as GPP and a Virtex II Pro as the reconfigurable hardware component.

3.1.2 The Delft Workbench tool chain

The Delft Workbench (DWB) [1] is a tool chain used for developing software for the MOLEN platform. It automates the process of compiling code for the MOLEN platform, including creating a hardware description for the FPGA, generating assembly code and replacing parts of the assembly code with the appropriate FPGA calls, as discussed in section 3.1.1 and in [18]. Of course, it is always possible to perform some part of the compilation process manually, as this may provide more efficiency.

The compiling process of the DWB is as follows. First, the profiler selects the parts of the code which are suitable for implementation in hardware. For these parts, the DWARV compiler generates a VHDL description. For this, the DWARV compiler uses an intermediate representation in the form of an HDFG, the format that the *HDFG Editor* will be using. The remaining code is compiled to normal assembly using the MOLEN GPP compiler, with FPGA calls inserted at the correct spots. The DWARV compiler is described in more detail in the following section.

3.1.3 The DWARV toolset

The DWARV toolset is part of the DWB and generates a hardware description from C source code[25]. The input for the DWARV compiler is C source code with pragma annotations to highlight the code that is to be implemented in hardware. The toolset consists of two separate modules: a Data Flow Graph (DFG) Builder and a VHDL Generator.

The purpose of the DFG Builder is to perform high-level optimizations, regardless of the hardware used, and create an intermediate representation. It is implemented as an extension of the SUIF2 compiler framework[2]. The output of the DFG Builder module is an Hierarchical Data Flow Graph (HDFG).

Upon completion of the DFG Builder, the resulting HDFG is processed by the VHDL Generator. This tool implements low-level optimizations that vary with different kinds of hardware and eventually generates a VHDL description.

3.1.4 Hierarchical Data Flow Graphs

HDFGs are flow graphs that contain information on data dependencies and precedence requirements [25]. They consist of two types of nodes. The simple nodes represent arithmetic and logical operations and data transfers to and from memory and registers. The compound nodes symbolize pieces of code and usually contain HDFGs themselves. There are two types of edges. The first represent data dependencies, indicating that the result of the first node is the input for the second node. The second type represents precedence requirements, indicating that the first nodes operation should be finished before the second nodes operation starts. From these HDFGs the DWARV VHDL Generator automatically creates a hardware description.

In the second version of the HDFG format[22] there are 48 different types of nodes and edges. To illustrate the variety of node types Table 3.1 shows the distribution of node types over different categories.

[2]More information on the SUIF2 compiler can be found on http://suif.stanford.edu/suif/suif2 as cited in [25]

Node category	Number of node types
Simple nodes	*45*
Operation nodes	26
Memory transfer nodes	2
Parameter transfer nodes	2
Constant nodes	1
Splitter nodes	1
Data storage nodes	3
Compound operation nodes	*3*

Table 3.1: Node types per category, as described in [22]

3.2 Scheduling

In a system using reconfigurable computing, parts of an application are executed in parallel. Some caution is needed in deciding which parts to parallelize. For instance, one operation might need the result of another operation as its input. If these operations take place on an FGPA, as is done in reconfigurable computing, the second operation needs to be placed after the first one. The process of assigning time slots operations is called scheduling.[3]

There are several ways of representing dependencies between operations. One of them is the HDFG, as discussed in Section 3.1.4. As nodes represent operations in an HDFG, a visualization of an HDFG gives a picture of possible orders of operations. The compound nodes allow one to look at parts of an application at a detailed level when needed. This makes the *HDFG Editor* a useful tool for manual scheduling.

A simple example of mapping an HDL description to an FPGA is given below. In part (a) of figure 3.1, (part of) an HDL description is given. The discription is mapped to the data flow graph in (b). From the graph is visible that the result of A+B needs to be calculated before E*(A+B) and (A+B)*(C+D) can be calculated. For the latter, (C+D) needs to be calculated first. But the operations A+B and C+D can be parallelized, as well as E*(A+B) and (A+B)*(C+D). In this example, however, it was chosen not to parallelize the two additions so that the the adder and multiplier can be reused. This results in the schedule given in part (c) of Figure 3.1 and the resulting hardware in part (d).

3.3 Pipelining

Pipelining is a technology where several steps within one instruction overlap steps from other instructions. It is one of the main reasons general purpose processors acquire their current speeds.

3.3.1 Pipeling principle

An analogy to pipelining is restaurant service. The non-pipelined process of dining in a restaurant is as follows:

1. A customer reads the menu and orders a meal.
2. When the meal is ordered, the cook prepares the food.
3. When the food is prepared, the customer consumes the meal.

When the food is consumed the customer pays and leaves. If more customers are waiting, the next meal can be ordered. Assuming each step takes 30 minutes and there are four customers, the non-pipelined approach is displayed in Figure 3.3.

[3] Actually, scheduling is broader and also considers space restrictions as well as time restrictions, but for this discussion only timing restrictions are relevant.

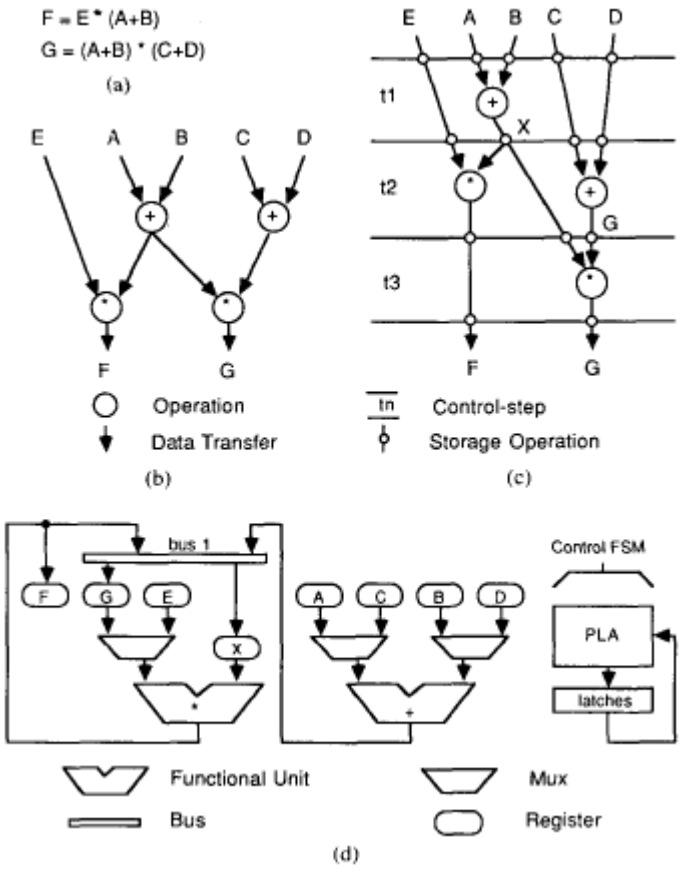


Figure 3.1: An example schedule. (a) HDL description. (b) Control data flow graph (CDFG). (c) Scheduled CDFG. (d) RTL architecture. Source:[16]

The pipelined approach to the same process is much more efficient. The efficiency is reached by not waiting for the entire process to finish before restarting the sequence. While the cook prepares the food, another customer can read the menu and order. While the customer consumes the food, the cook can prepare another meal. This process is depicted in Figure 3.2. Figures 3.2 and 3.3 show that for this analogy pipelining results in a completion time of 3 hours instead of 6 for the non-pipelined variant. It illustrates the importance of parallelization, compared to sequential execution of statements.

3.3.2 Hardware pipelining

In computer architectures, pipelining is achieved through parallelization of instruction steps. Hennessy and Patterson give the following example for the MIPS processor[15]. For the MIPS processor the following instruction steps are defined:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously.
3. Execute the operation while decoding an address.
4. Access an operand in data memory.
5. Write the result into a register.

These steps can be pipelined, similar to the restaurant analogy. Figure 3.4 depicts the pipelining of three load word instructions. In the above part, a timeline is given for the non-pipelined

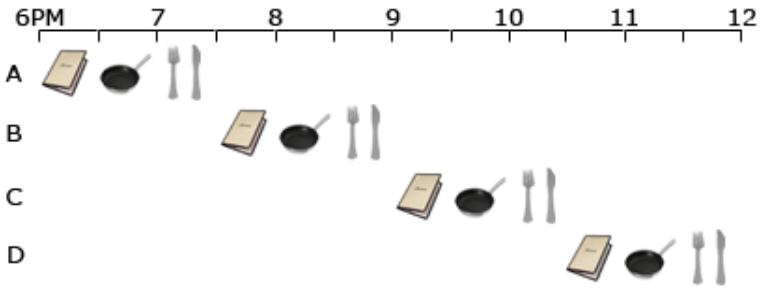


Figure 3.2: The restaurant service anology to pipelining; non-pipelined approach

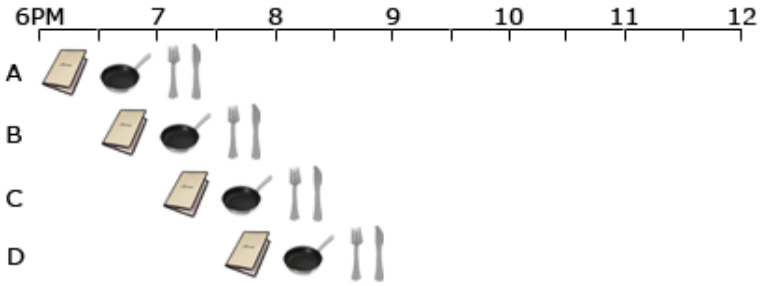


Figure 3.3: The restaurant service anology to pipelining; pipelined approach

case. However, each of the load word instructions is divisible into parts that can be parallelized and each of these instruction steps can be executed in parallel with other instruction steps.

Not every series of instructions can be executed in parallel as simple as in Figure 3.4. Branch instructions are very difficult to parallize, as after the branch instruction there are two possible follow-up instructions. Only after the complete execution of the branch it is known which instruction is next, but due to parallelization the next instruction needs to start right after the instruction fetch step of the branch. To solve this problem, a technique called branch prediction is used[15]. One of the two possible instructions is guessed. If afterwards this guess turns out right, the execution continues. If the guess turns out wrong, the pipeline is filled with wrong instructions, which need to be cleared before execution can continue. This clearing stage is called flushing. While flushing, nop[4] instructions are put into the pipeline until it is 'clean' again. Thereafter, normal execution continues.

3.3.3 Software pipelining

Extra pipelining can be achieved by concurrent scheduling of loop iterations. This is, however, a complex task that would result in complex hardware. Hence, this scheduling of loop iterations is achieved during compilation, and is called software pipelining. Lam describes an algorithm that obtains software pipelining for innermost loop iterations[14].

Instead of sequential execution of loop iterations, Lam defines an iteration interval to schedule iterations concurrently. By identifying minimally indivisible sequences of operations the iteration interval is retrieved. Subsequently new iterations of the loop are scheduled to begin each interval.

