MANAGING DEPENDENCIES IN KNOWLEDGE-BASED SYSTEMS: A GRAPH-BASED APPROACH

Martin Tapankov

THESIS WORK 2009
PRODUCT DEVELOPMENT AND MATERIALS ENGINEERING
MANAGING DEPENDENCIES IN KNOWLEDGE-BASED SYSTEMS: A GRAPH-BASED APPROACH

Martin Tapankov

This thesis work is performed at Jönköping Institute of Technology within the subject area Product Development and Materials Engineering. The work can also be a part of the university’s master’s degree. The author is responsible for the given opinions, conclusions and results.

Supervisor: Fredrik Elgh

Credit points: 30 ECTS credits

Date:

Archive number:
Abstract

In knowledge-based engineering, the inference engine plays an important part in the behaviour of the system. A flexible and adaptive execution scheme allows the designer to experiment with different modes of operation and selecting an appropriate one with respect to the initial data set and the execution goal.

In this project, an extension of an existing research prototype software in the field of knowledge-based engineering will be developed, with the goal of building a reliable and easy to use dependency resolution engine that will replace a less-than-ideal current implementation of the same. A discussion will be included how the knowledge concepts and objects can be represented in an abstract mathematical form, converting at the same time the problem of dependency resolution to a more formally specified one in terms of the data abstraction proposed. Some algorithms and methods that are used to operate on the data set will be discussed from both a theoretical and programming point of view, analysing their complexity, proposing and testing their implementation. Graphical interface controls that can be used to visualize and understand easily the relations in the available knowledge base will be also demonstrated.

The testing and verification of the resulting software will be presented, comparing its behaviour against reference tools serving similar purposes. Methods for validating the consistency of the knowledge base will also be discussed. Finally, the integration of the newly-developed code within the context of the prototype will be discussed, commenting on the new features and functionality gained.

Keywords  dependence resolution, knowledge base, database, KBE, DSM, programming, algorithms, directed acyclic graphs, .NET framework
# Table of Contents

**Abstract**

1. **Introduction**  
   1.1 Background ........................................... 1  
   1.2 Purpose and Goals .................................... 5  
   1.3 Project Scope ......................................... 5  
   1.4 Thesis Outline ....................................... 5  

2. **Theoretical Background**  
   2.1 Knowledge-Based Systems  
      2.1.1 Knowledge Base .................................... 7  
   2.2 Data Abstraction  
      2.2.1 Mathematical Model  
      2.2.2 Computational Abstraction  
   2.3 Problem Abstraction .................................. 14  
   2.4 State Space Search Strategies  
      2.4.1 Brute Force Search  
      2.4.2 Forward Chaining  
      2.4.3 Backward Chaining  
   2.5 Dependency Structure Matrix  

3. **Project Design**  
   3.1 Design Goals ......................................... 20  
   3.2 Development Platform ................................ 20  
   3.3 Integrated Development Environment .................... 21  
   3.4 Source Code Control .................................. 21  
   3.5 Guiding Principles in Software Construction .............. 22  
   3.6 Prior Art ............................................... 22  
   3.7 Third-Party Components ................................ 23  
   3.8 Licensing .............................................. 23  
   3.9 Software Availability .................................. 24  
   3.10 Quality Assurance .................................... 24  

4. **Results** ........................................... 25
## List of Figures

1. Relationships between design variables and knowledge objects ........................... 2
2. Functional Parts of the Kongsberg Prototype for ADAPT Project ......................... 3
3. ProcedoStudio.NET — Main Screen ......................................................... 4
List of Figures

4 ProcedoStudio.NET — Execution Control ............................... 4
5 A sample knowledge base ................................................. 9
6 Relationships between parameters ..................................... 9
7 Relationships between knowledge objects ........................... 10
8 An example directed graph and its corresponding adjacency matrix .... 11
   8.1 Directed graph .................................................. 11
   8.2 Adjacency matrix ................................................ 11
9 Reachability matrix corresponding to the graph shown on Figure 8.1 12
10 Course prerequisites — sample DSM ................................. 17
11 Course prerequisite — partitioned DSM ............................. 18
12 Data Selection view in Debris ........................................ 27
13 Data Validation view in Debris ....................................... 28
14 Execution Control view in Debris .................................... 28
15 DsmView visual component .......................................... 29
   15.1 Original DSM .................................................. 29
   15.2 Optimized and partitioned DSM ................................ 29
16 MatchingGrid visual component ..................................... 30
17 KnowledgeBase Editor ................................................ 31
18 Cyclic graphs .......................................................... 36
   18.1 Simple cyclic graph ............................................ 36
   18.2 Compound cyclic graph ....................................... 36
19 Main Form Screenshots ............................................... 51
   19.1 Data Selection .................................................. 51
   19.2 Data Validation ................................................ 51
20 Course Prerequisites DSM — Comparison between DSM MIT and Debris 52
   20.1 Course Prerequisites — Original .............................. 52
   20.2 Course Prerequisites — DSM MIT ............................. 52
   20.3 Course Prerequisites — Debris ................................ 52
21 Random Items DSM — Comparison between DSM MIT and Debris 53
   21.1 Random Items — Original .................................... 53
   21.2 Random Items — DSM MIT ................................... 53
   21.3 Random Items — Debris ....................................... 53
22 Kongsberg Automotive DSM — Comparison between DSM MIT and Debris 54
List of Algorithms

22.1 Kongsberg Automotive — Original ........................................ 54
22.2 Kongsberg Automotive — DSM MIT .................................... 54
22 Kongsberg Automotive DSM — Comparison between DSM MIT and Debris ........................................ 55
22.3 Kongsberg Automotive — Debris ........................................ 55
23 Fläktwoods DSM — Comparison between DSM MIT and Debris ................................. 55
23.4 Fläktwoods — Original ........................................ 55
23 Fläktwoods DSM — Comparison between DSM MIT and Debris ................................. 56
23.5 Fläktwoods — DSM MIT ........................................ 56
23.6 Fläktwoods — Debris ........................................ 56
24 Execution Control in ProcedoStudio.NET and Debris ........................................ 57
24.7 Knowledge Object sequence obtained with ProcedoStudio.NET ........................................ 57
24.8 Knowledge Object sequence determined by Debris ........................................ 57

List of Code Samples

4.1 SQL Statement matching Knowledge Objects versus Parameters ........................................ 26
4.2 SQL Statement matching Knowledge Objects versus Variables ........................................ 26
4.3 Warshall’s algorithm in VB.NET ........................................ 32
4.4 Warren’s algorithm in VB.NET ........................................ 33
4.5 Topological Sort algorithm in pseudocode ........................................ 33
4.6 DSM Partitioning in pseudocode ........................................ 37
1 Introduction

This section will provide the background and rationale for this thesis work. The project goals, scope and outline will also be discussed.

1.1 Background

This section will provide the context in which the current master project has been developed; the rationale for many design and implementation decisions stem from the requirements and specifics of the project.

**ADAPT Project — Overview** The ADAPT project, as described in (Sunnersjö, 2007), is a joint cooperation effort between Sandvik AB, Thule AB, Kongsberg Automotive AS, Volvo Aero, and Jönköpings Tekniska Högskolan (JTH), and is situated in the field of design automation. The research group addresses two main questions:

- *Management of design knowledge.* The problem of documentation, structuring, validation and traceability of executable design rules to their associated design knowledge, in order to preserve reliability and compliance with company standards. Investigate also how such a system may help with modification of the knowledge base that are consistent with already implemented design knowledge.

- *Multiple knowledge sources and flexible solution search.* How can a computerized design system draw conclusions from multiple and overlapping data sets in order to emulate the design decision process undertaken by human designers? What meta knowledge strategies are relevant for applications in engineering design?

The project’s findings are to be explored by building a methodology to be used in a prototype software application to test and verify the proposals for improved knowledge management and flexibility.

**Design Process Description** In the ADAPT project, the design process is viewed as a set of *knowledge objects* and *design variables*. A design variable can be thought of a single piece of data that describes quantitatively a characteristic of the product, and can be usually adjusted by the product designer. Most, if not all, of those properties are dependent on each other — for example, a component’s mass is directly proportional to its volume. The design variables may range from simple numerical values (integer, real, enumerated sets, engineering quantities), string, boolean, or more complex data types. A set of valid design variables can uniquely describe the features of a product.

The means by which the unknown design variables are calculated are referred to as *knowledge objects*. The knowledge objects can be likened to a black box that
Background

take a set of inputs and produce one or more outputs — the internal operations of each knowledge object are essentially hidden from the calling program. There is no definition how the calculation is performed — it can be a simple algebraic formula, a table lookup, an optimization procedure, user input data, numerical simulation, statistical prediction, or even arbitrary selection. The exact procedure depends on the type of the knowledge object involved and the available tools or external software used that can manipulate the specific object. For example, a typical knowledge object may be a program for numerical estimation of several variables, or a CAD program script file for geometry manipulation. The software controlling the execution of knowledge objects need not know the details of how the different knowledge objects are handled in their respective environments — an interface layer between the calling program and the actual execution software allows transparently handling knowledge objects that are not compatible with one another. The flexibility imparted in this representation scheme allows adding knowledge objects of types that were not envisioned initially — the only thing that is needed is to add methods that function as wrappers around the external tools, performing the input, output and control operations.

An example view of the design process description in graphical form, consisting of knowledge objects and design variables, can be seen on Figure 1 below. The design variables themselves can be used as input parameters to other knowledge objects, evaluating new design variables in the process. Knowledge objects without input ($K_1$ and $K_2$ on the figure below) play the role of customer/designer specification — they are used to provide the starting values of a subset of the design variables without introducing conceptually new entities in the model.

![Diagram](image)

*Figure 1: Relationships between design variables and knowledge objects*

In order to evaluate all the design variables (which are the real-world data that the user is interested in), the knowledge objects need to be evaluated in a specific order. However, since the dependencies between the knowledge objects and the variables can be altered at any time, the exact order of execution cannot be defined at design time, and can change by introducing new relations, design variables and knowledge objects. Thus, a flexible method is required that is capable of finding an order of execution of the available knowledge objects at runtime, ensuring that
each and every knowledge object has satisfied dependencies — that is, all of its input parameters are already known and evaluated.

**ProcedoStudio.NET** A prototype software (called ProcedoStudio.NET) has been developed as a part of the ADAPT project, to test and validate the concepts, ideas and methodology developed. The program uses as a test example a knowledge base for heating components in car seats, developed in collaboration with Kongsberg AB by JTH researchers. ProcedoStudio.NET aims at resolving the knowledge object dependencies at runtime by resolving the dependencies between them. The software links to other commercial and university-developed software, aiming at bringing together the functionality of several tools under one place of control. A view of the main components of the system can be seen on Figure 2 below:

![Functional Parts of the Kongsberg Prototype for ADAPT Project](image)

**Figure 2: Functional Parts of the Kongsberg Prototype for ADAPT Project** *(Elgh, 2008)*

The application developed uses a database backend for persistent information storage and retrieval, presenting the data to the user on demand; the database consists of information about a product’s structure, design variables, their calculated values and relationships between them. Additionally, the database stores information about the separate projects (that is, the separate data sets the program has evaluated) already executed, both for archival and reference purposes.