Using this interval, both resource constraints and precedence constraints must be taken into account. Concurrent scheduling of two loop iterations may require twice the amount of registers required if no pipelining is used. Of course, these registers must be available. As several loops can be scheduled in different ways, Lam recommends "to use software pipelining aggressively, by assuming that there are enough registers"[14]. Should the amount of registers

[4] No operation

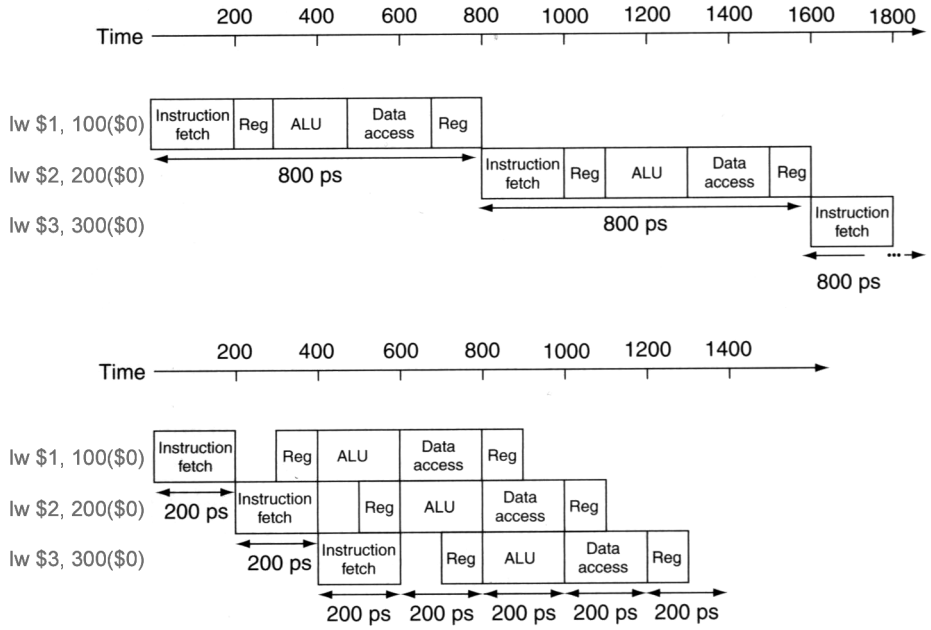


Figure 3.4: MIPS pipeline example. Source: [15]

prove insufficient, simple techniques to serialize iterations can be used to decrease the number of registers in use.

Precedence constraints refer to operations that are not part of a minimally indivisible sequence of operations, but do require another operation to finish before executing. A solution to this problem is modulo variable expansion. This method speeds up code by using different registers in alternating iterations. An example is writing a value in a register and using it two cycles later, as depicted in Figure 3.5[14]. Would only register `R1` be used, these operations would not be scheduled as efficiently, as the `Use(R1)` must be executed before the `Def(R1)` operation in a subsequent iteration. Using both `R1` and `R2` registers, however, this loop is considerably sped up. Notice that here a tradeoff is made between speed and register usage.

```
      Def(R1)
      op        Def(R2)
L:    Use(R1)   op        Def(R1)
                Use(R2)   op        Def(R2)   CJump L
                          Use(R1)   op
                                    Use(R1)
```

Figure 3.5: Modulo variable expansion. Source: [14]

Lam also uses hierarchical reduction to increase parallelization. A daunting task is the scheduling of loops with conditional statements. After independent scheduling of the `THEN` and `ELSE` branches, the entire branch is then reduced to one large block, with a length of the maximum of both branches. This block can subsequently be scheduled with other operations.

Hierarchical reduction also applies to the first and last operations of a loop; the ones that are not completely parallelized with other loop iterations. By reducing the loop to a large block, it can be parellelized with other operations.

3.4 Hierarchical Data Flow Graph representations

HDFGs are stored in files in one of two formats: simple text (ASCII) or binary. Also, two versions of the format exist. A full description of the binary formats can be found in [23] and [22].

The ASCII version is the easiest to humanly decipher. It is text based and can be read without need for conversion of any kind. As a consequence of this, these formats are also the easiest to be used for manual graph manipulation.

The binary format is the most space efficient and the easiest to parse for computers. It stores numeric values directly and as such no text to numeric conversion needs to be applied. This results, however, in a file that is only readable through deciphering it byte by byte. The full description of the version two binary HDFG file is documented in [22] and summarized hereafter.

Standard conventions in the binary file format are the representation of numeric values, vectors, strings and data types. Numeric values are stored as 32 bits unsigned integers in little endian[5] format. Not available or not applicable values are stored as `0xFFFFFFFF`. Vectors are preceded by an integer representing the total number of vector elements. The size of one vector element depends on the vector context. Strings are character vectors, so a string is preceded by its length and then one byte for each string character is stored. A terminating zero is not included.

Data types are either local or non-local. Non-local data types are 'common' data types such as integer, boolean and floating point. Each of these non-local data types has a numeric constant associated to represent the data type in the binary files. Local data types are arrays, structures and unions. These local data types vary in amount of memory they require and are specified more accurately. First a numeric constant is stored, identifying the data type as an array, structure or union. Subsequently a numeric identifier for the specific data type is stored. At the end of the file complete descriptions of all such data types are collected.

The structure of a version two binary HDFG file is as follows. First, the graph name is stored. Subsequently, a vector of nodes follows. Each node consists of a unique numeric identifier followed by a numeric node type. This numeric identifier i is not necessarily incremental nor is $i \notin \{0, 1, \ldots, n-1\}$. Depending on the node type the following bytes can be interpreted in several ways. Hence the number of nodes is known, but not the size (in bytes) of the vector. Due to the adaptable nature of the *HDFG Editor* all such node type descriptions are included in the repository (Appendix B). An example of a node description is the arithmetic addition operation node. After the node identifier and type, four fields follow. First two node numbers representing input nodes are stored. After that a data type is stored. The last field is a vector with node numbers representing precedence nodes.

After all nodes have been described a list of sections is stored. Each section is preceded by a section type number. In each section a vector of descriptors is included. Each descriptor has a header containing a unique numeric descriptor identifier, a name string and an integer representing the number of instances of this section in the HDFG. What follows after the descriptor header depends on the section type. For adaptability purposes, these section types are defined in the repository. Though every descriptor identifier must be unique, multiple sections with the same section type number can be included in the binary file.

3.5 Existing graph viewing applications

There already exist applications for viewing graphs. In this section, five of those programs are looked at more closely.

[5] With little endian the least significant bite is stored first

One application is uDraw. uDraw is able to draw graphs and offers a few visualization options, such as giving nodes a custom shape and color. On the uDraw download page[13] the creators state that the source code of the program will become available in the future, but it was not at the time of writing. uDraw is usable in two ways. The first is to use it as a stand-alone application. The second is as a graph drawing API in combination with a custom created program. However, connecting a program to uDraw is fairly complicated and is done either by pipes or TCP/IP sockets [12].

Another application for viewing graphs is Graphviz. Graphviz is open-source, runs on multiple platforms and has many features regarding layout and style [5]. Graphviz does not have a graphical user interface by itself but a few are available [7]. However, after examination of a few of those interfaces it is concluded that they have little functionality for editing graphs.

The Graphical Editing Framework[3] (GEF) is an Eclipse plugin, capable of, among other things, visualizing graphs. It is open-source and has editing capabilities. An advantage is the number of functionalities supported by GEF such as file dialogs, common controls and keyboard navigation. A disadvantage of GEF is that it is an Eclipse plugin and thus requires Eclipse to be installed[20].

GEF is layer over another Eclipse plugin: Draw2D. This plugin is a tookit for drawing and supports graphs. Other then GEF, it does not support editing of any kind, leaving that to be implemented[20]. Same as with GEF, Draw2D remains an Eclipse plugin, requiring Eclipse to be installed.

Finally, there is a program created by Hurkmans and Kentie who got the same assignment as this project. Their program only works with an old version of the .dfg file format, as the format was renewed in the past year. Also, the program is not platform independent and it becomes slow when used with large graphs[8].

4 Requirements for the Graph Viewer

Before starting the actual development and actually writing the code the requirements were listed and the set analyzed. The list can be separated into functional requirements and non-functional requirements. Functional requirements represent functionality the *HDFG Editor* should have, whereas non-functional requirements list what limitations and boundary conditions should be observed. To ensure that the wishes of the end users were not improperly interpreted a set of use cases was designed.

The functional and non-functional requirements are described in Sections 4.1 and 4.2 respectively. Use cases are illustrated in Section 4.3 and Appendix B.

4.1 Functional requirements

It proved to be efficient to separate the functional requirements into four subsets, each representing a major functionality. These major functionalities are file handling, graph visualization, graph manipulation and repository functions. To prioritize these functionalities a MoSCoW document (Must have, Should have, Could have, Would have) was written to speed up the writing process and ensure the availability of a working program at the end of the timeline. The MoSCoW document can be found in Appendix A.

4.1.1 Functional requirements for file handling

The *HDFG Editor* is required to have the following functionality for file handling:

- Loading binary and text files containing HDFGs.
- Storing HDFGs into a binary or text file.
- Exporting HDFGs to .jpg and .eps files.
- Loading the repository from file.
- Storing the repository to file.

4.1.2 Functional requirements for graph visualization

The *HDFG Editor* is required to have the following functionality for viewing graphs:

- Visualizing the loaded graphs.
- Zooming in and out.
- Opening subgraphs.
- Merging sub-graphs within the upper-level graph.
- Associating a property sheet with every node and edge. This property sheet has to be displayable as a separate dialog box.

4.1.3 Functional requirements for graph manipulation

The *HDFG Editor* is required to have the following functionality for editing graphs:

- Creating a new graph.
- Creating a new node.
- Creating a new edge.
- Applying semantic checks when nodes and edges are added.
- Moving nodes and edges by dragging them on the screen.
- Modifying the values of existing properties. During this modification, a semantic check has to be applied.

- Deleting nodes and edges.

4.1.4 Functional requirements for the repository

The *HDFG Editor* is required to have the following functionality regarding the repository:

- Importing repository data.
- Creating new types of nodes.
- Creating new types of edges.
- Modifying existing types of nodes, including their properties.
- Modifying existing types of edges, including their properties.

4.2 Non-functional requirements

For the *Graph Viewer*, the following non-functional requirements apply:

- The program is open-source.
- The program is platform independent.
- The program is stand-alone and does not require additional applications to be installed.

4.3 Use cases

Use cases present a way to clarify the different usage scenarios that occur[2]. As often the end-user has a different point of view than the programmer use cases are a way to make sure the wishes of the end-user are met.

For most functional requirements, a use case is described. Some functionalities, however, do not require user interaction, removing the necessity for a use case. These requirements are:

- Visualizing loaded graphs
- Applying sematic checks

A full list of use cases is included in Appendix B.

5 Design choices for the Graph Viewer

Before being able to write the program itself, a few choices had to be made. First, a starting point had to be chosen, such as starting with an existing application or building the complete application ourselves. Even though it was chosen to build the complete apllication, part of the functionality of the program has already been created and made available by others in the form of a library. These libraries are often open source, which is a requirement for the *HDFG Editor*. As there was no need to redo the work already done by others, libraries were used wherever possible. In this chapter the choices for the code base is explained in Section 5.1. The choices for the graphics library, the file input/output library, the graphical user interface library and the image export library, respectively, are considered in Section 5.2. In Section 5.3, the design for the graphical user interface is presented.

5.1 Available code bases

From the related research, the following options for a code base become apparent:

- Start with the program from Hurkmans and Kentie and improve it.
- Start with uDraw and create a program that works together with it.
- Write an interface for Graphviz.
- Use the drawing component from the Graphviz source and create a program around it.
- Use GEF or Draw2D as a codebase.
- Write the complete program completely.

Each of these options has its advantages and disadvantages, listed below.

The program from Hurkmans and Kentie already has the desired look and feel. It also is able to load HDFG files, but only in an old format, as the file format was renewed in the past year. In addition to adding support for the new file format, the drawing component also would have to be recreated, as the current draw component becomes too slow with large graphs and uses GDI+, which is not platform independent[17]. Finally, the program does not have a repository. Adding a new node type forces one to recompile the program. These disadvantages would cause a lot of rewriting, so it was decided not to choose this option.

Starting with uDraw has the advantage that the drawing part already is done. However, uDraw is not open-source. Therefore, this option was dismissed quickly.

Writing an interface for Graphviz looks attractive, as it completely handles all drawing functionality. However, as Graphviz is solely a drawing tool, it becomes difficult to modify graphs. Graphviz takes as input a graph in DOT language, which doesn't specify coordinates for nodes[6]. This means it becomes very difficult to handle mouse click events, for example. Dragging nodes also becomes very difficult. Therefore it is more attractive to use (part of) the source code of Graphviz and build our own application around it. That way, the program can still control the position of the nodes. One disadvantage is that the complete source code of the program would have to be examined thoroughly.

As stated in the related research, GEF and Draw2D require Eclipse to be installed. This is against the non-functional requirements as those require the application to be stand alone. It is, however, possible to detach GEF from Eclipse. When asked if this is supported: "The official answer is No, this is not supported"[20]. Still, GEF is open source so it is possible to alter the source code so that it no longer requires Eclipse. Unwilling to use undocumented third party code that accomplishes this, altering the GEF source code is a daunting task.

The last available option is write the whole program. This gives complete control over the program but has a single large disadvantage in that it is a lot of work.

In the end it was chosen to create the complete program from scratch. Every option required a lot of code to be created and it was decided that the ability to design the whole program structure would increase programming efficiency.

5.2 Graphics library

For the graphics library, an enormous amount of options was available. Examining every option in depth would require too much time, so most options were considered only briefly. In the end, three candidates were considered more closely:

- GDI+
- Irrlicht Engine
- Simple DirectMedia Layer

GDI+ is a 2d graphics library. It provides all functionality needed for the program and it is fairly easy to use. The largest disadvantage of GDI+ is applications using it can only run on Windows, which goes against the non-functional requirements.

The Irrlicht Engine is not just a 2d graphics library; it provides a lot of additional functionality, such as 3d rendering and graphical user interface building, making it seem a less attractive choice than one of the more suited libraries. However, its documentation[4] is very good as it is extensive and accessible and provides a lot of information. It also is compatible with a variety of display drivers, making it platform independent.

Simple DirectMedia Layer (SDL) is a library written over OpenGL, making it compatible with all commonly used platforms[17]. It provides all graphic functionality needed by the program and some more, but not as excessive as the Irrlicht Engine. The main drawback of SDL is that its documentation isn't very good and that it is low-level, meaning some additional work would still be needed to be able to use it.

In the end, the Irrlicht Engine was chosen, as it is accessible and easy to use. Also, the additional functionality it provides is useful as for example GUI building is a time-consuming task and Irrlicht automates the process.

5.3 User interface

The user interface is important as it is the only way the user has to communicate with the program. As it handles all interaction it should be user-friendly and easy to understand. After analyzing the requirements one of the first steps was the creation of a concept user interface.

The concept interface clarifies the overall layout of the program. First of all, the window consists of roughly four sections. On top there is the menu and a toolbar with buttons that provide common functionality. At the bottom there is a status bar that displays what action is currently being performed, such as processing a file or exporting to image. The center left section is the actual body of the program. This is where graphs will be drawn and where nodes and edges can be selected. Note that there are two separate canvases where graphs can be drawn to enable viewing multiple graphs at the same time, for example a subgraph and its parent. Different open graphs can be accessed through a set of tabs, directly above the canvases. On the right is a property listing, which contents will vary depending on what is selected on the left.

As the *HDFG Editor* has several requirements that must be activated through menus (see the use cases in Chapter 4) there is a large set of commands in the menu. Highlighting the most important ones in a toolbar is a previously proven efficient way of improving the accessibility of the program. The menus themselves are grouped in three sections. The file menu contains basic commands such as 'New graph', 'Save', 'Open' and other commands one expects to find

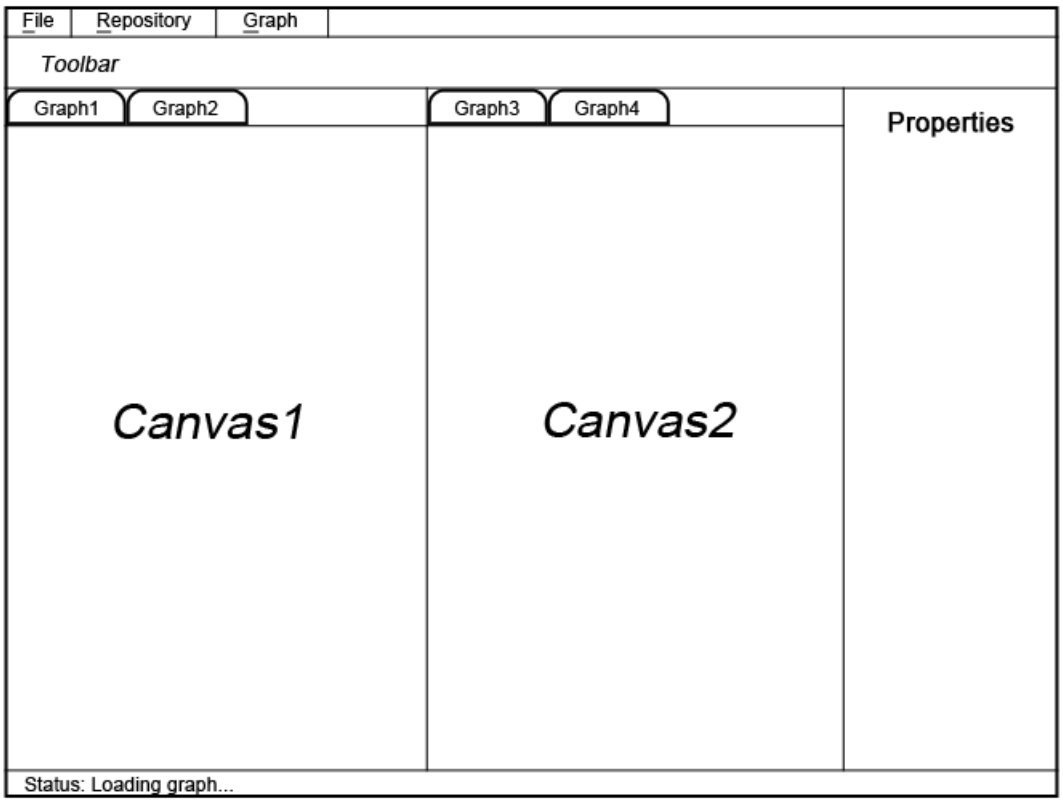


Figure 5.1: The concept user interface

in a 'File' menu. They will be present in the toolbar as well. The 'Repository' menu contains commands for the repository such as 'New node type'. As these commands will not be used often they will not be present in the toolbar. The 'Graph' menu contains commands specific for a graph such as 'New node'. Most of these commands will also be added to the toolbar.

The status indicator visualizes otherwise invisible actions. As the program will handle large files it is probable that processing such a file will take some time. In this case the status indicator will let the user know what action the application is performing and more importantly how much of the process has already been completed. This improves the user-friendliness of the application.

It is important that at least two graphs can be visualized at the same time. This automatically requires at least two canvases within the program. It should also be possible to open several (sub)graphs at the same time and easily navigate between them. Tabs enable users to navigate between graphs in a canvas and by dragging a tab the graph can be moved from one canvas to the other.

Each object, be it a graph, node or edge has its own set of properties. These properties must be easy to find. The properties bar on the right presents a universal way of displaying said options and makes editing them trivial. The content of this bar varies depending on the object that is selected. For example, if a node is selected its name and type will be among the displayed properties.

5.4 Dynamic behaviour

The graph viewer is to be a dynamic application in the sense that each displayed item can change in both appearance and binary representation. Two ways to realize this present themselves: inheritance tree and external representation.

Object-oriented programming naturally allows general classes to be specified through an inheritance tree. A general node class can then be extended for each node type, resulting in a very easily modifiable program structure. To change this program structure, however, the application is to be recompiled. This requires a compiler, the entire graphics library Software Development Kit (SDK) and some knowledge of the existing code.

Another way to achieve dynamic behaviour is through external representation of certain classes. This external information is then read, parsed and interpreted during runtime. By editing this file the runtime behaviour of the application can be altered, making it dynamic. This external file is named a repository.

Both solutions have advantages and disadvantages.

An inheritance tree is computationally faster. Compilation optimizes the application structure and an external representation has overhead due to reading the file. Also, during runtime information needs to be looked up. An inheritance tree does not have this disadvantage due to the static program structure.

Modification of an inheritance tree, however, can only happen by editing the source code. This requires access to a compiler, source code and the graphics library SDK. Editing a repository is much simpler: a text editor is all one needs.

On the one hand, an inheritance tree offers more flexibility then an external representation. Every single aspect can easily be modified, whereas a repository only supports dynamic behaviour for certain aspects.

On the other hand, the existing code is vastly more complex then the repository file syntax as it is greater in size and prone to style. The source code is to be analyzed whereas a simple file description is enough for the repository.

A repository was implemented. The dynamic behaviour is most important for edge and node types. Seen as how this is a small aspect the flexibility of an inheritance tree does not offer much advantage. Also, having a repository still enables one to edit the source code, leaving said flexibility intact.

5.5 Repository

Having decided to use a repository, it is important to decide what information it contains. The following data is implemented:

- **Node types**
  With over 40 different types[22] the node types are the main reason for implementing dynamic behaviour. The repository should contain enough information to read its binary representation from file. Aside from the numeric node type constant, this implies that each node property should be documented in the repository. Also important is the node category: simple, storage or compound. Furthermore, display properties such as node shape and symbol should be stored.
- **Edge types**
  With two supported edge types at the time of writing, edge types do not require dynamic behaviour. Nonetheless, adding edge types is a requirement and as such they should be included. For extendibility purposes, properties are to be supported as well. Edge representation in the binary file format should also be stored in the repository.
- **Section descriptions**
  The binary representation consists of a main graph, followed by a number of sections[22]. There are different types, each with a different representation. Sections have associated node types, so these should be stored. Sections have properties that should be included

to allow interpretation of binary files. The section category, compound or storage, is to be stored as well.

- **Data types**
  Several data types are supported in the binary file format. As they all have a numeric representation these should be included in the repository. Several local data types are supported as well so these should be distinguishable.
- **Constant sizes**
  Some, but not all of the forementioned data types have a previously defined constant size. These should be stored in the repository.
- **Custom fields**
  With properties for node types, edges and sections that vary in both type and representation, custom fields must be included. Custom fields should allow vector support, and compound properties.

6 Implementation of the Graph Viewer

This chapter presents the implementation details of the *HDFG Editor*. Several algorithms are discussed and analyzed, and several implementation decisions are documented.

The program structure is explained in detail in Section 6.1 through a class diagram. The second section illustrates the process of loading and storing graphs. Details on the implementation of the repository are displayed in Section 6.3. Algorithms for array indexation and node placement are discussed in Sections 6.4 and 6.5 respectively. Implementation details for HDFG visualisation are described in Section 6.6.

6.1 Class Diagram

A class diagram provides a solid basis for the entire coding process[2]. In an early stage this diagram was designed to help clarify the program structure. Figure 6.2 depicts the main application objects. Based on this diagram the classes can be built separately.

The class diagram is based on independent functionalities being handled by objects. These functionalities are as follows.

The Window class creates the Graphical User Interface (GUI) and handles related events such as resizing and clicking on toolbar buttons. It contains a Tab object, which is a container for graphs and their drawing environment. Such a drawing environment is a Canvas object.

Each Canvas object handles the drawing for one instance of the Graph class. It is responsible for drawing every node and edge. It also handles events such as clicking inside the canvas area to highlight nodes and edges. The linked Graph object has lists of Node and Edge objects. The Graph class also handles node positioning.

The Node class is extended by the SectionNode class, which in turn is extended by the CompoundNode class. These represent simple, storage and compound nodes respectively.