The application connects to a database at startup, retrieving the already executed projects, and presenting a choice for reviewing an old project, or creating a new one (Figure 3). Upon starting a new project, the user is required to specify the location of the input data required — in the specific case, a set of MathCAD sheets — that are used to provide values for the input parameters. The program may then be run, and will show the knowledge objects in order in which they were executed (Figure 24.7).
In the current version of the ADAPT project, the execution control (the inference engine) is realized using the KnowledgeWare workbench in CATIA, which is capable of serving as a primitive rule-inference system. This solution does not allow truly automatic evaluation of the knowledge object sequence, since it requires some tedious manual work in advance to prepare the execution template in CATIA for each conceptually new knowledge base. Hence, it was considered advantageous to develop an alternative inference engine to streamline and simplify the execution control of the knowledge objects, and this is one of the main goals of the this thesis work. Additionally, some graphical components and controls that help the knowledge engineer to see and understand the relationships between the entities in the knowledge base was also considered to be of value.
1.2 Purpose and Goals

The project is aimed at completing the following goals and tasks:

- Explore the problem of dependency resolution in terms of data abstraction, representation, and available algorithms.
- Propose software implementations (if more than one possibility is available) of the dependency resolution problem.
- Build a graphical application (or graphical interface components) that can be used to visualize the structure and the relations of the data in the knowledge base.
- Test the results from implemented software against existing tools with comparable functionality.
- Integrate the developed software in the existing ProcedoStudio.NET, eliminating completely the dependence on CATIA for determining the order of execution of knowledge objects.

1.3 Project Scope

The project is not aimed at producing a fully-working and tested software — it is intended to serve as a proof-of-concept that the ideas and methods used are indeed possible to develop and implement in a program. To this end, no software assurance of any kind is to be expected, although great care would be taken to ensure that the core functionality of the program is accurate and works reliably against different data sets.

The software development is limited by the constraints that the ADAPT project, and in particular ProcedoStudio.NET has, in terms of software platform, programming language, development environment and database format and schemas. These specific constraints will be further discussed in section 3.

1.4 Thesis Outline

The thesis is logically organized in several parts. In the current introductory section, the project background, goals and scope have been described.

In section 2, a more thorough theoretical discussion on the main ideas and concepts behind the ADAPT project — knowledge based systems, data abstraction and representation, state space search and dependency structure matrices, are discussed.

Section 3 deals with the overall design of the project — the specified design goals, the selected development platforms and tools, code licensing and availability issues. An investigation of available software solutions that feature functionality
similar to the one the project is to have is also included. A brief discussion of the used third-party components can be found as well.

Section 4 is dedicated exclusively to the results achieved in this project, both in terms of theory (algorithms and their design), and in practical implementation. Some testing examples are included as well, comparing the output of the program with tools with similar functionality. The integration of the developed software in the ProcedoStudio.NET application is also described.

In section 5, the conclusions of the performed work are summarized and related to the initially-stated goals of the project.

Section 6 discusses some future directions in which the project may be developed in.

Finally, several appendices are included as well — a short formal description of some graph theory concepts, a brief introduction to computational complexity, representative algorithm code samples, and screenshots of the software in action.
2 Theoretical Background

This section will provide more information about the theoretical knowledge on which the implementation of the project will be based. A brief overview of some concepts related to knowledge-based engineering will be presented, as a basis for describing the actual practical problem in a more formal manner. Additionally, the data abstraction used in the project is included, as well as some techniques pertaining to dependency resolution using this representation. Finally, a quick introduction to dependency structure matrices as a tool to visualize parameter dependencies is also available.

2.1 Knowledge-Based Systems

In traditional programming, the domain knowledge and the controlling logic are more or less mixed together in the code — the software engineer explicitly specifies the domain knowledge inside the program. This imposes significant maintenance overhead when the domain knowledge changes frequently, or when the number of rules, relationships and constraints is significant. Hence, for such applications, a clear separation between knowledge and control is required, to allow changes and improvements in both without interference between the two.

Knowledge-based systems (KBS) aim at exactly that. Essentially, they consist of two parts — a knowledge base, which contains only the information about the domain, and an inference engine, which determines the control flow and how the domain knowledge is to be applied, modified and contributed to. Separating knowledge from control allows much more flexibility in defining new facts, rules and relationships, or altering existing ones. The inference engine is more or less static during the development and use — once completed, the inference engine can process completely different data sets. [Hopgood 2001]

2.1.1 Knowledge Base

In general, the knowledge base is a collection of rules and facts, ranging from simple relationships to complex rules, dependencies and structures, and these are collectively referred to as knowledge. The representation details of the knowledge in the knowledge base are largely irrelevant to its behaviour — it is a matter of parsing correctly the knowledge to and from data structures that are convenient to operate on from a programming point of view.

In this work, the knowledge base will be described as a collection of knowledge objects, each of which having a set of input parameters and output variables. The variables and parameters will be collectively referred to as knowledge items, without making explicit distinction between them, as they may serve as both in the context of different knowledge objects.

A knowledge object is said to provide a knowledge item if it is an output variable
for this object, and to depend on a knowledge item if the latter is listed as one of its input parameters.

When a knowledge object is executed, the knowledge items that it provides are evaluated. The execution can only take place when all the dependencies, or input parameters, are satisfied (i.e. they are known or have been previously evaluated).

Unless otherwise stated, the term parameter will refer to a knowledge item that is an input to a knowledge object (that is, its dependency), while variable will be used to denote an entity that is a result of a knowledge object invocation (i.e. provided by the knowledge object).

2.2 Data Abstraction

The knowledge base structure presented above is convenient from a practical point of view. However, to embody these ideas and concepts in a computer program, the knowledge base concepts must be converted to a more formal and abstract representation that is easy and convenient to operate on programmatically.

2.2.1 Mathematical Model

This subsection will discuss the formal mathematical formulation of the problem, introducing the basic structures that will be used to represent the knowledge base hierarchy.

Knowledge Base Representation One common way of representing different configurations of objects and connections is using a graph model. Graphs can describe electrical circuits, roadways, organic compounds, ecosystems, database relationships, et cetera (Gross and Yellen, 1999). In practice, every structure that can be represented by a set of objects with relationships between them can be modelled by a graph — and such is the case with the knowledge base used in this thesis. The knowledge objects and design parameters can be represented with nodes in a graph, and the arcs will depict their explicit dependencies one on another (both “provide” and ”depend” relationships).

\footnote{Please note that the terms variable and parameter have different meaning when one talks about programming — parameters usually refer to the formal parameters of a routine, while variables are the data entities that hold information during the lifetime of the program.}

\footnote{Some of the most important properties of graphs as related to this work are defined in Appendix A. Most of the terminology will be used without prior formal definition or explanation, and the reader is expected to look up an unknown term in the appendix, or a relevant textbook.}
Consider, for example, the following simple knowledge base:

![Figure 5: A sample knowledge base](image1)

Here, $K_i$ represents knowledge objects, and $P_j$ stands for parameters, or knowledge items. Since dependencies are one-way relationships (as opposed to constraints, for example), the edges in the graph have assigned directions — if item $b$ depends on item $a$, this is represented by an arrow whose tail points at $a$, and whose head points at $b$. Graphs that have such one-way relationships are called directed graphs.

This structure maps directly to the knowledge base hierarchy that is used in the product — however, from an abstraction point of view, it is less-than-ideal, the reason being that the different kind of entities (knowledge objects and knowledge items) are mixed in one and the same graph — ideally, those should be separated in order to process them more effectively and without introducing unnecessary complexity.

**Parameter Dependencies** One way of reducing the complexity is expressing the same relationships in terms of parameters only — since they are really what connects the separate knowledge objects together. It is therefore possible to eliminate the knowledge objects from the graph, leaving only the parameters and their direct relationships, as shown on Figure 6 below. The knowledge objects are implicitly defined in this graph, as a set of arcs that link their inputs and outputs.

![Figure 6: Relationships between parameters](image2)
As the knowledge objects define the mapping between the input parameters and output variables, the relationships between those are defined within the context of the knowledge object itself. In many cases (such as this particular project), the internal workings of a knowledge object are either unknown or irrelevant — they can be regarded as black boxes with inputs and outputs, with no information how the latter relate to the former. In such cases, the inputs can be mapped to the outputs in such a way that every output depends on every input. This arrangement is known in graph theory as complete bipartite graph. On other occasions, the explicit dependencies between the parameters are known in advance, and in such cases these are to be used instead, forming instead an incomplete (as opposed to complete) bipartite graph.

**Knowledge Objects Dependencies**  
Alternatively, the knowledge items may be eliminated from the graph, leaving only dependencies between the knowledge objects. This arrangement can be thought of as data flow graph, as the links between the separate knowledge objects are the data (the design parameters) that passes through them. This has been done on Figure 7 below. The data source knowledge objects represent the customer specification with the starting set of input parameters.

![Figure 7: Relationships between knowledge objects](image)

One can see that the relationships between knowledge objects are not as simple and straight-forward as in the case of parameters, because there is no direct mapping between an graph entity (such as a node or arc in a graph) and a knowledge entity (be it a knowledge object or a design parameter). The possibility for more than one relationship between two entries in such a graph makes its processing more difficult from programming and mathematics point of view, and thus this representation is deemed unusable. It can, however, be used for representation purposes to visualize the flow of data between the knowledge objects.
2.2.2 Computational Abstraction

This subsection will present some graph representations that are commonly used in computer science. Some methods for manipulating these data structures will also be discussed.

**Adjacency Matrix** The digraph model of the problem is a useful mathematical construct for depicting the relationships between the knowledge objects and knowledge items, but is of little use for computational purposes. Hence, the graph must be modelled by using a data structure that is simple to operate on by means of programming.

A common representation of digraphs (and graphs, for that matter) is the adjacency matrix (usually denoted by $A_D$ or $M$). The adjacency matrix has the same dimensions as the number of nodes in the graph, and the value of its cells indicate the presence or absence of a relationship between the nodes with the corresponding row and column ids. The adjacency matrix and the underlying graph are in one-to-one relation — each graph has a single adjacency matrix, and each adjacency matrix correspond to exactly one graph (Figure 8).

![Directed graph and its corresponding adjacency matrix](image)

*Figure 8: An example directed graph and its corresponding adjacency matrix*

The parameter dependency graph that will be represented by an adjacency matrix imposes unique restrictions on it, namely:

- In case of a digraph representing a dependency structure, a self-loop on a node would indicate that the corresponding item depend on itself — an impossible situation. Thus, the adjacency matrix of the digraph must have a zero trace vector (i.e. zero elements on the main diagonal).
• Between each two connected knowledge items \(a\) and \(b\), there must be at most one directed edge from \(a\) to \(b\), and at most one from \(b\) to \(a\).\(^3\)

• For each two elements \(a, b\) in the graph, if there’s a relationship between \(a\) and \(b\) such that \(a\) depends on \(b\), then the corresponding entry in the adjacency matrix \(M[a, b] = 1\). Otherwise, \(M[a, b] = 0\).