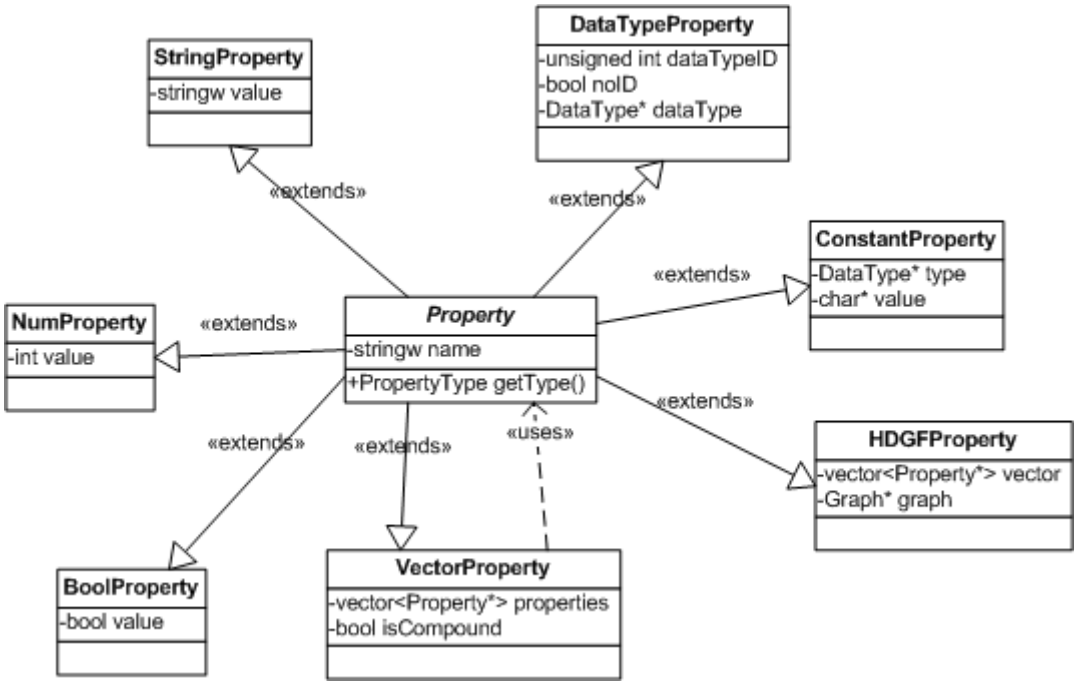


Figure 6.1: The property inheritance tree

The Window class has access to the FileHandler. Every action that has to do with file reading or writing is implemented as a static method in the FileHandler class. Its loadRepository

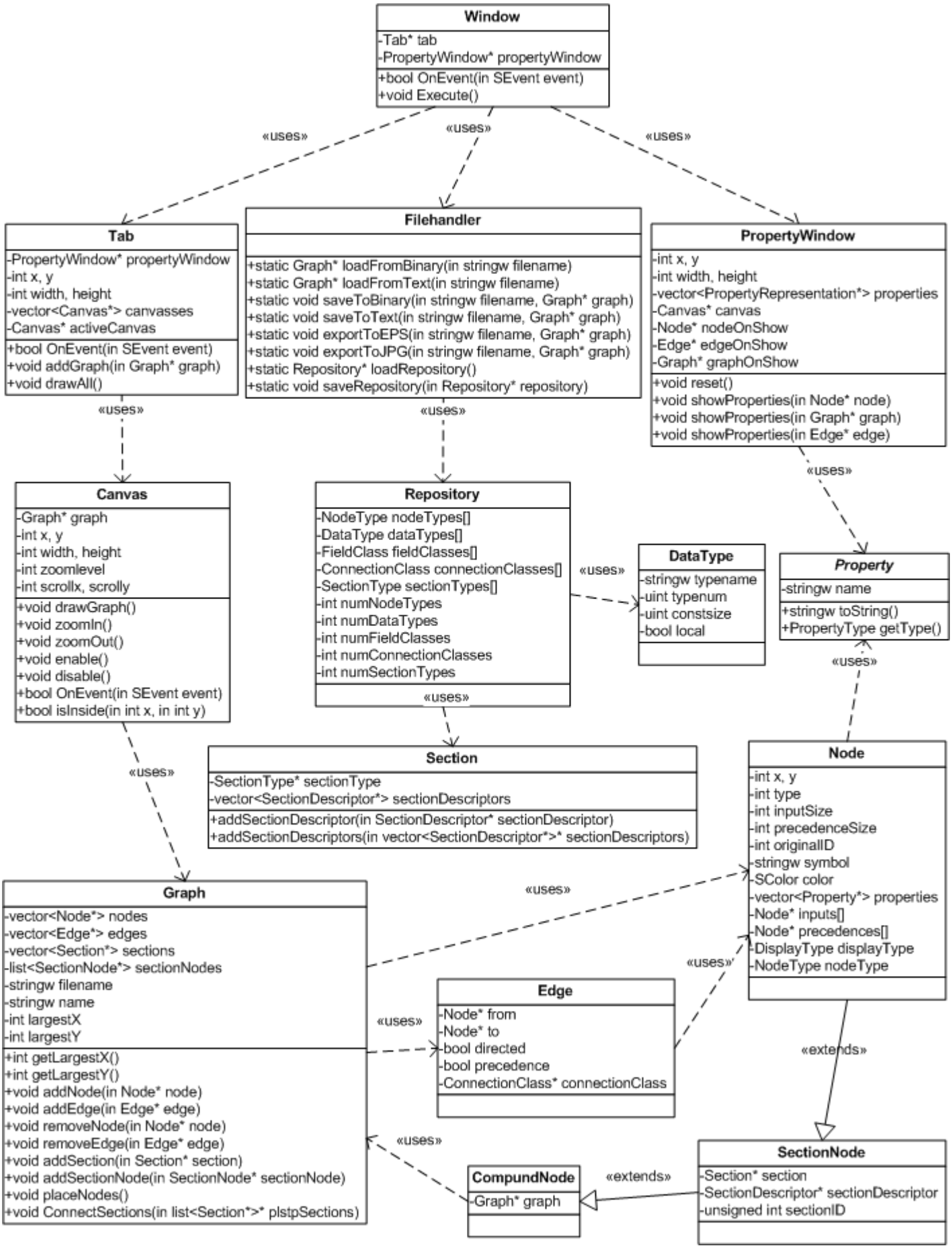


Figure 6.2: The main class diagram

method reads, parses and interprets an input file and creates an instance of the Repository class. This class is an encapsulation of a database for information included in the repository file, such as node types and section types (Section 5.5). Most of these are saved using internal structures.

There are two information sets that are not saved as structures. The first are sections, these are stored as Section objects. Instances of this class hold arrays of section descriptor classes themselves. The second are data types. Data types are stored as DataType objects due to the difference in local and non-local data types, and the usage of extra ID numbers.

Each Node, Edge and Section object has a set of properties. These are all instances of the abstract Property class. The Property class is extended by seven classes as depicted in Figure 6.1. Each of these classes is distinguishable through an internal enumeration and the getType method.

Some extensions are simply encapsulations for generic data types such as BoolProperty for booleans, NumProperty for integers and StringProperty for strings. Others represent more complex structures such as the ConstantProperty which represents a value in memory of variable size and type. The DataTypeProperty references to a DataType object, providing additional information on the availability of an ID number. The VectorProperty is either a vector of unknown length of properties of a similar type, or a compound property where several properties are grouped. The HDFGProperty is only used in sections and specifies a link to an instance of the Graph class.

The Window class also creates one PropertyWindow object. This results in an area where the PropertyWindow class draws controls for every property. Examples of such controls are edit and combo boxes. The PropertyWindow handles events for its own controls.

6.2 User interface

During the implementation the concept user interface as described in Chapter 5, was altered slightly. The implemented user interface is depicted in Figure 6.3.

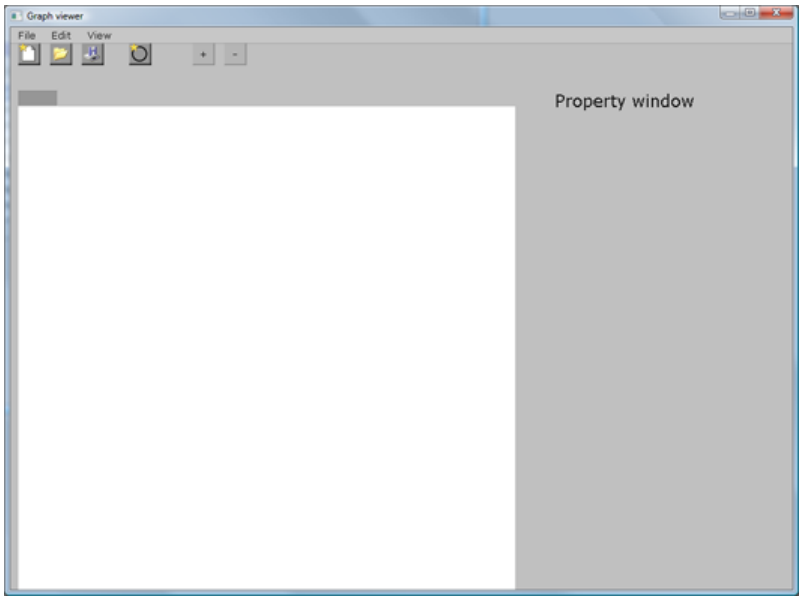


Figure 6.3: The implemented user interface

There are two major changes compared to the concept user interface as illustrated in Figure 5.1.

First, there is only one canvas, instead of two canvases. Due to time constraints the canvas array is not implemented. The application architecture, however, allows for extention with an array of cavases. This is accomplished through the separation of the Tab and Canvas classes.

Second, no status bar is present. The status bar was to display what action is currently performed. It is not implemented due to lack of time.

6.3 Loading and storing graph files

The process of loading HDFG files consists of five steps. The first is selecting the file to be opened and selecting which type of file is selected. As two versions of binary HDFG files exist[22][23] and they are not easily distinguishable from each other, the user is prompted for the version of the file. After validity checks on the filename, the file is opened.

The second step is reading the nodes. Each node has a unique number. The nodes are read and indexed (in constant time per node) to allow fast access to the nodes. Properties are simply interpreted and saved, but edges require different handling. As connections are represented by node numbers and such nodes may not have been encountered in the file yet the numbers are saved, but edges are not yet created. Hence, this step results in a complete list of nodes with information on connections. This step relies heavily on the repository as each node representation is documented there.

When all nodes have been processed edge creation can take place in step three. Using the indexed nodes and the saved connection numbers edges can be created. This is a $O(mn)$ process where n refers to the number of nodes and m to the number of connections. This complexity is a result of the lookup process as each connection number has to be matched to an existing node, and this process is repeated for every edge.

Step four entails placing the nodes throughout the canvas. In step two and three the complete graph structure has been formed, however, visualization cannot take place as each node requires a set of coordinates. The placement of these nodes and the algorithm used is described in Section 6.4.

The last step is the interpretation of sections. With an unknown number of sections this step is continued until the end of the file is reached. Section interpretation also relies heavily on the repository as the complete section descriptions are documented there. For compound section descriptors a new graph is read and steps two to four are repeated for this subgraph.

6.4 Repository

As described in Chapter 5, the repository allows for dynamic behaviour. It is a text based file named `repository.txt` and is located in the executable directory. Its complete syntax is described in Appendix C.

The repository is read, parsed and interpreted when the application is first started. Only one version can be active at a time. As a change in the repository redefines the way in which a binary file is read, the application must be restarted when such a change is made. An example of a repository entry is depicted in Figure 6.4.

Three techniques are used to increase the readability and usability of the file and assist the user in modifying it.

First, comments are supported. Useful comments identify sections and clarify otherwise abstract pieces of the repository. With these comments the end-user is able to edit the repository by simply commenting out old features to replace them by new ones, for debugging purposes.

Second, integers can be represented as decimal or hexadecimal. Because numeric constants in hexadecimal format are used in the binary file specification it greatly increases the readability

to be able to use hexadecimal constants. On the other hand, when specifying the length of an array decimal values are easier. Both formats can be used, with the hexadecimal values identified by the `0x` prefix.