These restrictions imply that the adjacency matrix of the problem will be a *binary matrix* — the only allowed values in it are zero and one.

**Reachability Matrix** While the adjacency matrix represents only the *direct* connections between the nodes, it cannot show the indirect dependencies between the knowledge items. The *reachability matrix* (denoted by \(M^*\)) can be used to visualize all the direct and indirect relationships between the nodes in a digraph. The reachability matrix that corresponds to the graph shown on Figure 8.1 is shown on Figure 9.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 9: Reachability matrix*

It should be noted that while every digraph has a single reachability matrix, the inverse is not true — there might be graphs which share one and the same reachability matrix, but are topologically different. This problem is further discussed in section 4.2.4.

The complexity class\(^4\) of a straight-forward matrix multiplication is \(O(n^2)\). Having matrices of higher powers (up to \(n\)) raises the complexity for \(M^n\) to at least \(O(n \log_2 n)\). Having to calculate the Boolean disjunction between the power matrices will raise the complexity even more, making the calculation of the reachability matrix a particularly computationally-intensive process.

\(^3\)If both directed edges are present, that implies a circular dependency between \(a\) and \(b\).

\(^4\)See Appendix B for a brief introduction to computational complexity.
Warshall’s Algorithm  A more efficient algorithm for calculating the reachability matrix was originally presented in (Warshall, 1962), and is usually referred to in literature as Warshall’s algorithm. The algorithm is defined in the following way:

Algorithm 2.1 Warshall’s Algorithm

1. Set $M^* = M$.
2. Set $i = 1$.
3. $(\forall j : m^*_{ji} = 1) (\forall k) set \ m^*_{jk} = m^*_{jk} \lor m^*_{ik}$.
4. Increment $i$ by 1.
5. If $i \leq n$, go to step 3; otherwise stop.

The complexity class for the algorithm is $\mathcal{O}(n^3)$ — the incrementation between step 2 and 4 is a linear operation and can be done in $\mathcal{O}(n)$, while step 3 is quadratic ($\mathcal{O}(n^2)$), as $k$ and $j$ are bounded between 1 and $n$. The proof of the algorithm is too long to be presented and elaborated upon here, but can be found in (Warshall, 1962).

Warren’s Algorithm  An improvement over Warshall’s algorithm has been proposed and proved in (Warren Jr., 1975). The algorithm is slightly more sophisticated, and looks as follows:

Algorithm 2.2 Warren’s Algorithm

1. Do Steps 2–3 for $i = 2, 3, \ldots, n$.
2. Do Step 3 for $j = 1, 2, \ldots, i - 1$.
3. If $M(i, j) = 1$, set $M(i, \ast) = M(i, \ast) \lor M(j, \ast)$.
4. Do Steps 5–6 for $i = 1, 1, \ldots, n - 1$.
5. Do Step 6 for $j = i + 1, i + 2, \ldots, n$.
6. If $M(i, j) = 1$, set $M(i, \ast) = M(i, \ast) \lor M(j, \ast)$.

This algorithm has a worst-case complexity of $\mathcal{O}(n^3)$ — the same as Warshall’s algorithm. However, the best-case complexity is only $\mathcal{O}(cn^2)$. The worst case is approached with highly connected graphs, while the best case applies to very scarce adjacency matrices. In a real-world scenario of variable dependencies, it is unlikely that the resulting graph will be dense — in fact, there’s a very high probability that the opposite is true, thus bringing the complexity to a much more manageable level of $\mathcal{O}(cn^2)$, where $c << n$. 
It is not probable that an algorithm with complexity less than $O(n^2)$ can be developed for the general case — even the simplest matrix traversal cell-by-cell takes $O(n^2)$ time, hence it is the theoretical minimum for a reachability matrix computation.

### 2.3 Problem Abstraction

Abstracting the problem representation requires an abstraction of its definition to match the new data structures. To this end, the initial problem of dependency resolution between parameters and knowledge objects can be reduced to the following:

**Definition.** Given is a finite set $S$ of $n$ objects, with precedence relations of the type $x \prec y$. Find a topological sort (see Appendix A) of $S$.

Here, the dependencies between the objects can be thought of as a precedence relation, in the sense that if $x$ depends on $y$, it must be evaluated before it. The topological sort in the graph would give a sequence for each no object is preceded by one that depends on it.

As the problem naturally contains a finite number of elements, the possible orderings between those are also finite — they are simply the number of permutations of $n$ elements, namely $P_n = n!$. Hence, the solution of the problem would be finding a sequence that satisfies the condition given in the definition above. The possible strategies to achieve that will be elaborated in the following section.

### 2.4 State Space Search Strategies

As already discussed, the problem has been reduced to finding a sequence of certain properties among all the possible (a finite number) such sequences. If all the separate sequences are represented by a node in a graph, such graph is called state space representation of the problem, with the nodes being the “states” (Luger, 2005, p. 87). The nodes are connected with arcs which define the steps in the problem-solving process. One or more initial states are also defined, as well as goal conditions which are the solution to the problem. The problem-solving problem can then be converted to a search process for a solution path from an initial state to a goal state.

There are three main search strategies that can be used to drive the discovery process — brute force search, goal-driven search and data-driven search. Each of these will be discussed in turn below.

#### 2.4.1 Brute Force Search

Brute force search can hardly be called a strategy on its own — it is the process of simply evaluating all the possible combinations of variables in the search space,
and indeed a solution is guaranteed to be found. For a small number of variables involved, this may seem a reasonable choice. However, with a simple calculation one might prove that the number of possible calculations of $n$ objects is $N = n!$ — this is simply the number of permutations of $n$ objects. Even if one can assume that statistically a result could be found in $N/2$ trials (possibly even less, if the problem has more than one solution), that still has a complexity class of $O(n!)$. 

At this point, the brute force search should be discarded as an unsuitable strategy for even small non-trivial problems, as the search space expands exponentially with linear increase of the variables. Therefore, a better strategies must be devised, which will be presented in the following sections.

2.4.2 Forward Chaining

In forward chaining (also referred to as data-driven search), the system does not have a specific objective that need to be achieved, other than discovering (in this case — evaluating) as much information as possible. The objects are evaluated as soon as their dependencies have been resolved. The system is unable to “look in the future” and predict whether a specific piece of information will be indeed useful. Instead, it reveals all the possible data, in the hope that some useful knowledge will be discovered in the process. In a nutshell, forward chaining starts with the known facts of the problem (in this case, the initial state), and the rules for obtaining new facts (the precedence operator) that can lead to the goal (Hopgood, 2001, p. 8).

Topological Sorting The topological sort algorithm is the straight-forward application of the problem definition already described. It computes a linear sequence of a directed acyclic graph constrained by the partial orderings of its vertices; that is, a labelling $1, 2, 3, \ldots, n$ for the vertices such that for any directed arc $uv$, with vertices $u$ labelled as $i$, and $v$ labelled as $j$, $i < j$. More informally, the topological sort finds a way to arrange the variables in such a way that no variable is used before it has been provided. The algorithm for topological sorting is well-known and documented: see for example (Knuth 1973, p. 258), (Haggarty 2002, p. 151) or (Gross and Yellen 1999, p. 373). The core idea is finding an object that is not preceded by any other in the graph (such object is guaranteed to exist in a directed acyclic graph), and then remove it from the graph. The resulting graph is also partially ordered, and the process is repeated until the graph becomes empty. The elements taken away from the set are added to a linear sequence in order of their elimination. In the particular data structures used in the program, the node relations are stored in two separate sets — an antecedent set and a reachability set, and are adjusted after each node elimination.

The algorithm for topological sort can be described in the following way (Gross and Yellen 1999, p. 375):
State Space Search Strategies

Algorithm 2.3  Topological Sorting
For $i=1$ to $n$

   Let $s_i$ be a minimal element of poset $(S, \prec)$.
   
   $S = S - \{s_i\}$

Return $\langle s_1, s_2, \ldots, s_n \rangle$

It should be noted that this algorithm is applicable only for acyclic graphs — property 2 of the partial ordering exclude the possibility of closed paths in the graph. In practice, an implementation of this algorithm will fall into an endless loop if it encounters a cyclic dependency in the graph, since at a certain step there won’t be any objects with fully-satisfied dependencies.

2.4.3 Backward Chaining

*Backward chaining* (also known as *goal-driven* search) has a specific objective set in advance — a non-empty subset of the objects that need to be evaluated. The system then tries to discover what other objects need to be evaluated in order to find the ones that are looked for. Goal-driven search delivers a more focused solution, disregarding information that is not relevant to the searched set, taking into account only the knowledge that is needed to arrive at the objective, thus making it more appropriate for very large solution spaces, only small portion of which are deemed to be required to reach a specific solution [Hopgood 2001, p. 9].

**Backtracking** The topological sort algorithm is not the only way by which a topologically sorted sequence of a digraph can be obtained — it is possible to start from a given goal state, and then “backtrack” to an initial set. The algorithm can be summarized as shown below:

**Algorithm 2.4** Backtracking Algorithm

1. Add the goal set to the sequence of evaluated objects
2. Find the antecedents of the given goal set.
3. Add those to the sequence of evaluated objects, if not already on the list
4. Substitute the goal set with the antecedent set.
5. Loop through step 2 unless the goal set is empty.

Reversing the discovered sequence will produce the solution path from the initial state to the given goal state.
2.5 Dependency Structure Matrix

*Dependency Structure Matrix* (DSM), also known as *Design Structure Matrix*, is a representation and analysis tool for system modelling (Browning [2001]), capable of visualizing complicated dependencies, including feedbacks and coupled relations. DSM is used in various areas of technology — it has been successfully applied in project management and task scheduling (Chen [2003]), product development (Helo [2006]), supply-chain management (Chen and Huang [2007]) and software architecture analysis, among others.

The DSM displays the relationships between components in a compact matrix form. For example, consider the following simple DSM, representing the course prerequisites in a university:

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>ID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced biotechnology</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biotechnology</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cell biology</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNA structures</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enzyme activity</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Food science</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Genetic engineering</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Human biology</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 10: A sample DSM showing course prerequisites in a university program. Example taken from [Haggarty, 2002, p. 151].*

Every filled cell not on the main diagonal represent a dependency between the entities with the corresponding labels; for instance, the course *Genetic engineering* is dependent on *Cell biology*. One might find similarities between DSMs and adjacency matrices of digraphs, and the DSM may indeed be the transpose of the adjacency matrix (however, it may as well be a reachability matrix, and the DSM behaviour would not change).