Finally, strings are preferred to integers. This means that names are preferred to ID numbers and enumerations are represented as strings. The HDFG binary file specification uses numeric constant values to identify different structures. These constants are of course included in the repository. However, instead of referring to these constants within the repository, one uses the name to increase readability.

`#Loop section description`	*Comment identifying the item*
`SEC_LOOPS`	*Name that can be referred to*
`0x000000028`	*Numeric constant for binary file (hex)*
`SECTION_COMPOUND`	*Enumeration value*
`1`	*Integer value identifying array length (dec)*
`TYPE_LOOP`	*Previously defined node type name*
`3`	*Integer value identifying array length (dec)*
`Post flag condition`	*Name for display in application*
`VALUE_BOOL`	*Enumeration value*
`Condition number`	*Name for display in application*
`VALUE_INT`	*Enumeration value*
`HDFG section`	*Name for display in application*
`HDFG_GRAPH`	*Enumeration value*

Figure 6.4: Repository section example

6.5 Indexation of arrays

Array indexation allows to speed up array searches. With the loading of nodes and creation of edges a lot of node types are referenced by an internal number. If an array must be searched this takes linear time. As this step is performed at least once for each node, this results in a complexity of $O(np)$ where n is the number of nodes and p the number of node types. An existing solution for this is the map. A map is sorted by internal numbering, which takes $O(p\log(p))$ time, but due to this sorting items can be found in $O(\log(p))$ time. This is then a complexity of $O(n\log(p) + p\log(p))$ As many, but not necessarily all of the internal numbers lie in the range of $k \in \{0, 1, \ldots, n-1\}$ another method presents itself: indexation.

Say an array has n elements. The available array indices are $i \in \{0, 1, \ldots, n-1\}$. Say an element has an internal number $k \in \{0, 1, 2, \ldots\}$. If an element has index $k < n$ it is saved with index $i = k$. Elements that have a number $k \geq n$ are placed in the remaining positions of the array. The process is described in Algorithm 6.1.

6.5.1 Proof of array indexation algorithm

The first for-loop iterates over all elements. Its purpose is to place the elements with an internal number $k < n$ in the array with k as its index. As a consequence, it does not place all items in *Elements*. This is easily recognizable due to the store operation in an if-statement (lines 3-5). The second for-loop iterates over all elements as well. This for-loop, however, stores the items that have not been handled in the previous loop. As both arrays have the same size there is enough space for each element. Also, due to the requirement that k is unique, no two items are stored using the same array index. Conclusion: all elements are successfully stored and indexed.

Algorithm 6.1 Pseudo code for array indexation

Require: *Elements* is an array of size n with elements with a unique internal number k
Require: *Array* is an empty array with size n

```
1: for all i ∈ {0, 1, ..., n − 1} do
2:    Let e be the ith element of Elements and k be its internal number
3:    if k < n then
4:       Store e in Array with index i
5:    end if
6: end for
7: i = 0
8: for all e ∈ Elements do
9:    if e is not stored then
10:      while i < n and e is not stored do
11:         if Nothing is stored in Array on index i then
12:            Store e in Array with index i
13:         end if
14:         i = i + 1
15:      end while
16:   end if
17: end for
```

6.5.2 Complexity analysis of array indexation algorithm

The first for loop is executed exactly n times. The internal operations are all constant time so the run-time of this loop is $O(n)$. The second for loop is executed once for each element, so also exactly n times. Inside this for loop is a while loop that is executed as long as a predefined integer $i < n$. On every iteration of this while loop i is incremented, but it is never decremented or reset. This way the loop cannot be executed more then n times. This results in a complexity for the second for loop of $O(n)$. Hence, the algorithm runs in linear time.

6.5.3 Indexed array lookups

To perform a lookup on the index Algorithm 6.2 is used. This very simple piece of code takes constant time for items with an internal number $k < n$ and linear time for the others. Hence, the entire process of indexation is only useful if the internal number condition is satisfied for most of the array elements. If this is the case, the total complexity of indexation and p lookups is $\Omega(n + p)$. Worst case, the complexity of indexation and p lookups is $O(np)$.

6.5.4 Indexation implementation

If most of the internal numbers satisfy $k \geq n$ a map is a more efficient solution as then a comparison is made between $O(n)$ for an indexation lookup and $O(\log(n))$ for a map lookup. The total complexity for indexation is $O(cn)$. The constant c is linear to the number of items that have an internal number $k > n$. The total complexity for a map is $O(n \log(n))$. As long as $c < \log(n)$ indexation outperforms maps.

Array indexation is used for node types. At the time of writing, all node types for the version 2 HDFG file format specification[22] have an internal number $k < p$. With n the number of nodes, this means that instead of an $O(np)$ complexity while parsing graph files, the complexity is lowered to $O(n)$.

Array indexation is not used for nodes. No conclusion can be drawn on the probability of internal node numbers in the range $k \in \{0, 1, \ldots, n - 1\}$[24].

Algorithm 6.2 Pseudo code for indexed array lookup

Require: *Array* is an indexed array with size n
Require: k is the internal number of the sought element in *Array*

```
1: if k < n then
2:    return Element a of Array with index k
3: else
4:    for all i ∈ {0, 1, ..., n − 1} do
5:       Let a be the element of Array with index i
6:       Let t be the internal number of a
7:       if t = k then
8:          return a
9:       end if
10:   end for
11: end if
```

6.6 Node placement

When read from file, the nodes have to be positioned throughout the graph. There are many ways to obtain such a node distribution, many of which can be called optimal one way or another. Several algorithms have been tried and considered: random placement, random square placement, visually optimal placement and sequential placement.

The easiest and computationally fastest way is random placement, where the nodes are positioned at random. The algorithm runs in linear time, making it one of the fastest algorithms available. Because no grid is used the probability that edges cross nodes is lowered. Random node placement, however, can result in several undesired effects, the most important one being node overlap. This can be countered by checking for overlap, but this would increase the algorithms runtime to quadratic, eliminating its greatest advantage.

Random square placement also places the nodes at random, but in a grid, eliminating the cause to check for node overlap. As stated before, the grid is a cause for concern as now the probability that nodes overlap edges increases significantly. The random square placement algorithm runs in linear time, making it an attractive candidate.

Visually optimal placement is used by most graph visualization programs through force-directed or force-based algorithms. The purpose of such algorithms is to place the nodes in such a way that "all the edges are more or less equal length and there are as few crossing edges as possible" [21]. Force-based algorithms are based on physics where the graph is seen as a system and each node is seen as an electrically charged particle and each edge as a spring. Forces are now assigned and through iteration the system reaches an equilibrium.

Force-based algorithms provide one of the most visually attractive layouts, but are computationally very intensive. They have a running time of $O(n^3)$ where n reflects the number of nodes.

HDFG nodes represent operations and these operations are executed sequentially. Thus, it is possible to place the nodes in such a way that the graph represents the program structure. Algorithm 6.3 is an implementation of this placement method. It is based on the breadth first search algorithm in the way that it works through the graph layer by layer[11]. It is similar to the algorithm designed by Hurkmans and Kentie[9], but computationally faster as it runs in quadratic time.

Algorithm 6.3 Pseudo code for sequential node placement

Require: *Nodes* to be a list of all nodes

```
1: for all n ∈ Nodes do
2:    InEdges = number of incoming edges
3:    OutEdges = list of outgoing edges
4: end for
5: Candidates = ∅
6: Layer = ∅
7: LayerNumber = 0
8: while not all nodes have been placed do
9:    HorizontalIndex = 0
10:   while Layer ≠ ∅ do
11:      Get and remove l ∈ Layer
12:      Place l in a grid on coordinates (HorizontalIndex, LayerNumber)
13:      for all e ∈ l: OutEdges do
14:         With t = the target of e
15:         Add t to Candidates
16:         Decrement t: InEdges
17:      end for
18:      Increment HorizontalIndex
19:   end while
20:   if Candidates = ∅ then
21:      Add all non-placed nodes to Candidates
22:   end if
23:   Get the minimum number MinNum of incoming edges for all nodes in Candidates
24:   for all c ∈ Candidates do
25:      if (c has not been placed) and (c: InEdges = MinNum) then
26:         Add c to Layer
27:         Remove c ∈ Candidates
28:      end if
29:   end for
30:   Increment LayerNumber
31: end while
```

6.6.1 Proof of sequential placement algorithm

From line 8 it is clear that the algorithm does not finish unless all nodes have been placed. On every pass of the loop there is at least one node added to the *Layer* list (lines 24 to 29). Hence, the algorithm finishes with a complete set of placed nodes. Also, nodes do not overlap as after each placement *HorizontalIndex* is incremented and for each layer *LayerNumber* is incrementing resulting in different coordinates.

Ideally this algorithm displays the program structure top to bottom. It does this by never placing a node, unless all its parents have been placed. As edges represent logical and chronological links between operations, this way the graph is displayed properly. However, in the case of a cycle the simple test if all parents of a node have been placed is insufficient. The algorithm would enter an infinite loop which is clearly undesired. Instead of this simple test, the minimum number of input edges from nodes that have not been placed is calculated. If there is a node for which all parents have been placed this value *MinNum* $= 0$. In case of a cycle *MinNum* ≥ 0 but the algorithm will then still choose a node to place, preventing an infinite loop.

In the hypothetical situation that a graph contains a unconnected cyclic part, the algorithm could end up in an infinite loop. Not all nodes would be placed, but the *Candidates* list could be emptied, resulting in infinite empty layers. The simple test on lines 20-22 simply adds all non-placed nodes to the *Candidates* list if its empty, to prevent this situation.

6.6.2 Complexity analysis of sequential placement algorithm

Name n the total number of nodes and m the total number of edges. Initializing *Nodes* clearly takes $O(n)$ time. In the first for-loop all nodes and edges are passed, resulting in a run-time of $O(m+n)$. As each execution of the outer loop places at least one node, except the first pass, the runtime of the outer loop is $O(n)$. The combination of the outer and inner while loop is executed exactly n times as each node will end up only once in the *Layer* list. The for loop on lines 13 - 17 passes each outgoing edge for each node once. As each node is processed once each edge is only processed once so the complexity of this for-loop is $O(m)$. This results in a total complexity of the while loop of $O(m+n)$. Lines 20-22 have a complexity of $O(n)$. The calculating of the minimum number of edges on line 23 takes exactly $O(n)$ time. The for loop that directly follows loops through a number of nodes, and as it only contains constant time operations the time complexity is $O(n)$ as well. All these three structures are placed in the outer while loop so they are of total time complexity $O(n^2)$. All this results in a total time complexity of $O(n^2+m)$.

6.7 HDFG visualization

Visualisation of HDFGs is done with help of an external library. For drawing graphics, the Irrlicht library is used, as described in Chapter 5. However, a layer on top of Irrlicht is still needed as Irrlicht does not know how to draw nodes and edges. This layer is present in the form of the Canvas class. The canvas class is able to draw graphs, and provides functionality such as scrolling and zooming in and out. It also highlights selected edges and nodes by coloring them, which makes the look and feel of the program more attractive.

Drawing is done by calling the draw2DLine (for lines) and draw2DImage (for images) methods of the IVideoDriver class, which is provided by Irrlicht. This class completely abstracts away the underlying driver, so the user can use the video driver that gives the best performance on her platform, as described in [10]. As graphics usually are a large barrier for creating a platform independent program[6], this abstraction helped a lot for realizing platform

[6]This is because not every graphics driver is supported on every platform. For example, the DirectX video driver is only available on Windows.

independence.