The primary use of DSM is not simply visualizing dependency information, but also sorting through the components and finding an order in which no item appears before its dependencies. Since the as-input underlying matrix will rarely conform to this condition, it needs to be modified in such a way that this becomes true. This procedure is referred to as *partitioning* of the DSM. Essentially, the partitioning is simultaneously rearranging the columns and the rows in the matrix together with their indices. For example, the partitioned DSM of the one shown on Figure 10 is presented below:
In the figure, the different components have been combined in separate levels. Within each level, all items are independent one from another, and can be evaluated in parallel, and in any order.

One might observe from the figure that the shown ordering of the courses is consistent with the predefined prerequisite (dependencies). Note that the given sequence is by no means unique — in the general case, an item may be placed on different levels without necessarily violating the dependency constraints.

**Partitioning Algorithm** A popular DSM partitioning method has been presented in (Warfield, 1973). The algorithm presented is essentially a variation of the topology sorting of a digraph, with one important difference — the author uses the reachability matrix to solve the dependency sequence, and cycles in the digraph are transparently processed without special treatment. For acyclic digraphs, the adjacency matrix is sufficient to partition the DSM correctly, however.

**Algorithm 2.5 DSM Partitioning Algorithm**

1. Create a new partition level.
2. Calculate the reachability and antecedent sets $R(s)$ and $A(s)$.
3. For each element in the DSM, calculate the set product $R(s)A(s)$.
4. If $R(s)A(s) = R(s)$, add the element $s$ to the current level.
5. Remove the element $s$ from the list, and all references to it from the reachability and antecedent sets of all other elements.
6. Repeat from step 1, if the item list is not empty.

The antecedent set $A(s)$ is the set of row indices of non-zero elements in column $s$, while the reachability set $R(s)$ is the set of the column indices of the non-zero
elements in row $s$. The condition $R(s)A(s) = R(s)$, if true, means that element $s$
does not depend on any other element that is still in the list (it may, however,
depend on elements in the current level, in which case there is a circular depen-
dencies in the DSM).

The algorithm is capable of processing loops in the DSM, although this is not
immediately obvious. For example, consider two nodes $i$ and $j$ that are involved
in a loop. The reachability matrix will then contain non-zero elements at $M^*(i, j)$
and $M^*(j, i)$. Hence, item $j$ will appear in both the antecedent and reachability
sets for $i$, and vice versa. Therefore, when the set intersection is calculated, these
items will be both eliminated from the pool of unevaluated entities at the same
step. This, of course, extends to cycles containing any number of items, since all
the nodes in the cycle will appear in each others’ antecedent and reachability sets.

An implementation of the partitioning algorithm is proposed in Section 4.2.5.
3 Project Design

In this section, the infrastructure of the project implementation will be revealed, including the topics of selecting a development platform, integrated development environment.

3.1 Design Goals

Considering the limitations of the current project implementation, cited in section [1] the following design goals have been set:

- The newly-designed system should be capable of communicating with the database, and retrieve the necessary information to infer the relationships between the knowledge objects and the design parameters

- Some means of visualizing these dependencies should be presented to the user, in order to allow him/her to add, change and remove information in a straight-forward manner

- Methods for dependency resolution must be devised that are capable of ordering the knowledge objects in such a way as to ensure that all their dependencies are satisfied. These methods should be easy to use from a programming point of view, and are aimed to replace the current CATIA-based rule inference mechanism.

3.2 Development Platform

The current version of ProcedoStudio.NET is developed in Microsoft .NET Framework 2.0, and it is reasonable to build the program for the current project in the same environment from software integration point of view.

The .NET Framework  The main design goals of the .NET framework are component infrastructure, language integration, Internet interoperation, simple development, reliability and security [Tai and Lam 2002].

The most important feature of the .NET framework is the CLR — a runtime engine that ensures the activation of objects; loads required classes; enforces security checks; performs memory management, just-in-time compilation, execution and garbage collection. CLR is an implementation of the ECMA-335 Common Language Infrastructure (CLI) standard [ECMA International 2006], similar in operation and concept to the Java Virtual Machine (JVM). The .NET-supported languages are compiled to bytecode and at runtime the bytecode is converted to code native to the operating system. Since all .NET languages support one and the same set of available types, classes and methods, they generate equivalent bytecode during compilation. Hence, the compiled binary components are completely compatible one with another, regardless of the programming language that
have been used, thus allowing the mixing of sources from separate languages in one and the same project.

A full discussion of the .NET environment is beyond the scope of this thesis — one might consider sources such as [Tai and Lam 2002] for more elaborate discussion of .NET features.

3.3 Integrated Development Environment

The selection of the .NET framework more or less predetermines the choice among the Integrated Development Environments (IDE). Visual Studio is provided by Microsoft and targets specifically .NET development, among its other functionality. It should be noted that several other alternative IDEs are available (for example, SharpDevelop [5]) that target .NET development, and are free and open-source. However, their capabilities are limited in some aspects compared to Visual Studio, but may be a compelling alternative from a cost point of view.

In this project, Visual Studio 2005 is the selected IDE.

3.4 Source Code Control

Software source and revision control has proved to be a critical part of contemporary software development life-cycle, and are the basis on which other functions of Software Configuration Management systems (SCM) are built. With version management, several versions of the source code is stored, essentially a snapshot of the current development tree. Some of the most important functions of revision control systems are creating and new versions, identifying changes to the components of a version, merging two separate versions, finding the differences between two separate versions of a file or set of files, reverting to a previous version, branching to a new development tree, etc. [Leon 2005].

The advantages of full-scale SCM system are most certainly overweighted by the overhead it requires for such a small project, but a source code control tool is an indispensable part of the development process for non-trivial projects. Additionally, version control systems are of assistance during the deployment phase of a software product, as the ability to have simultaneously several build configurations (stable version and testing/debugging version, for example) contribute to the efficiency of the software development process. Several proprietary and open-source alternatives exist, each focused at different auditory, project scale, and provided functionality.

Subversion [6] (commonly abbreviated as SVN) is an open-source, cross-platform, centralized source control system that is used in multitude of small and medium-sized projects. It is available for different operating system (including Linux,
MacOS X and Windows). Additionally, there is also a Visual Studio connector to SVN (AnkhSVN[7]) that simplifies building and managing the source by providing these capabilities transparently and seamlessly inside the IDE itself. Furthermore, the hosting provider of the project, SourceForge.net, also gives a possibility to manage a SVN repository of the source code online. These features are sufficiently compelling in favour of SVN as the source code revision control system that will be used.

3.5 Guiding Principles in Software Construction

Some of the most important principles of software development, according to (McConnell, 2004), are minimal complexity, ease of maintenance, loose coupling, extensibility, reusability and portability. As the system will be further integrated into an already developed program, these principles must be taken into consideration to allow seamless and straightforward integration of the resulting software in ProcedoStudio.NET. The main goal will be to produce a modular design with high level of abstraction and well-defined classes and objects that correspond directly to less-abstract real-world entities, in the spirit of the object-oriented programming paradigm.

3.6 Prior Art

As one of the central tenets of contemporary software engineering is code reuse, an effort has been made to research the availability of already developed software solutions that can eliminate redundant and unnecessary coding and testing. While searching, two important criteria has been set up, stemming from the nature of the project:

- **Source Code Availability**. An important issue in every software project that uses third-party components is the redistribution and usage rights granted by the attached software license or EULA. Most, if not all, commercial software developed restricts at least its free redistribution, and is available as a limited time trial. That would severely restrict the functionality of the program to be developed, as well as its applicability — it is unreasonable to expect that a user would consider paying for third-party software simply to use a prototype program like the one developed during this project.

  That being said, software released with less-restrictive terms of use and redistribution (such as most open-source software), will be significantly favoured and preferred.

- **Platform Compatibility**. Due to the fact that the current project is developed using the .NET framework in Windows environment, the selected solution should be at least directly compatible in binary form (for example,
a .NET assembly, or a COM object library). Software that requires complex bridging to .NET (written in Java, or targeting alternative operating system) will have to provide exceptional functionality to be considered.

There are several commercial software solutions that implement and use DSM for data visualization in the fields of product development and software architecture (see, for example, the stand-alone tools listed at http://www.dsmweb.org/). However, the software listed uses DSM as a side function, not as its main functionality. Additionally, all of those are commercial offerings, with free and limited trial versions, and paid-for real product, thus making them unsuitable for use in the project.

As far as open-source and freely available tools are concerned, the offerings are few (for example, dtangler, Antares DSM, or jDSM). All these programs (which are at different stages of feature completeness and stability) are written in Java, and as already explained above, bridging the code to .NET is a significant work by itself, and it would be much easier to simply develop a new application than to bridge architectures that are not directly interoperable.

The search conducted did not produce a single tool that is capable of partitioning and visualizing DSMs, and that meet the criteria outlined above, thus requiring the development of a custom DSM control that can provide the necessary functionality for visualizing design parameters and their relations.

3.7 Third-Party Components

The developed program uses a numerical library called ILNumerics.Net for the matrix classes and related methods it provides. The library is open-source software, and is provided free of charge for use in both commercial and non-commercial environments, licensed under the terms of LGPL. The license does not place restrictions on the use of the software, linking against third-party components released under incompatible licenses, or distributing it with such components, thus making it suitable to use together with the proprietary .NET framework, and distribute it with the developed software.

3.8 Licensing

As an university project, the author believes that putting the source code under a liberal software license with minimal restrictions to use, modify and redistribute the program and derivatives aligns well with the openness and publicity inherent to academic institutions. Hence, the source code will be placed under the open-source BSD license, allowing commercial and non-commercial use, modification

http://www.dsmweb.org/
https://sourceforge.net/projects/antaresdsm/
https://sourceforge.net/projects/jdsm/
http://ilnumerics.net
http://www.gnu.org/licenses/lgpl.html
and redistribution, with a requirement for attribution to the original author. The full text of the license conditions is available from http://www.opensource.org/licenses/bsd-license.php

3.9 Software Availability

The program developed in this project is available at http://debris-kbe.sourceforge.net/, free to use, modify and redistribute, provided that the minimum licensing conditions are met, as described in section 3.8.

3.10 Quality Assurance

The software produced alongside this thesis did not undergo a thorough testing and quality control — it is intended to be a proof-of-concept prototype, showcasing the main functionality and features desired without committing to any assurances to fit a particular purpose. However, care should be taken to ensure that the core functionality works as reliably and accurately, while little or no attention will be paid to an occasional graphical interface inconsistency. The program assumes in some places that the data it is supplied is both accurate and available and input testing and validation functionality is to be added sparingly.
4 Results

In this section, the application of the concepts developed in the theoretical background will be discussed. Some sample algorithm implementations will be shown. Additionally, the developed program will be tested against alternative solutions to verify the correctness of the proposed implementation. Furthermore, the integration of the developed software within the context of ProcedoStudio.NET will also be discussed.

4.1 Implementation Details

This section will provide more detailed information on the actual implementation of the conceptual model developed in section 3. This information is primarily intended to serve as developer’s manual to describe the main functional parts the program consists of, their operation, implementation and specifics.

4.1.1 Base Classes and Methods

The core data structures and methods are defined in a separate library termed KnowledgeTemplates. The following classes and modules are available:

- **BinaryMatrix**. This class defines the base properties and behaviour of a square binary matrix, with facilities to add, remove, change and retrieve matrix elements. A Transpose method is also provided.

- **Connectivity**. This module allows retrieval of a DSM table from a properly-formatted comma-separated value (.csv) file. CSV files are in text format, and can be opened and saved to using any spreadsheet program.

- **ItemCollection** is a thin wrapper around the **List** class provided by the .NET framework, designed to be used when a collection of knowledge items is defined in the program. No special functionality is provided to this class at the moment.

- **KnowledgeObject** class defines the structure of a knowledge object, with properties like Name, Id, and Provides and Depends lists.

- **KnowledgeBase** is a class defining an aggregation of knowledge objects, with additional properties such as AdjacencyMatrix and ReachabilityMatrix. This is the data structure that contains all the information that defines the knowledge base.

- **Numeric** module provides algorithms for cycle enumeration and Warren’s and Warshall’s algorithms used to generate a reachability matrix.

- **SequenceCollection** class is a “list-of-lists” hierarchical structure used primarily to store the sequence obtained after DSM partitioning. Separate Levels (equivalent to DSM’s levels) are defined, each level consisting of one or more items.
4.1.2 Database Backend

The original program uses a Microsoft Access file database as persistent storage. However, the .NET framework provides transparent connection to the database component through the ADODB interface, regardless of the underlying database format. Virtually any ODBC and/or SQL-compliant database format can be used, provided that the corresponding database connectors are available.

Database connection functionality is performed by the Debris.DBConnector module, with methods for data retrieval using SQL.

**Database querying** The sample database accompanying the pilot ADAPT system includes more information than is needed for this project, distributed along more than 10 tables. The data that the program requires is how the knowledge objects are related to the parameters (both as dependencies and providing relations). Since the database schemas can vary from database to database, with database tables and columns having different names and data types, the task of navigating and retrieving the necessary information requires building and validating complex search queries. Providing support for flexible SQL parsing and generation from the user interface is beyond the scope of this thesis, as its complexity, consistency and security implications require significant resources and time. Instead, it was considered advantageous to select an already defined query from the database, which will dynamically retrieve data from it. The complexity is then shifted from the user interface to the database backend, where the facilities to control, test and customize a search query are already available. The database administrator/user is responsible to creating and validating the necessary SQL views that provide the data in the format expected by the program.

The program requires two queries for data input: a matching between the parameters and the knowledge objects that require them, and a matching between the variables and their corresponding provider, without any additional information. The following queries are used with the sample database:

**Code Sample 4.1** SQL Statement matching Knowledge Objects versus Parameters

```sql
SELECT ParameterIdFromRules.Name, Variable.FriendlyName
FROM ParameterIdFromRules, Variable
WHERE ([ParameterIdFromRules]![VariableFK]=[Variable]![Id]);
```

**Code Sample 4.2** SQL Statement matching Knowledge Objects versus Variables

```sql
SELECT KnowledgeObject.Name, Variable.FriendlyName
FROM KnowledgeObject INNER JOIN Variable ON KnowledgeObject.Id = Variable_DEFINEDBy
ORDER BY KnowledgeObject.Name;
```
The program assumes that the first column in each query contains the knowledge objects, and the second one contains the names of the parameters or variables, correspondingly. The program reads the available knowledge objects from both queries, creating `KnowledgeObject` objects, and filling their `Provides` and `Depends` lists.

### 4.1.3 Graphical User Interface

The graphical user interface is an important mediator between the user and the software product.

**Main Form** The program features a main form, which exposes the most important functionality of the program, without necessarily defining the expected workflow a user is expected to follow. The main form is intended to simply gather most of the functionality of the developed components in one place, showcasing their features and behaviour. It will not find place as a part of the ADAPT project.

- **Data Selection.** The user opens a database, and is required to select two queries from it (one that matches parameters to knowledge objects, and another to match variables to knowledge objects). The query results can be viewed by the user, as are the exact SQL statements that generate them (Figure 12).

  ![Figure 12: Data Selection view in Debris. For a larger preview, please consult Figure 19.1 in Appendix D.](image)

  The user needs to select the database to open (which should be in Microsoft Access .mdb format) from the **File → Open Database...** menu. The required SQL views should then be selected on the **Selection** tab page, in the drop down menus labelled **Parameters** and **Variables**, correspondingly.

- **Knowledge Base Generation.** After the selection is complete, the data can be parsed into the knowledge base data structure the program uses. This is can be done by clicking the **Generate Data** button on the main toolbar (see Figure 12). In order for this button to be active, both queries must be selected and should have passed the basic validation checks, indicated by the green LEDs besides each table.
Implementation Details

- **Data Validation.** Some basic data checks are showcased as well, including a preliminary test for circular dependencies, finding variables that are not provided by a knowledge object, multiple providers for a variable. The checks are activated by clicking the **Validate** button on the **Validation** tab page (see Figure 13). The data is then parsed into a tree view hierarchy, with top-level nodes being the knowledge objects, their dependency and provides lists as child nodes, and the design parameters as leaves.

![Figure 13: Data Validation view in Debris. For a larger preview, please consult Figure 19.2 in Appendix D](image)

- **Execution Control.** After the knowledge base has been filled with data, the execution control may be initiated, by clicking the **Configure Execution** button on the main toolbar. This prepares the list of available design parameters in the knowledge base and enables selecting the execution parameters (Figure 14). The two basic modes of execution are selected through the radio buttons labelled **Forward Execution** and **Backward Execution.** Two scopes of operation are also available (**Knowledge Objects** and **Variables**), depending on which dependencies need to be resolved.

![Figure 14: Execution Control view in Debris. For a larger preview, see Figure 24 in Appendix D](image)

In case of forward execution, nothing more need to be configured, and the dependency resolution can be initiated by clicking the **Execute** button. This will populate the right tree list with a proposed sequence of execution, partitioned by levels as in a DSM.
If backward execution is desired, the user needs to select the sought-after parameters in the list box labelled Selected Parameters, by using the arrow controls added right above the box. This list should hold only the parameters that need to be found — the program will return the absolute minimum of other parameters that need to be found in order to evaluate the ones needed. After clicking the Execute button, the list will appear (in dependency-resolved order) in the left list, excluding the goal parameters.

The program also features several other visual components, which will be described in the following sections.

**DSM Interface Component** The program features a stand-alone visual component that visualizes the DSM, called DsmView. The control contains all the necessary logic to partition the underlying DSM and present the rearranged table to the user. The user can switch between a view of the original and rearranged DSM from a context menu. A screenshot of the component in action is shown on Figure 15.

![Figure 15: DsmView visual component. On the left, loaded is an unoptimized DSM, with the context menu shown; on the right, the same DSM is optimized and partitioned.](image)

The blue squares indicate a dependency that is satisfied by the current arrangement, while the red ones show when an item depends on another that appears after it in the current view. The alternating green and white background for the items in the DSM mark the separate levels into which the DSM has been partitioned.

The DSM is populated by supplying the adjacency matrix and an ItemCollection containing the variable names. Functionality concerning adding and removing variables, and maintaining consistency doing so, is also included. The component raises two events, Optimized and Original to signal its parent control of its changed state. The control depends only on the KnowledgeTemplates library, as it uses several of the classes defined in it. DsmView is supplied as a separate
library project, and can be included and linked to at design time, simplifying the usage of the component in other software packages.

**MatchingGrid Component**  The MatchingGrid (Figure 16) control is a rectangular table, intended to show the dependencies between parameters and knowledge objects. The *provides* and *depends* relationships are colour-coded to distinguish one from another. In the example shown, the parameters are represented by the rows in the MatchingGrid, while the knowledge objects occupy the columns. Thus, an item with indices \((i, j)\) indicates that the corresponding relation exists between parameter \(i\) and knowledge object \(j\).

![Figure 16: MatchingGrid visual component. The depends relationship is denoted with red, and provides relation — with blue.](image)

The table serves an additional purpose when both the rows and columns index sequences correspond to the optimized sequences in the parameter and knowledge objects DSMs, respectively (this situation is depicted on Figure 16). In this case, the *provides* relation will occur always before the *depends* relation for each parameter. In turn, all the knowledge object dependencies will appear in the corresponding column before the parameters the object provides.

Additionally, the MatchingGrid component can also show the latest time at which a knowledge object can be executed, by examining the minimum column offset between all the parameters a knowledge object provides, and first knowledge object
in which they are required. For example, on Figure 16 the parameter Harness Resistance is evaluated in the first knowledge object, but its execution may be deferred until the knowledge object with ID 2, as this is the first knowledge object where the provided parameters are required.

Knowledge Base Editor  The knowledge base editor is a separate form using some of the controls already described in the previous subsections. It contains two DSMs — a parameter DSM and a Knowledge Object DSM, and a matching grid (Figure 17).

Figure 17: KnowledgeBase Editor. In the upper right corner — the knowledge object DSM. In the lower left corner — parameter DSM. In the lower right corner — MatchingGrid component. In the upper left corner — selection and manipulation controls.

The knowledge base editor is intended as a front-end tool to facilitate easy and straight-forward creation and modification of knowledge bases, together with transparent communication with the database to save and retrieve the modified data. It would be most useful in the beginning of the product design, when the parameters and especially their dependencies are still worked out. The editor could point out inconsistencies in the supplied data (e.g. parameters that are not provided, or knowledge objects without output). Additionally, cyclic dependencies between parameters and/or knowledge objects can also be visualized.

At this stage, the knowledge base editor is not complete — better synchronizing and consistency between the views is required, especially when the data is edited instead of simply viewing it. However, optimizing all views simultaneous
works as expected, and parsing of a supplied knowledge base is also performed accurately. The tool is included as a preview option in the modified ProcedoStudio.NET application.

4.2 Algorithm Design

This section will present an actual implementation of the algorithms discussed in sections 2.2.2, 4.2.5 and 4.4.2 namely algorithms for constructing the reachability matrix, for DSM partitioning and for topology sorting. Some algorithms for manipulating digraph cycles will also be considered here.

4.2.1 Reachability Matrix Generation

Two algorithms for constructing the reachability matrix of a digraph have been proposed in section 2.2.2 — Warshall’s algorithm and Warren’s algorithm. A sample code for both of them can be found below.

**Warshall’s Algorithm** Implementing Warshall’s algorithm seem to be a complex task, given the very formal description presented; however, from a programming point of view, it is very simple and straight-forward, as can be seen from the code sample below:

**Code Sample 4.3 Warshall’s algorithm in VB.NET**

```vbnet
W = AdjacencyMatrix
For k = 0 To n - 1
    For i = 0 To n - 1
        For j = 0 To n - 1
            W(i,j) = W(i, j) Or (W(i,k) And W(k,j))
        Next
    Next
Next
```

This algorithm has been tested with adjacency matrix of around 40 elements, and a measurable delay between the start and the finish of the program can be observed. As the algorithm complexity is of order $O(n^3)$, a two-fold increase of the number of elements in the matrix will result in eight times increase in computation time — an unfavourable situation. Thus, Warshall’s algorithm is not being used in the program, but the algorithm is kept available for reference purposes.