Drawing a graph is an $O(n+m)$ task, as the program can simply run through the nodes and edges and draw each one. The drawNode method simply draws a circle, rectangle or diamond at the location of the node. Drawing edges is somewhat more complex when one or both of the nodes to be connected is located outside the canvas (i.e. the part of the screen that is used for drawing this graph). To determine what part of an edge has to be drawn, an algorithm was created. The pseudo code for an undirected edge is depicted in Algorithm 6.4. The version for directed edges is almost the same.

Algorithm 6.4 Pseudo code for drawing an undirected edge

```
1: Get the begin- and endpoints (x1, y1) and (x2, y2) of the edge.
2: if Both points are inside the canvas then
3:     Draw a line between them.
4: else
5:     Determine the line equation y = a*x + b of the canvas.
6:     Find the intersections between the line and the borders of the canvas.
7:     if One of the points is located inside the canvas then
8:         Find the intersection closest to the invisible point.
9:         Draw a line between the visible point and the intersection closest to the invisible
           point.
10:    else
11:        Draw a line between the intersections, if there are any.
12:    end if
13: end if
```

7 Evaluation of the HDFG Editor

During and after the implementation, the program has been evaluated, both by the programmers and by the potential users. In this section the results of those evaluations will be described; the programmers evalution in Section 7.1 and the end-user evaluation in Section 7.2.

7.1 Programmer evaluation

Before the implementation, serveral requirements were determined for the *HDFG Editor*. In the end, it appeared impossible to implement all functional requirements, mainly because of lack of time. When deciding which requirements to drop, the MoSCoW document from appendix A was used.

The following requirements are fully implemented and working properly:

- Loading the repository from file.
- Visualizing the loaded graphs.
- Creating a new graph.
- Creating a new node.
- Creating a new edge.
- Loading binary v2 files containing HDFGs.
- Moving nodes and edges by dragging them on the screen.
- Deleting nodes and edges.
- Creating new types of nodes.
- Creating new types of edges.
- Modifying existing types of nodes, including their properties.
- Modifying existing types of edges, including their properties.

Some functionality was implemented in a different way than described in the design choices, for varying reasons.

Associating a property sheet with every node and edge. This property sheet has to be displayable as a separate dialog box. It was decided that the separate dialog box doesn't really have a function as the window already has a properties section[7]. The properties sheet is simply displayed in the properties section of the window.

Storing the repository to file and importing repository data. The program comes with a repository filled with node and edge types which is loaded automatically when the program starts. This file is always called "repository.txt" and always located at the same path. The possibilities for importing other repositories and exporting ones own reposotiry to other users are dropped because of lack of time.

The following requirements were dropped because of lack of time:

- Exporting HDFGs to .jpg and .eps files.
- Zooming in and out.
- Opening subgraphs.
- Loading binary v1 files containing HDFGs.
- Loading text files containing HDFGs.
- Storing HDFGs into a binary or text file.
- Merging sub-graphs within the upper-level graph.

[7] See Figure 5.1

- Applying semantic checks when nodes and edges are added.
- Modifying the values of existing properties. During this modification, a semantic check has to be applied.

Although menus are part of both the use cases and the concept user interface they have not been implemented. Available functionality can be reached through toolbar buttons or interaction with the canvas.

There are also a few non-functional requirements. **The program is open-source.** This requirement is easiliy adhered to simply by including the source files with distributions of the program.

The program is stand-alone and does not require additional applications to be installed. There is one external library used by the application: the Irrlicht grapics library[8]. This library is included with the application in the form of a .dll file (for windows) or a .so file (for linux). Thus, nothing needs to be pre-installed on the users system.

The program is platform independent. All written code and all libraries used are platform independent, so in theory the program is platform independent as well. However, the program has not been tested on other platforms than Windows XP and Vista.

7.2 End-user evalutation

Before being able to get opinions from the users, some thought was put into the kind of input desired from them and how to get it. This verification methodology is described in section 7.2.1. Using said methodology two evaluations were done with the users, which are described in sections 7.2.2 and 7.2.3.

7.2.1 Verification methodologies

Before starting the evaluation process, it is needed to decide what data is to be gathered during the evaluation sessions and how to process the data. This is called the verification methodology. For software, a good way to evaluate a product is by having potenial users use it[2].

In the evaluation process two methodologies are used. Both are equally important as they represent two aspects of user interaction with the application. Both tests focus on user interaction, and less on bugs. Though it is very important to solve bugs, at this point the main purpose of the test is to gain feedback on the usability of the program. Both methodologies feature short short documents handed out to the testees that are included in Appendix D.

The first step focuses on end users that do not read the manual and wish to quickly use the application. It entails handing the user a list of program functionalities and asking them to complete a set of tasks. The tasks on this list represent all functionality the program has at this point. Without additional information, the user must attempt to complete said tasks in a way he or she deems logical. Hence, irregularities in how the testee desires to use the *HDFG Editor* and how the application actually functions are discovered.

The second step focuses on end users that skim or read the manual before using the application. It also entails handing the user a list of program functionalities, but the know beforehand how each task should be performed. The user will follow steps as the programmers have developed them. This way the end user is forced to think and act similarly to the programmers. The purpose of this way of evaluating is the same as in step one, but it

[8] See Section 5.2

removes user frustration from being unable to complete a task in a way the testee believes logical.

7.2.2 Evaluation results

As mentioned in the programmers evaluation, the *HDFG Editor* is far from finished. With the short available time, it made more sense to focus on trying to get the program finished than on evaluating. There have been two short evaluations, from which the following points became apparent.

- The look and feel of the program is good.[19]
- There are still several bugs. Not all are very serious, however, some are (e.g. crashes).
- Some functionality is implemented in an unexpected manner. For example, drawing an edge is done by dragging the right mouse button between two nodes. This may be a quick way, but is not what the user expects from experience with other applications.

8 Conclusions and recommendations

In this chapter the conlusions of the project are stated. In section 8.1, final thoughts on the *HDFG Editor* are given. In section 8.2, recommendations for use of the *HDFG Editor* are listed.

8.1 Conclusions

Due to time constraints, it was not possible to finish the *HDFG Editor*À lot of functionality is still missing; not even all must-have functionality has been implemented. However, that does not mean that all the work put into it is wasted. The work done is a good start.

This is partly because the work turned out te be much more than expected. The version 2 file format is complex and the whole repository system had to be created. As such, a lot of time was put into creating the filehandler and the repository. While these are two good classes now, creating them them consumed more time than was planned, leaving less time for other, also crucial parts of the program.

One point where time could have been saved was the choice of the graphics library. While Irrlicht provided everything expected from it, it was still a lot of work to build all the drawing from nothing. Therefore, it would have been better to put that effort into setting up a precreated graph-drawing environment.

This does not mean that all the work put into the *HDFG Editor* is of no use. While the parts mentioned above (the filehandler, repository and the graphics part) took a long time to create and are not completely bug-free yet, they provide a solid basis for the program.

8.2 Recommendations for future use

In its current state, it is not recommended to use the *HDFG Editor* for anything but visualizing version 2 graphs.

If someone picks up the *HDFG Editor* project in the future, it is recommended to use parts of the current code, if not continuing with the whole code. Especially the repository and the methods for loading it should be useful, as they are completely finished and working properly.

Reccommendations for future additions to the application also entail the creation of a graphical user interface for editing the repository. Though it is possible to accomplish this at his point through editing the "repository.txt" file, a user interface would make the process more accessible and more user friendly.

Terms and abbreviations

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CDFG Control Data Flow Graph

DFG Data Flow Graph

DWARV Delft WorkBench Automated Reconfigurable VHDL generator

DWB Delft WorkBench

FPGA Field Programmable Gate Array

GDI+ Graphics Device Interface+

GPP General Purpose Processor

GUI Graphical User Interface

HDFG Hierarchical Data Flow Graph

HDL Hardware Description Language

Irrlicht Engine Open source high performance real-time 3D engine

ISA Instruction Set of an Architecture

Little endian Numeric representation that places least significant byte first

MOLEN Polymorphic processor

MoSCoW document Must have, Should have, Could have, Would have document

MUX Multiplexer

OpenGL Open Graphics Library

SDK Software Development Kit

SDL Simple DirectMedia Layer

SUIF Stanford University Intermediate Format

TCP/IP Transmission Control Protocol / Internet Protocol

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits

VLIW Very Long Instruction Word

WYSIWYG What You See Is What You Get

XREG Exchange Register

References

[1] Bertels, K., Vassiliadis, S., Panainte, E. M., Yankova, Y., Galuzzi, C., Chaves, R. and Kuzmanov, G. (2006). *Developing applications for polymorphic processors: the Delft Workbench*. Tech. rep., Delft University of Technology.

[2] Bruegge, B. and Dutoit, A. (2004). *Object-Oriented Software Engineering*. 2nd ed. Upper Saddle River, NJ: Pearson Education, Inc.

[3] Eclipse. *Eclipse Graphical Editing Framework*. `http://www.eclipse.org/gef`. Visited on April 28th, 2008.

[4] Gebhardt, N. *Irrlicht Engine 1.4 API documentation*. `http://irrlicht.sourceforge.net/docu/index.html`. Visited on May 20th, 2008.

[5] Graphviz. *About Graphviz*. `http://www.graphviz.org/About.php`. Visited on May 20th, 2008.

[6] Graphviz. *The DOT Language*. `http://www.graphviz.org/doc/info/lang.html`. Visited on May 20th, 2008.

[7] Graphviz. *Graphviz - resources*. `http://www.graphviz.org/Resources.php`. Visited on May 20th, 2008.

[8] Hurkmans, T. (2008). Personal Note.

[9] Hurkmans, T. and Kentie, M. (2007). *Ontwerp en implementatie van een Graafviewer en -editor voor de Delft WorkBench*. Tech. rep., Delft University of Technology.

[10] Irrlicht. *Irrlicht: supported drivers*. `http://irrlicht.sourceforge.net/features.html#drivers`. Visited on May 16th, 2008.

[11] Kleinberg, J. and Tardos, E. (2006). *Algorithm Design*. Boston: Pearson Education, Inc.

[12] Krieg-Bruckner, B. *How to connect an application to the uDraw API*. `http://www.informatik.uni-bremen.de/uDrawGraph/en/service/uDG31_doc/api_connect.html`. Visited on May 19th, 2008.

[13] Krieg-Bruckner, B. *uDraw features.* `http://www.informatik.uni-bremen.de/uDrawGraph/en/download/download.html`. Visited on May 19th, 2008.

[14] Lam, M. (1988). Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In: *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 318–328.

[15] Patterson, D. A. and Hennessy, J. L. (2005). *Computer Organization and Design*. 3rd ed. San Francisco: Morgan Kaufmann.

[16] Pauilin, P. and Knight, J. (1989). Force-Directed Scheduling for the Behavioral Synthesis of ASICs. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8.

[17] SDL. *SDL: What Platforms Does It Run On?* `http://www.libsdl.org/intro.en/whatplatforms.html`. Visited on April 24th, 2008.

[18] Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G. and Panainte, E. M. (2004). The Molen Polymorphic Processor. In: *IEEE Transactions on Computers*, 1363–1375.

[19] Vos, M. (2008). Personal Note.

[20] wiki, E. G. *GEF Developer FAQ - Eclipsepedia*. `http://wiki.eclipse.org/index.php/GEF_Developer_FAQ`. Visited on April 28th, 2008.

[21] Wikipedia. *Force-based algorithms*. `http://en.wikipedia.org/wiki/Force-based_algorithms`. Visited on May 2nd, 2008.

[22] Yankova, Y. *DWARV v2.0 Intermediate Representation*. Internal Documentation.

[23] Yankova, Y. (2007). *HDFG v1.0 File Format*. Internal Documentation.

[24] Yankova, Y. (2008). Personal Note.

[25] Yankova, Y., Kuzmanov, G., Bertels, K., Gaydadjiev, G. N., Lu, Y. and Vassiliadis, S. (2007). DWARV: Delft Workbench Automated Reconfigurable VHDL Generator. In: *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, 697–701.