**Warren’s Algorithm** Warren’s algorithm is divided in two parts, each of which evaluating either the upper or the lower diagonal matrix of the reachability matrix.
4.2.2 Topological Sort

An algorithm that implements topological sort as presented in (Gross and Yellen, 1999) can be found in (Haggarty, 2002, p. 152). In pseudo-code, the algorithm can be expressed as follows:

**Code Sample 4.5 Topological Sort algorithm in pseudocode**

```
for each v in V
    calculate the antecedent set A of v
    while unlabelled vertices v remain for which A(v) is not empty
        label = label + 1
        u = a vertex with empty A(u)
        assign label to u
        for each unlabelled vertex v in V
            A(v) = A(v)-{u}
    end
end
```

The algorithm complexity is variable depending on the particular problem, but the worst-case complexity is at most $O(n^3)$, if we consider the calculation of the
antecedent set an unit operation. An algorithm that utilizes parallel computation, as opposed to the sequential algorithm defined above, is shown in \( E_r, 1983 \). The theoretical worst-case time complexity of such algorithm is \( O(n) \), but actual performance would be lower, as the separate processors involved the parallel computation need to be synchronised.

An actual implementation of this algorithm in VB.NET won’t be shown here due to its length, but can be seen in Appendix C.1

### 4.2.3 Backtracking

Following the backtracking description in section 2.4.3, a routine has been constructed that implements the algorithm in Visual Basic.NET (see Appendix C.2). The algorithm has as an input a subset of all design parameters that need to be evaluated. The `PrepareDependencies` function scans the knowledge base for the dependencies of all nodes currently needed by those already evaluated. While the list of these dependencies is not empty (that is, some items on the list still need some other parameters to be evaluated), those in the dependency list are added to a temporary one (stored in `CurrentList`). After that, they are added to the list of already evaluated items (`VariableList`), and are removed from the dependency list. The dependency list is then updated using the `PrepareDependencies` routine, and using `CurrentList` as the source for the required variables. The temporary list is then cleared, and the while loop is continued.

### 4.2.4 Digraph Cycles

If the knowledge base has a circular dependencies between two or more objects, then an exact execution sequence cannot be found, since in cyclic relationships none of the elements in a cycle can be evaluated before the others — they need to be determined simultaneously. A topological sorting implementation that does not anticipate this problem is bound to fall into an endless loop wherever such relationships are encountered. To this end, some method to deal with the presence of cycles in a graph is required, and that will be described in this subsection.

In a nutshell, in order to determine and eliminate circular relationships between knowledge items (and knowledge objects), the underlying graph must first be tested for the presence of cycles. The cycles can be then “compressed” into a single node, (which will not be in a circular relationship with any other), thus making the graph acyclic. Topology sorting can then be applied, and in the end, the compound variables can be replaced with the corresponding nodes that comprised it initially.

Cycle handling consists of four separate operations — cycle detection, cycle enumeration, cycle elimination, and cycle expansion — all of which will be discussed below.
Cycle Detection  Given the reachability matrix, it is trivial to test the graph for cycles of any length — since the reachability matrix shows whether a path of any length exists between any two vertices in the digraph. Thus, if two vertices $i$, $j$ are involved in a loop, the following condition is both necessary and sufficient:

$$M^*(i, j) = M^*(j, i) = \text{True}$$  \hspace{1cm} (1)$$

The above really means that if there is a path from node $i$ to node $j$ and a path between $j$ and $i$, then the nodes are involved in a cycle.

Cycle Enumeration  It should be noted that the condition shown in relationship [1] only shows whether the nodes are involved in a cycle; if a cycle consists of more than two nodes, or there are multiple cycles, the above statement is not sufficient to enumerate all nodes that are involved in a cycle. Therefore, a method to evaluate the separate cycles should be designed, that can group all the pairs of vertices that are involved in circular relationships, into their corresponding cycles.

One may easily observe that if there is a circular dependency between nodes $i$ and $j$, and between nodes $j$ and $k$, then there must be a circular dependency between $i$ and $k$ as well, meaning that all three nodes $i$, $j$ and $k$ are involved in a cycle. Hence, the cycle lists can be “grown” iteratively, starting from an arbitrary pair, adding its vertices to the current cycle list, then adding all nodes from pairs that contain at least one of the nodes already in the cycle list, until the pool of possible candidates is exhausted. Then, the nodes that form the current cycle list are removed from the list of possible candidates, and the procedure is repeated again, selecting new arbitrary pair and “growing” another cycle. The whole algorithm terminates when there are no more nodes for evaluation.

The concept can be illustrated in pseudo-code, as shown below:

**Algorithm 4.1  Cycle Enumeration**

Find all pairs that have circular dependency
While there are unevaluated pairs
  Add new cycle to the cycle list
  Add members of a (random) unevaluated pair to the cycle list
  For Each Pair in PairList
    If the cycle list contains a member of the current pair
      Add the other member to the current cycle, if it is not present
    End If
  End For
  Remove the considered pairs from the pair list
End While

One might also ask whether cycles-within-cycles can also be detected; to answer this question, one must remember that the reachability matrix is not a unique representation of a digraph — each digraph has a unique reachability matrix, but
the opposite is not true. However, regardless of the complexity and the topology of the cyclical subgraph, each node is reachable from each of the remaining nodes that comprise the cycle. To illustrate this by example, consider Figure 18 below:

![Simple cyclic graph](image1) ![Compound cyclic graph](image2)

**Figure 18: Cyclic graphs**

Both of these graphs contain six nodes, and evidently they have different topology. In the simple cycle graph on Figure 18.1 it is obvious that every node is reachable from every other node. The same holds for the graph on Figure 18.2: nodes 1 to 5 are in a simple cycle, just as in the other graph; node 6 can be visited only by going through node 5, but since 5 can be reached from every other node, then node 6 is also reached by any other node. The same reasoning can be applied to the nodes reachable from 6: as node 1 is reachable from 6, and from node 1 all remaining nodes can be visited, then every node in the graph is reached by node 6. Hence, for both graphs, each node can be reached from every other node. Consequently, both graphs have identical reachability matrices, although their adjacency matrices are not the same. This reasoning shows that from a reachability point of view, compound cycles are indistinguishable from simple cycles — but both can be treated in the same way by the algorithm.

**Cycle Elimination** After the nodes involved in cycles have been detected, they must also be removed from the reachability matrix. This can be done by “packing” all items in a cycle together into a compound variable, that will have no circular dependency with any other object. All variables that reference an object in the cycle need to be updated to reference the compound variable instead to preserve consistency, as the compound variable’s components are eliminated from the reachability matrix.

The procedure is repeated for all cycles that are encountered in the graph, introducing a new compound variable for each cycle. Additionally, it is necessary to keep track which variables are compound, and which of the original ones they replace.
**Cycle Expansion** The simplest way of expanding a cycle is to replace the previously created compound variables with their constituents after the graph has been topologically sorted (this algorithm has been discussed already in section 2.4.2). The pieces of data required are the original reachability/adjacency matrix, the node sequence obtained by the topological sort, and the mapping between the compound variables and the original variables produced after cycle enumeration. The operations can be presented shortly as follows:

**Algorithm 4.2 Cycle Expansion**

For each compound variable in the sequence
- Replace the compound variable with the simple variables it is comprised of
- Rearrange the adjacency/reachability matrix according to the expanded sequence

Matrix rearrangement can be completed in $O(n^2)$, while substitution of the compound variable with its components is done in linear time.

### 4.2.5 DSM Partitioning

The algorithm described in section 2.5 can be described concisely by the following pseudo-code as well, with a more complete code sample available in Appendix C.3:

**Code Sample 4.6 DSM Partitioning in pseudocode**

```plaintext
CalculateInitialAntecedentSets()
CalculateInitialReachabilitySets()

While UnlabelledItems > 0
    Sequence.AddNewPartitionLevel()
    For Each s In ReachabilityMatrix
        If NoDependencies(s) and AlreadyConsidered(s) Then
            AddToLevel(CurrentLevel, s)
        End If
    Next
    RemoveDependencies(ReachabilitySets, Sequence.Level(CurrentLevel))
    RemoveDependencies(AntecedentSets, Sequence.Level(CurrentLevel))
    UpdateConsideredList(Sequence.Level(CurrentLevel))
    Unlabelled = Unlabelled - Sequence.Level(CurrentLevel).Count
    CurrentLevel = CurrentLevel + 1
End While
```

The proposed implementation of the Warfield’s algorithm is capable of optimizing the DSM without having to deal specifically with circular dependencies — these are resolved by virtue of the algorithm’s design. The items in cyclic relations will invariably end up in one and the same level in the partitioned DSM. However, the algorithm does not guarantee that those objects will be grouped next to each
other in the final arrangement. Indeed, it has been found during the work on
the algorithm that if one parameter happen to be positioned in the same level
as several others that form a cycle, it may end up positioned in between them
implying that it is a part of the cycle, which is not true. This can lead to mislead-
ing conclusions on the actual dependencies of the items in the DSM, and it has
been decided that the DSM will be subjected to cycle elimination (as described in
section 4.2.4) prior to applying the partitioning algorithm, with subsequent cycle
expansion. The benefit from this more elaborate sequence of operations is that
during the elimination of cycles, the objects belonging to a cycle are guaranteed to
appear adjacent to each other in the final sequence, thus excluding the possibility
of misinterpreting the results.

The complexity class is rather difficult to estimate in this case, but the sepa-
rate functions have complexity classes as follows:

- Create initial antecedent/reachability sets \( O(n^2) \)
- NoDependencies() \( O(cn^2), 0 \leq c \leq 1 \)
- RemoveDependencies() \( O(cn^2), 0 \leq c \leq 1 \)
- UpdateConsideredList() \( O(n), 0 \leq c \leq 1 \)

### 4.3 Program Testing and Comparison

Building a software application on sound theoretical principles is not enough to
guarantee its correctness — it needs to be verified against known problems and
investigate whether the solutions are similar, exact, or different. As the devel-
oped program has two parts that produce measurable output, namely the DSM
component and the sequence-generating algorithms, they are the ones that will be
validated. If the software produces consistently correct results in several test cases,
then it is reasonable to believe that it is implemented correctly. Of course, that
still does not guarantee 100% accuracy, but it is rare that a software application
will claim such thing.

#### 4.3.1 DSM Component Testing

To be able to verify the DSM component, a small test application has been built
that utilizes the DSM control. The user needs to open a .csv (comma-separated
value) file that the program can parse to obtain the stored information. The .csv
file format is compatible with virtually any spreadsheet program available, includ-
ing Microsoft Excel and OpenOffice.org Calc. Additionally, the format is text-
based and can be directly edited by the user (although it is not user-friendly).

**Test Cases** Four test cases will be used, with increased complexity. The first
one is the simple course prerequisite example, shown in Figure 10. The second one
is a 20-item DSM, filled manually with random data. The third and the fourth
are DSMs used and developed during various projects in Tekniska Högskolan i
Jönköping, for the companies Kongsberg Automotive ASA and FläktWoods AB.
Reference Software  The software is compared against the DSM Excel macro developed at MIT (further referred here to as “MIT tool”), and freely available from http://grierson.nysedii.buffalo.edu/dsmweb/dsm_tools/DSM-MIT-ver-1_9.zip The macro uses Microsoft Excel VBA macros to partition a DSM, among its other functionality that is irrelevant to the testing procedure. The user is required to fill in the required data in a tabular format, and the program automatically partitions the DSM, pointing out the circular dependency blocks and the different partition levels in a visual format. The MIT tool produces data distributed along several sheets in the table file, but the ones that are relevant for this test are the DSM Input and AEAP tabs.

Test Program Availability  The testing program, together with the test cases can be downloaded from http://sourceforge.net/project/platformdownload.php?group_id=259682 (direct link), or through the project website at http://debris-kbe.sourceforge.net/ The package is accompanied with all the test cases shown here, and the corresponding reference tables produced by the MIT tool. Short instructions how to use the program are also included.

Test Results  The results from the tests are given in Appendix D — Figures 20 to 23.

One can observe that the Debris DSM implementation partitions the DSMs exactly in the same way as the MIT tool — the separate levels contain one and the same parameters. However, the results are not identical in ordering within a level, but since a topological sort in a set is not a unique sequence in the general case, such behaviour is to be expected. The program is able to properly recognize and combine together the variables that are in circular dependencies, and exactly the same loops as in the MIT tool can be easily found and identified. As far as computation performance is concerned, the proposed DSM component seem to partition markedly faster the DSM, although no quantitative tests have been performed in this regard. It should be noted as well that the MIT tool provides some extra functionality and produces more data than the Debris implementation, which is single-purpose.

The four tests supplied range from almost trivial (Figure 20), to real-world scenarios (Figures 23 and 22). In all cases, the program behaves consistently good, and returns stable, accurate and reproducible results in all test cases. Hence, it is can be concluded that the proposed implementation of a DSM visual component is accurate.

4.3.2 Forward and Backward Chaining Algorithms Testing

Only the forward chaining algorithm has been tested against an external tool, since there was no comparable functionality currently in ProcedoStudio.NET. The knowledge object sequences can be seen on Figures 24.7 and 24.8 on page 57. The resulting sequences are not identical, as already discussed, but the proposed exe-
execution order is consistent with the imposed dependencies between the knowledge objects.

One important difference between the old and the new dependency resolution systems exists — in the old version, the knowledge objects are executed as soon as they are discovered. In the newly-proposed variant, the sequence is obtained prior to the execution of the knowledge objects. Prior evaluation of the execution order is viewed as advantageous in terms of data consistency — if the program encounters e.g. a circular dependency between the knowledge objects, it will be able to signal the user and not waste time calling other routines simply to discover at a later stage that the execution cannot be completed. Additionally, the partitioning of the final sequence makes possible parallel execution of knowledge objects that are situated in one and the same level in the DSM, thus allowing smart planning of allocation of computational resources before the actual execution.

4.4 Integration with ProcedoStudio.NET

As stated in the project goals, the software developed in this thesis is intended to replace and extend the functionality of the ProcedoStudio.NET application, developed as a part of the ADAPT project. The previous version of ProcedoStudio used the KnowledgeWare workbench from CATIA to implement a crude dependency resolution scheme.

The software libraries developed in this thesis have been successfully integrated in the ProcedoStudio.NET application, removing the runtime dependency on CATIA for determining the resolution order. As the changes were implemented “under the hood”, from a user perspective the changes in the interface are minimal. However, from a programming point of view, the dependency resolution procedures are much more simpler and short. At this stage, only the forward chaining routine is available in the ProcedoStudio. However, it shouldn’t require much effort and coding to use the “as late as possible” evaluation scheme provided by the backtracking algorithm.

In addition, the knowledge base editor is added as a preview option to the main program window, allowing one to study the knowledge base contents and the relations between its constituents without opening the database and studying the data relations from there. Further integration could be achieved when the editor is supplied with the necessary functionality to accurately create and modify new knowledge bases, adding them to the existing ones in the database.

Furthermore, as the knowledge object order is dynamic and determined at runtime (opposed to the more restricted older implementation), the program is able to ask for the necessary knowledge objects that are to be used as input, determining their type and how to deal with them as they are supplied by the user. With partial evaluation (as supported by the backtracking algorithm), the program would only ask for the really required data, sparing the user the need to give all the details in advance.
5 Conclusion

The previous sections have discussed various aspects of the development of the project as related to the goals stated in section 1.2. This section will try to summarize the most important aspects of the thesis outcomes.

In section 2.2, a data representation model has been proposed, which converted the problem from a practical one to a more formal and abstract one. Moreover, the abstraction allowed the application of established mathematical principles and concepts in searching for the solution of the dependency resolution problem. Some theoretical algorithms as related to this have been introduced, which were subsequently analysed and implemented, as shown in section 4.2.

Database connectivity methods have been also developed, allowing transparent retrieval of the necessary information to construct a knowledge base without relying on a specific database schema. This has been achieved by proposing a simple and common formatting of the necessary data, achieved by predefined SQL statements inside the database, instead of developing a custom SQL builder controllable by the user interface. This imparts additional flexibility in the program, as the database schema need not conform to any hard-coded rules — it is the responsibility of the database administrator/user to develop the necessary SQL statements.

Several graphical user interface components have been constructed, namely the DsmView and the KBaseEditor controls (section 4.1.3). These can successfully be used for simple and informative visualization of the relationships between the entities in the knowledge base without relying on direct database access and querying.

Methods for inspecting the consistency of the data in the knowledge base have been presented in section 4.1.3. They can provide feedback to the user during development or modification of the knowledge base, thus helping the user to spot and rectify omissions and errors in the knowledge base structure and design.

The algorithm implementations (and thus the algorithms themselves) have been tested against reference tools of the same principles, in section 4.3.2. The obtained results proves that both the proposed algorithms and their implementation are reasonably accurate\(^\text{13}\) with the outcomes of the proposed solution and the reference being comparable or identical.

Finally, parts of the developed software in this thesis have been successfully used to replace the old inference system in ProcedoStudio.NET with the newly proposed one (see section 4.4). The new functionality leaves room for extension, allowing exploration of further strategies for execution control.

\(^\text{13}\)As noted in [McConnell 2004, chapter 22.1], a non-trivial program can only be proven wrong; proving it 100\% correct is virtually impossible.
6 Future Work

With the accomplishment of the project goals (discussed in the previous section), the work on the project has been completed. However, some of the concepts and the software developed can be extended further beyond the scope of the master thesis.

Visual Components One of the most useful assets of the project are the developed visual components (DsmView and KBaseEditor). Their design allows easy reuse in other software projects, as proved by integrating them effortlessly in the existing ProcedoStudio.NET. The KBaseEditor component can be extended to allow not only displaying knowledge base relations (as it is currently), but also to directly create and modify new knowledge bases and record them into the database. Additionally, the data from the knowledge base editor could also be exported to a convenient format for further manipulation in, for example, a spreadsheet program. Some functionality for this exists at the moment, but it has not undergone sufficient testing to declare it complete and stable.

Dependency Resolution Algorithms The methods for execution control developed feature data-driven and goal-driven approach to dependency resolution. While the main goal of dependency resolution in terms of knowledge objects has been accomplished, the user is currently unable to select a specific execution strategy (for example “as early as possible”, or “as late as possible”), that could serve better his/her needs. In addition, one may be interested in partial evaluation of the knowledge objects (for example, due to long execution time of a knowledge object, or simply unavailable data). The backtracking algorithm is capable of catering to this needs, although it would give a sequence of parameters rather than knowledge objects. Conversely, the currently used forward-chaining procedure operates only on knowledge objects, and in “as soon as possible” fashion. Therefore, additional functionality can be added to accommodate different user needs in terms of execution control.

Future Development As stated in section 3.8, the project source code will be made available freely in both binary and source code form. The project is currently hosted at http://debris-kbe.sourceforge.net/ with an intention to continue the development of the project’s main components to provide a starting point for researchers and engineers for reusing, testing and adapting new ideas based on the concepts of knowledge based engineering.
References


Sunnersjö, Staffan (2007), Strategies for adaptable design automation systems in the manufacturing industry, Research project application, Swedish Knowledge Foundation.


A  Elements of Graph Theory

The definitions given here can be found in [Gross and Yellen 1999], unless otherwise specified.

Definition. A graph $G = (V, E)$ is a mathematical structure consisting of two sets $V$ and $E$. The elements of $V$ are called vertices (or nodes), and the elements of $E$ are called edges. Each edge has a set of one or two vertices associated to it, which are called endpoints.

Definition. A directed graph (or digraph) is a graph whose edges are directed — the edge has distinguished endpoints marked as head and tail, with an assumed direction from its tail to its head.

Definition. A graph is simple if it has neither self-loops (an edge which joins an endpoint to itself) or multi-edges (a set of edges having identical endpoints).

Definition. A bipartite graph $G$ is a graph whose vertex-set $V$ can be partitioned into two subsets $U$ and $W$, such that each edge of $G$ has one endpoint in $U$ and one endpoint in $W$. The pair $U, W$ is called a (vertex) bipartition of $G$, and $U$ and $W$ are called the bipartition subsets.

Definition. A complete bipartite graph is a simple bipartite graph such that every vertex in one of the bipartition subsets is joined to every vertex in the other bipartition subset.

Definition. A directed walk from $v_0$ to $v_n$ is an alternating sequence

$$W = \langle v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n \rangle$$

of vertices and arcs, such that $\text{tail}(e_i) = v_{i-1}$ and $\text{head}(e_i) = v_i$, for $i = 1, \ldots, n$.

Definition. A trail is a walk with no repeated edges.

Definition. A path is a trail with no repeated vertices.

Definition. A digraph is acyclic if for each two vertices $v_i, v_j$ belonging to $V$, if there exists a directed path from $v_i$ to $v_j$, then there is no directed path from $v_j$ to $v_k$ and vice versa.

Definition. The adjacency matrix of a digraph $D$, denoted by $A_D$, is the matrix whose columns and rows are indexed by identical orderings $V_D$, such that

$$A_D[u, v] = \begin{cases} 
\text{the number of arcs from } u \text{ to } v & \text{if } u \neq v \\
\text{the number of self-loops at } v & \text{if } u = v
\end{cases}$$

Definition. A partial ordering of a set $S$ (poset) is a relation between the objects of $S$, which may be denoted with the symbol “≤”, satisfying the following properties for any objects $x, y$ and $z$ (not necessarily distinct) in $S$:

- If $x \preceq y$ and $y \preceq z$ then $x \preceq z$. (Transitivity.)
- If $x \preceq y$ and If $y \preceq x$ then If $x = y$. (Antisymmetry.)
• $x \preceq x$. (Reflectivity.)