Appendix A MoSCoW document

A MoSCoW document (Must have, Should have, Could have, Would have) presents a way to prioritize the various tasks appointed to a team of programmers[2]. During the process of developing an application many unexpected events may occur that can cause drastic delays. It is important that, should something go amiss, a working application can be delivered upon reaching the deadline. This document separates requirements in categories to discern requirements that are vital to the application from those that provide functionality that can be missed. One should always fully intend to complete both the Must have and Should have categories.

Must have functionality

- Loading binary and text files containing HDFGs.
- Storing HDFGs into a binary or text file.
- Visualizing the loaded graphs.
- Opening subgraphs.
- Creating new graphs.
- Creating new nodes.
- Creating new edges.
- Deleting nodes and edges.
- Modifying the values of existing properties.
- Associating a property sheet with every node and edge. This property sheet has to be displayable as a separate dialog box.

Should have functionality

- Exporting HDFGs to .jpg and .eps files.
- Zooming in and out.
- Merging sub-graphs within the upper-level graph.
- Moving nodes and edges by dragging them on the screen.
- Modifying existing types of nodes, including their properties.
- Modifying existing types of edges, including their properties.
- Loading the repository from file.

Could have functionality

- Applying semantic checks when nodes and edges are added.
- Applying semantic checks upon modification of properties.
- Creating new types of nodes.
- Creating new types of edges.
- Storing the repository to file.
- Importing repository data.

Would have functionality

- Creating a GUI for the repository.

Appendix B Use Cases

The following use cases represent all functionalities that require user interaction.

Use case 1: Loading binary and text files containing HDFGs

Actors: User
Entry condition: None
Flow of events:

1. The user clicks the 'open file' button or chooses the 'open file' option from the file menu.
2. The user uses the file dialog to navigate to the file he/she wants to load.
3. The user clicks the 'open' button
4. The user selects the HDFG representation version number in a dialog box.
5. The contents of the file are displayed.

Exit condition: The selected file is shown on the screen.

Use case 2: Storing HDFGs into a binary or text file

Actors: User
Entry condition: The user has loaded a graph
Flow of events:

1. The user clicks the 'save' button or selects the 'save' option from the file menu.
2. The user uses the file dialog to navigate to the desired location.
3. The user enters the desired file name.
4. The user uses the combo box to select the type of file to export to.
5. The user clicks the 'save' button.

Exit condition: The HDFG has been saved to disk.

Use case 3: Exporting HDFGs to .jpg and .eps files

Actors: User
Entry condition: The user has loaded a graph
Flow of events:

1. The user selects the 'export to picture' option from the file menu.
2. The user uses the file dialog to navigate to the desired location.
3. The user enters the desired file name.
4. The user uses the combo box to select the type of file to export to.
5. The user clicks the 'save' button.

Exit condition: A picture of the graph is stored on the hard drive.

Use case 4: Loading the repository from file

Actors: User
Entry condition: The user has not loaded or created a graph
Flow of events:

1. The user selects 'Load Repository' from the 'Repository' menu.
2. The user highlights a repository file in the file dialog.
3. The use clicks the 'Load' button.

Exit condition: The repository is loaded from file.

Use case 5: Storing the repository to file

Actors: User
Entry condition: The user has loaded a repository from file
Flow of events:

1. The user selects 'Save Repository' from the 'Repository' menu.
2. The user enters a filename in the file dialog.
3. The user clicks the 'Save' button.

Exit condition: The repository is stored to file.

Use case 6: Zooming in and out

Actors: User
Entry condition: The user has loaded a graph
Flow of events:

1. The user clicks the button for zooming in/out.
2. The graph is shown larger/smaller.

Exit condition: The graph is shown larger/smaller.

Use case 7: Opening subgraphs and compound nodes

Actors: User
Entry condition: The user has loaded a graph
Flow of events:

1. The user double-clicks on a compound node.
2. A new tab is created containing the subgraph.

Exit condition: A new tab containing the subgraph has been opened.

Use case 8: Merging subgraphs within the upper-level graph

Actors: User
Entry condition 1: The user has loaded a graph
Flow of events 1:

1. The user selects a compound node.
2. The user clicks the button for exploding a subgraph or selects 'explode' from the right-click menu.
3. The compound node is replaced by the subgraph.

Entry condition 2: The user has loaded a subgraph
Flow of events 2:

1. The user clicks the button for merging the subgraph into the upper-level graph.
2. In the upper-level graph the corresponding compound node is replaced by the subgraph.

Exit condition: The subgraph has been merged into the upper-level graph.

Use case 9: Associating a property sheet with every node and edge

Actors: User
Entry condition: The user has loaded a graph
Flow of events:

1. The user right-clicks on a node or edge and selects 'properties'.
2. A property sheet containing the properties is displayed.

Exit condition: A property sheet is displayed.

Use case 10: Creating a new graph

Actors: User
Entry condition: None
Flow of events:

1. The user presses the 'new graph' button.

2. An empty graph appears.

Exit condition: The user has created a new graph.

Use case 11: Creating a new node

Actors: User

Entry condition 1: The user has loaded a graph

Flow of events 1:

1. The user presses the button for creating a new node.
2. The user clicks at the location desired for the node.
3. A new node appears at the desired location.

Flow of events 2:

1. The user clicks the right mouse button at the location desired for the node.
2. A new node appears at the desired location.

Exit condition: A new node is present in the graph.

Use case 12: Creating a new edge

Actors: User

Entry condition 1: The user has loaded a graph

Flow of events 1:

1. The user clicks the button for creating a new edge.
2. The user fills out a dialog box with the type of edge that is to be created.
3. The user clicks two nodes.
4. A new edge is created between the two clicked nodes.

Flow of events 2:

1. The user selects two nodes.
2. The user clicks the button for creating a new edge.
3. The user fills out a dialog box with the type of edge that is to be created.
4. A new edge is created between the two selected nodes.

Exit condition: A new edge has been created

Use case 13: Moving nodes and edges

Actors: User

Entry condition: The user has loaded a graph

Flow of events:

1. The user selects the nodes and edges to be moved.
2. The user drags the nodes and edges to the desired location.

Exit condition: The nodes and edges have moved to the desired location.

Use case 14: Modifying the values of existing properties

Actors: User

Entry condition: The user has loaded a graph

Flow of events:

1. The user right-clicks a node or edge and selects 'Properties' from the menu.
2. The user fills out the property settings in the dialog.
3. The user clicks 'OK' to change the properties and close the dialog.

Exit condition: The properties of a node have been changed.

Use case 15: Deleting nodes and edges

Actors: User

Entry condition: The user has loaded a graph

Flow of events:

1. The user selects nodes and/or edges.
2. The user clicks the button for deleting nodes and/or edges or presses the delete key.
3. The nodes and/or edges are deleted.

Exit condition: The selected nodes and/or edges are deleted.

Use case 16: Importing custom-created types of nodes, edges and properties from a repository (library)

Actors: User
Entry condition: The user has loaded a graph
Flow of events:

1. The user selects 'Import' option from the 'Repository' menu.
2. The user uses the file dialog to navigate to the repository he/she wants to import.
3. The user clicks the 'open' button.
4. The data from the selected repository is added to the current repository.

Exit condition: The repository is imported.

Use case 17: Creating new types of nodes

Actors: User
Entry condition: None
Flow of events:

1. The user selects 'Node types' from the 'Repository' menu.
2. The user clicks the 'New node type' button in the dialog.
3. The user fills out a dialog box with the settings for the new node type, including properties.
4. The user presses 'OK' to create the new node type and close the dialog box.
5. The user presses 'OK' to close the node types dialog.

Exit condition: A new type of node has been added to the repository.

Use case 18: Creating new types of edges

Actors: User
Entry condition: None
Flow of events:

1. The user selects 'Edge types' from the 'Repository' menu.
2. The user clicks the 'New edge type' button in the dialog.
3. The user fills out a dialog box with the settings for the new edge type, including properties.
4. The user presses 'OK' to create the new edge type and close the dialog box.
5. The user presses 'OK' to close the edge types dialog.

Exit condition: A new type of edge has been added to the repository.

Use case 19: Modifying types of nodes

Actors: User
Entry condition: None
Flow of events:

1. The user selects 'Node types' from the 'Repository' menu.
2. The user selects a node type and clicks the 'Edit' button in the dialog.
3. The user fills out a dialog box with the settings for the node type, including properties.
4. The user presses 'OK' to edit the node type and close the dialog box.

5. The user presses 'OK' to close the node types dialog.

Exit condition: A node has been edited the repository has been updated.

Use case 20: Modifying types of edges

Actors: User
Entry condition: None
Flow of events:

1. The user selects 'Edge types' from the 'Repository' menu.
2. The user selects an edge type and clicks the 'Edit' button in the dialog.
3. The user fills out a dialog box with the settings for the edge type, including properties.
4. The user presses 'OK' to edit the edge type and close the dialog box.
5. The user presses 'OK' to close the edge types dialog.

Exit condition: An edge has been edited and the repository has been updated.

Appendix C Repository description

The repository contains information on the HDFG graph format. The purpose of this is to increase the adaptability of the *HDFG Editor*. If for instance a new node type will be used in the HDFG format one needs only to add said node type to the repository.

The repository is stored in a separate file, located in the same folder as the executable. The filename is `repository.txt` and cannot be changed. At this point it is not possible to load a repository from a different location.

The syntax of the file format is strict. The repository file contains ASCII text and is read line by line, top to bottom. Lines can have a maximum length of 256 characters. Numeric constants can be represented by decimal values or hexadecimal values, when they should be prefixed with `0x`. Examples are `126` for a decimal representation and `0x7E` for a hexadecimal representation of the same number. Empty lines are ignored and single line comments can be placed in the file by starting a line with the # sign.

The repository can be divided into sections. They may not be omitted, nor may their sequence of appearance change. Each section starts with a quantifying number of how many items appear in that section. Should a section contain no items the quantifying number at the start of each section should be set to zero, but not be left out.

1. Data types (including local data types)
2. Constant field sizes for different data types
3. Connection classes
4. Custom field classes
5. Node types
6. Sections

Each section will be described in more detail in the following paragraphs.

Data types

The *HDFG Editor* handles many data types, each of which has a unique numeric representation in the binary file format, see [22]. To increase adaptability these numeric representations are read from the repository so that they can be changed and new ones can be added.

The syntax of the data types section is as follows (Table C.1). The first line contains an integer value of how many data types the repository contains. The second line holds another integer value of how many of these data types are non-local. In other words, the total number of data types minus this number equals the total number of local data types.

Next, the non-local data types are summed. The first line contains a string representing the data type. The second line contains a numeric constant by which the data type can be identified. Such a set of two lines appears for every non-local data type, so i times.

Last, the local data types are summed. Again, the first line contains a string representing the data type. The second line contains a numeric constant. Such a set of two lines appears for every local data type, so $n - i$ times.

Constant field sizes

As the size of a numeric constant varies by data type it is useful to include this information in the repository. This increases adaptability as more sizes can be added and changed. This information is also critical for the parsing of node representations in binary files [22]. For each previously declared data type a constant field size may be specified in this section. It is not obligatory to specify a constant field size for every data type, however, in this case no constants of this type should be used.

Table C.1: Data type description syntax

Recurrence		Line	Description
1		n	Integer representing the total number of data types that follow
		i	Integer representing how many of these data types are non-local
	i	`name`	String naming the non-local data type
		`dt`	Numeric constant representing the non-local data type
	$n-i$	`name`	String naming the local data type
		`dt`	Numeric constant representing the local data type

The syntax of this section is as follows (Table C.2). The first line contains an integer value of how many field sizes the repository contains. Next, the field sizes are listed. The first line of such a list item contains a string, identifying the data type. This string must match one of the earlier specified data types (case insensitive). The second line of such a list item is an integer representing the field size for a constant of this data type. The size is in bytes. This list contains n items.