The expression $x \preceq y$ is read “$x$ precedes or equals $y$.” (Knuth, 1973)

**Definition.** Two elements $x$ and $y$ of a poset $(S, \prec)$ are *comparable* if either $x \prec y$ or $y \prec x$.

**Definition.** A **total order** on a set $S$ is a partial order on $S$ such that every two elements are comparable.

**Definition.** A total order $\prec^t$ on a set $S$ is *compatible* with a partial order $\prec$ on $S$ if $x \prec y \Rightarrow x \prec^t y$.

**Definition.** A **topological sort** of a poset $(S, \prec)$ is a construction of a total order $\prec^t$ on $S$ that is compatible with $\prec$.

**Definition.** A **minimal element** in a poset $(S, \prec)$ is an element $m \in S$ such that for all other $x \in S$, $x \nprec m$.

**Definition.** Let $G = (V, E)$ be a digraph with $n$ vertices, and $M$ be its adjacency matrix. A non-zero entry in $M$ indicates the existence of a directed arc in $G$. The logical Boolean product $M^k$ records the existence of paths of length $k$. Finally, the **reachability matrix**

$$M^* = M \text{ or } M^2 \text{ or } M^3 \text{ or } \ldots \text{ or } M^n$$

records the existence of paths of any length between vertices (Haggarty, 2002).
B Computational Complexity

It is often beneficial in mathematics and computer science to measure how long time an algorithm will take to solve a problem of given size. The “time” as an absolute measure in this case serves little purpose, as it is dependent on everything from the computer hardware, computer architecture, operating system, programming language, compiler, concurrently running processes, and so on. In order to eliminate these variables, it is necessary to introduce a measure of performance that does not depend on those factors, but is rather a characteristic of the algorithm itself.

The notion of time-complexity function $f(n)$ is then introduced, which shows how the execution time of the algorithm depends on the size of the problem. An important consideration is to show how the algorithm performs for large problems (at which the algorithm design becomes much more important).

First, the notion of function dominance can be expressed formally as follows:

**Definition.** Let $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$. The function $g$ dominates $f$ (or $f$ is dominated by $g$) if there exist constants $m \in \mathbb{R}^+$ and $k \in \mathbb{Z}^+$ such that $|f(n)| \leq m|g(n)|$ for all $n \in \mathbb{Z}^+$, where $n \geq k$.

Informally, this means that for sufficiently large values of $n$, $f(x)$ is upper bounded by $g(x)$; that is, the function $f(x)$ grows at the same rate (or lower) as $g(x)$. This can be expressed also with the so-called big-oh notation as $f \in \mathcal{O}(g)$ (Grimaldi, 2003).

In algorithm analysis, this concept is used to show how the number of steps an algorithm requires depend on the size of the problem. For example, consider an algorithm $A(n)$, requiring $2n^3 + 5n + 7$ operations to complete. As the size of the problem $n$ tends to infinity (for all practical purposes, meaning that $n$ grows to very large numbers), the total number of steps the algorithm requires is determined primarily by the term $n^3$, expressed as $A(n) \in \mathcal{O}(n^3)$.

In practice, algorithms with as low growth rate as possible are desired — an algorithm having, for example, a growth rate of $\mathcal{O}(n^4)$ means that doubling the size of the problem will increases the number of operations (hence time) required by a factor of $2^4 = 16$. Such fast growth severely limits the applicability of this algorithm for non-trivial problems. Algorithms running in logarithmic or linear time are most often sought after, although $\mathcal{O}(n^2)$ algorithms can still be of utility.
C  Code Samples

C.1  Topological Sorting in VB.NET

Public Function TopologicalSort(ByVal KBase As KnowledgeBase) As SequenceCollection(Of String)
'
    Implements a forward-chaining topology sort, as given in (Haggarty 2002, p. 151-152)
'
    Dim KB As KnowledgeBase = KBase.Copy
    Dim Unlabelled As Integer = KB.Count
    Dim CurrentLabel As Integer = 0
    Dim RemoveSequence As New List(Of KnowledgeObject)
    Dim KBSequence As New SequenceCollection(Of String)

    While Unlabelled > 0
        KBSequence.AddNewLevel()
        '
        ' identify the knowledge objects without dependencies...
        For Each CurrentObject As KnowledgeObject In KB
            If CurrentObject.Deps.Count = 0 Then
                ' add them to the sequence at the current level
                CurrentObject.Label = CurrentLabel
                KBSequence.AddToLevel(KBSequence.Count - 1, CurrentObject.Name)
            End If
        Next

        For Each CurrentObject As KnowledgeObject In RemoveSequence
            RemoveDependencies(KB, CurrentObject.Provides)
        Next
        RemoveSequence.Clear()

        For Each CurrentObject As String In KBSequence.Level(KBSequence.Count - 1)
            If KB.Contains(CurrentObject) Then
                KB.Remove(CurrentObject)
            End If
        Next

        CurrentLabel = CurrentLabel + 1
        Unlabelled = Unlabelled - KBSequence.CurrentLevelCount(KBSequence.Count - 1)
    End While

    Return KBSequence
End Function

Private Sub RemoveDependencies(ByVal KBase As KnowledgeBase, ByVal DependencySequence As List(Of String))
    Dim RemoveList As New List(Of String)
    For Each CurrentObject As KnowledgeObject In KBase
        For Each Dep As String In CurrentObject.Deps
            If DependencySequence.Contains(Dep) Then
                RemoveList.Add(Dep)
            End If
        Next
    Next
End Sub
Backtracking in VB.NET

```vbnet
    For Each CurrentItem As String In RemoveList
        CurrentObject . Depends . Remove (CurrentItem)
    Next

    RemoveList . Clear (
    Next

End Sub

C.2 Backtracking in VB.NET

Public Function BackwardInference(ByVal KBase As KnowledgeBase, ByVal RequiredList As List(Of String)) As List(Of String)
    ' backward inference on a variable level
    Dim DependencyList As List(Of String) = PrepareDependencies(KBase, RequiredList)
    Dim VariableList As New List(Of String)
    Dim CurrentList As New List(Of String)

    While DependencyList.Count > 0
        For Each Dependency As String In DependencyList
            CurrentList . Add (Dependency)
        Next

        For Each CurrentItem As String In CurrentList
            If Not VariableList.Contains(CurrentItem) Then
                VariableList . Add (CurrentItem)
            End If
            DependencyList . Remove (CurrentItem)
        Next

        DependencyList = PrepareDependencies(KBase, CurrentList)
        CurrentList . Clear ()
    End While
    Return VariableList
End Function

Private Function PrepareDependencies(ByVal KBase As KnowledgeBase, ByVal RequiredList As List(Of String)) As List(Of String)
    Dim RequirementsList As New List(Of String)

    For Each CurrentObject As KnowledgeObject In KBase
        For Each CurrentItem As String In CurrentObject . Provides
            If RequiredList . Contains (CurrentItem) Then
                For Each Dependency As String In CurrentObject . Depends
                    If Not RequirementsList . Contains (Dependency) Then
                        RequirementsList . Add (Dependency)
                    End If
                Next
            End If
        Next
    Next
    Return RequirementsList
End Function

C.3 DSM Partitioning in VB.NET

Private Sub PartitionDsm(ByVal ReachabilityMatrix As BinaryMatrix)
```
The algorithm used can be found in (Warfield 1973). The reachability matrix is required to perform partitioning of the DSM.

```vbnet
Dim MatrixSize As Integer = ReachabilityMatrix.Count
Dim ReachabilitySet As New SequenceCollection(Of Integer)(MatrixSize)
Dim AntecedentSet As New SequenceCollection(Of Integer)(MatrixSize)
Dim ConsideredList As New List(Of Boolean)(MatrixSize)

' determine the reachability and antecedent sets for each variable in the DSM
For i As Integer = 0 To MatrixSize - 1
    ReachabilitySet.AddLevel(New List(Of Integer))
    AntecedentSet.AddLevel(New List(Of Integer))
    ConsideredList.Add(False)
Next

For i As Integer = 0 To MatrixSize - 1
    For j As Integer = 0 To MatrixSize - 1
        If ReachabilityMatrix(i, j) <> 0 Then
            ReachabilitySet(i).Add(j)
            AntecedentSet(j).Add(i)
        End If
    Next
Next

Dim Unlabelled As Integer = MatrixSize
Dim CurrentLevel As Integer
Dim VarSequence As New SequenceCollection(Of Integer)

While Unlabelled > 0
    VarSequence.AddNewLevel()
    ' for each variable in the reachability matrix...
    For i As Integer = 0 To MatrixSize - 1
        ' if no dependencies, and the case hasn't been considered, add them to the current level
        If ConsideredList(i) = False AndAlso NoDependencies(ReachabilitySet(i), AntecedentSet(i)) = True Then
            VarSequence.AddToLevel(CurrentLevel, i)
        End If
    Next
    ' remove the variables in the current level from both antecedent and reachability sets
    RemoveDependencies(ReachabilitySet, VarSequence.Level(CurrentLevel))
    RemoveDependencies(AntecedentSet, VarSequence.Level(CurrentLevel))
    ' go to next level, and adjust the number of unlabelled variables
    For Each CurrentObject As Integer In VarSequence.Level(CurrentLevel)
        ConsideredList(CurrentObject) = True
    Next
    Unlabelled = Unlabelled - VarSequence.CurrentLevelCount(CurrentLevel)
    CurrentLevel = CurrentLevel + 1
End While

End Sub
```
D Screenshots

19.1: Data Selection

19.2: Data Validation

Figure 19: Main Form Screenshots
20.1: Course Prerequisites — Original

20.2: Course Prerequisites — DSM MIT

20.3: Course Prerequisites — Debris

Figure 20: Course Prerequisites DSM — Comparison between DSM MIT and Debris
21.1: Random Items — Original

21.2: Random Items — DSM MIT

21.3: Random Items — Debris

Figure 21: Random Items DSM — Comparison between DSM MIT and Debris
### Figure 22: Kongsberg Automotive DSM — Comparison between DSM MIT and Debris

#### 22.1: Kongsberg Automotive — Original

#### 22.2: Kongsberg Automotive — DSM MIT
22.3: Kongsberg Automotive — Debris

Figure 22: Kongsberg Automotive DSM — Comparison between DSM MIT and Debris

23.4: Fläktwoods — Original

Figure 23: Fläktwoods DSM — Comparison between DSM MIT and Debris
### 23.5: Fläktwoods — DSM MIT
![Diagram showing Fläktwoods DSM — Comparison between DSM MIT and Debris]

### 23.6: Fläktwoods — Debris

![Diagram showing Fläktwoods DSM — Comparison between DSM MIT and Debris]

Figure 23: Fläktwoods DSM — Comparison between DSM MIT and Debris
Screenshots

**24.7: Knowledge Object sequence obtained with ProcedoStudio.NET**

**24.8: Knowledge Object sequence determined by Debris**

*Figure 24: Execution Control in ProcedoStudio.NET and Debris*