Table C.2: Constant field size description syntax

Recurrence		Line	Description
1		n	Integer representing the total number of sizes that follow
	n	`name`	String naming the data type
		`s`	Integer representing the constant field size in bytes

Connection classes

Connections between nodes vary in type and representation. Sometimes data edges are specified as an input number for a certain node, sometimes as an output number for a certain node. In the future, new edge types may be added, or current ones redefined. To make this completely adaptable these connection classes have been added to the repository.

The connection classes have a name, a type number and are of a certain connection format. Edges connect two nodes, so the connection format can be represented in four ways. Say that a node representation holds one other node number, an edge can be created from or to that node. A node representation can also hold two node numbers that need be connected, in which case the nodes can be connected either way as well.

The syntax of the connection classes is as follows (Table C.3). First, a line precedes the rest with a number on how many connection classes are available. The connection classes are now listed, the first line containing the name (a string), the second line an integer representing the node type and the third line an enumeration value representing the type. This enumeration has a string representation and can be one of four values. `CONN_FROM` and `CONN_TO` represent connections between a specified node and the context node. The two values correspond to an edge coming from the specified node and going to the specified node respectively. `CONN_FROM_TO` and `CONN_TO_FROM` represent connections between two specified nodes. The two values correspond to an edge leading from the first to the second node and vice versa.

Custom field classes

With adaptability as the major focus point and so few default data types, a set of custom field classes is necessary. There are five supported basic data types: `VALUE_INT`, `VALUE_BOOL`, `VALUE_STRING`, `VALUE_CONSTANT` and `DATA_TYPE`. They represent integers, booleans, strings, numeric constants and data type representations respectively.

Table C.3: Connection class description syntax

Recurrence	Line	Description
1	n	Integer representing the total number of classes that follow
n	`name`	String naming the connection class
	`s`	Integer representing the edge type
	`cform`	Enumeration representing the connection format

The custom field classes allow the creation of combinations of these data types to enable usage of almost any sequence of fields. This greatly increases the readability of the repository, but is also useful for nesting properties. In addition to the data types, the connection classes can also be specified to once again increase the adaptability.

Another powerful feature of custom field classes is the possibility of vectors. In the binary file format, vectors play a significant role. Defining a field class as vector allows for a set of fields to appear a number of times, and be interpreted as one set of fields.

The syntax of the custom field classes is as follows (Table C.4). The section is started with a number identifying the total number of custom field classes in the repository. After this the field classes are listed.

Starting the field class, the first line ontains the name (a string). The second line holds a boolean value identifying the field class as a vector. Next, an integer identifies the total number of fields this field class contains. The fields are then listed.

Each field is represented with two lines. The first line is the field name, a string that will represent the field within the program. The second line is a string representation of the data type or connection class.

Table C.4: Field class description syntax

Recurrence	Line	Description
1	n	Integer representing the total number of classes that follow
n	`name`	String naming the field class
	`vector`	Boolean identifying a vector
	p	Integer representing the number of fields
p	`fname`	String naming the field
	`ftype`	Enumeration representing the field type

Node types

With many different types of nodes and many different binary representations this section is the main reason for the existence of a repository. It is likely that new node types will be added to the HDFG representation or that current node types will change. Each node type has a unique name and unique numeric representation. Furthermore, each node type has a variable number of connections of variable types, as well as a variable number of properties of varying types and natures. A complete overview of node structures and numeric representations can be found in [22].

The syntax of the node type representations is as follows (Table C.5). The first line contains a number identifying the total amount of node types in the repository.

Next, the node types are listed. The first line of a node type contains the node type name. This is a unique string value with which the node type can be identified. The second line contains a numeric representation for this node type. This is also a unique value and is used in the binary file representation of an HDFG.

Next, a line with an enumeration value discribes the node category. The possible values are `CAT_SIMPLE`, `CAT_STORAGE` and `CAT_COMPOUND`. They refer to simple nodes, storage nodes and compound nodes respectively.

Some display properties have to be saved in the repository as well, so the next line holds an enumeration value identifying the shape of the node. Three basic values are `CIRCLE`, `RECTANGLE` and `DIAMOND`, each representing the respective shape. For these three values the next line contains a static symbol that will be displayed each node of this type. The three other possible enumeration values are `CIRCLE_DYN_TEXT`, `RECTANGLE_DYN_TEXT` and `DIAMOND_DYN_TEXT`. For these values, the text on the node is dynamic. The next line now holds an integer value representing the field number which value should be displayed on the node.

The sixth line contains an integer that represents the total number of fields this node type has. The order in which they appear must also be the order in which they occur in the binary file representations. After this the fields are summed.

A field or property, consists of two lines. The first one contains the property name. This name is used to represent the field in the graphical user interface. The second line is enumeration value representing the type of field. This value can either be one of the basic data types, a custom field class or a connection class.

Table C.5: Node type description syntax

Recurrence			Line	Description
1			n	Integer representing the total number of node types that follow
	n		`name`	String naming the node type
			`t`	Numeric constant representing the node type
			`cat`	Enumeration representing the node category
			`shape`	Enumeration representing the node shape
			`disp`	Node display text (value or field number)
			p	Integer representing the number of fields
		p	`fname`	String naming the field
			`ftype`	Enumeration value representing the field type

Sections

After the main graph has been described in a binary HDFG file, several sections are listed with information on storage types and sub-graphs[22]. The format of these sections in the binary files are included in the repository.

The syntax of the section representations is as follows (Table C.6). The first line contains a number identifying the total number of section definitions in the repository.

Subsequently the sections are listed. The first line contains a name with which the section can be identified. The second line contains a numeric representation of this section. This number is used to identify a section type in an HDFG binary file. The third line holds an enumeration value identifying the type of section. Possible values are `SECTION_COMPOUND` for compound node sections and `SECTION_STORAGE` for storage information sections.

Only a few node types belong to a section. An integer value represents the total number of associated node types. The next lines contain string representations of the associated node types.

In order to list the fields of a section, they are preceded by a line containing an integer. This is the total number of fields this section contains. The fields are listed afterwards. A field consists of two lines. The first is a name and the second an enumeration value representing

a basic data type or previously defined field class. Another possible value is `HDFG_GRAPH` representing a subgraph.

Table C.6: Section description syntax

Recurrence	Line	Description
1	n	Integer representing the total number of sections that follow
n	`name`	String naming the section
	`s`	Numeric constant representing the section
	`type`	Enumeration representing the section type
	t	Integer representing the number of node types
t	`ntype`	String value representing the node type
	p	Integer representing the number of fields
p	`fname`	String naming the field
	`ftype`	Enumeration value representing the field type

Appendix D Evaluation details

Evaluation phase 1 step 1

Thank you for participating in this evaluation of the *HDFG Editor*. We would like to ask you to complete a set of tasks. After these you will be prompted to answer some questions.

Tasks

Please complete the following tasks. You can grade the difficulty level of completion with either $-$, $+/-$ or $+$ where $+$ refers to easy and $-$ to hard.

Load an existing v2 graph.	$-$	$+/-$	$+$
Move several nodes around.	$-$	$+/-$	$+$
Modify the values of existing properties.	$-$	$+/-$	$+$
Create a new node.	$-$	$+/-$	$+$
Create a new edge.	$-$	$+/-$	$+$
Delete a node.	$-$	$+/-$	$+$
Delete an edge.	$-$	$+/-$	$+$
Create a new graph.	$-$	$+/-$	$+$
Create two new nodes.	$-$	$+/-$	$+$
Create an edge between the nodes created in the previous step.	$-$	$+/-$	$+$
Switch to the graph loaded in step one.	$-$	$+/-$	$+$
Close this graph, but leave the new one open.	$-$	$+/-$	$+$

Questions

Please answer the following questions.

1. What do you feel is missing in the application?

2. Did the application act the way you suspected it would?

3. Did you encounter any bugs or otherwise unexpected events?

4. Do you have any comments to add?

Thank you for your time! Your remarks are most appreciated.

Evaluation phase 1 step 2

Thank you for participating in this evaluation of the *HDFG Editor*. We would like to ask you to complete a set of tasks. After these you will be prompted to answer some questions.

Tasks

Please complete the following tasks using the instructions provided. You can grade the difficulty level of completion with either $-$, $+/-$ or $+$ where $+$ refers to easy and $-$ to hard.

Load an existing v2 graph. $-$ $+/-$ $+$
Press the toolbar button with the folder icon. A dialog box appears. Select the proper file and press OK.

Move several nodes around. $-$ $+/-$ $+$
Select a node by left-clicking on it. The node will be highlighted. Drag it around using the left mouse button.

Modify the values of existing properties. $-$ $+/-$ $+$
Select a node by left-clicking on it. The node will be highlighted and the property window will be filled with poperties for this node. Change a property value.

Create a new node. $-$ $+/-$ $+$
Right-click anywhere on the canvas. A new node will appear. Try also pressing the toolbar button displaying a circle with a star. A new node will appear in the upper left corner of the canvas.

Create a new edge. $-$ $+/-$ $+$
Drag the right mouse button from one node to another. A new edge will appear from the first to the second node.

Delete a node. $-$ $+/-$ $+$
Select a node by left-clicking on it. The node will be highlighted. Press the 'Delete' key. The node will disappear, together with all connected edges.

Delete an edge. $-$ $+/-$ $+$
Select an edge by left-clicking on it. The edge will be highlighted. Press the 'Delete' key. The edge will disappear.

Create a new graph. $-$ $+/-$ $+$
Press the toolbar button displaying a blank page with a star. A new tab is opened containing an empty canvas.

Create two new nodes. $-$ $+/-$ $+$
Right-click on two locations on the canvas. Two new nodes will be created.

Create an edge between the nodes created in the previous step. $-$ $+/-$ $+$
Drag the right mouse button from the one node to the other. A new edge will be created.

Switch to the graph loaded in step one. $-$ $+/-$ $+$
Left click the leftmost tab displaying the loaded graph name. The current graph will be minimized and the former tab activated.

Close this graph, but leave the new one open. $-$ $+/-$ $+$
Right click the tab to close it.

Questions

Please answer the following questions.

1. What do you feel is missing in the application?

2. Did the application act the way you suspected it would?

3. Did you encounter any bugs or otherwise unexpected events?

4. Do you have any comments to add?

Thank you for your time! Your remarks are most appreciated.

Appendix E User Manual

System Requirements

The *HDFG Editor* will run on any system, with the sole requirement that OpenGL drivers must be installed. For Microsoft Windows the Irrlicht dynamic linked library is required, for linux or macintosh users the library can be compiled using the Irrlicht SDK.

Manual

- **Creating a new graph**
  Click the leftmost button on the toolbar, displaying a blank page. A new tab will open, containing a new, empty graph.
- **Loading a version two HDFG from file**
  Click the button on the toolbar that displays an open folder. Using the file dialog, navigate to the .dfg file you wish to open. Select the file and click 'Open'. The graph will now be loaded.
- **Creating a new node**
  Using the mouse, right click anywhere on an empty space of the graph to create a new node.
- **Creating a new edge**
  Using the mouse, right click a node and drag the mouse to another node. An edge will be created from the first to the second node.
- **Moving a node**
  Using the left mouse button, drag a node from one place to another to move the node.
- **Deleting a node**
  Select a node using the left mouse button, and delete it using the delete key on the keyboard.
- **Deleting an edge**
  Select an edge using the left mouse button and delete it using the delete key on the keyboard.
- **Modifying node or edge properties**
  Select a node or edge using the left mouse button. The property window will display the selected items properties. Using the displayed controls, change property values.
- **Modifying the repository**
  Using a text editor, the repository can be modified. The complete syntax is detailed in Appendix C.

Appendix F CD-ROM