Detecting differences between versions of Microsoft Dynamics NAV

Morten Rhiger

IT University Technical Report Series **TR-2010-129**

ISSN 1600–6100 **September 2010**

ISSN 1600–6100

ISBN 9788779492219

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web **`www.itu.dk`**

Detecting differences between versions of Microsoft Dynamics NAV

Morten Rhiger

The IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S, Denmark

E-mail: mir@itu.dk
Homepage: www.itu.dk/people/mir

August 12, 2010

Abstract

Microsoft Dynamics NAV is an Enterprise Resource Planning system developed by Microsoft. NAV is customized to specific countries, industrial segments, and enterprises by software developers both from the Microsoft Cooperation and from a chain of independent partners.

In NAV, customizations are implemented as source-code modifications. This customization mechanism is flexible, since customizations are not limited to specific models imposed by implementation language or external customization tools. Therefore, it directly contributes to maintaining and enlarging the set of countries, industrial segments, and enterprises reachable by NAV. However, the customization mechanism of NAV imposes a significant cost of migrating customizations between different versions of the core product.

This reports describes tools developed with the purpose of detecting and characterizing customizations applied to Microsoft Dynamics NAV. We propose to detect customizations by identifying *differences* between original and customized versions of NAV. In order to enable analyzes of the detected customizations, we furthermore suggest that differences are represented as hierarchically structured*tree alignments*.

The primary components of the tools are an implementation of by Jiang, Wang, and Zhang's algorithm for computing tree alignments and its use in a tool that detects changes between two different version of Microsoft Dynamics NAV. The report serves both as a manual to using the tools and to modifying the tools. The tools described in this report were implemented by the author between August 2008 and July 2009 when he was associated with a research project on Evolvable Software Products[1] involving the IT University of Copenhagen and Microsoft Dynamics, Vedbæk, Denmark.

[1] `http://www.itu.dk/research/sdg/doku.php`

Contents

List of Figures

Chapter 1

Introduction and motivation

A software system is subject to recurring modifications that extend the system with new features, adapt it to new environments, eliminate its weaknesses and errors, or otherwise improve the system [11]. When several different groups of developers maintain a software system, they face the challenge of managing these modifications in such a way that the modifications can be combined in the final system while reducing the coupling between them.

Modern programming languages provides features to help maintaining and combining different developers modifications. However, in situations where linguistic features are either not present or not sufficient, the most flexible solution is probably to represent modifications as changes to the source code and then merge, either manually or using automated merge tools, the different developers modifications into the final system.

This report is about detecting changes in the source code of the Microsoft Dynamics NAV software system. The primary goal of this work has been to investigate new features that can help managing source-code modifications in NAV.

1.1 Microsoft Dynamics NAV

Microsoft Dynamics NAV is an Enterprise Resource Planning (ERP) system targeted at small and medium-sized businesses. NAV provides support for a wide range of business aspects, including financial management, sales and purchase, supply-chain management, marketing, and customer relationship management. NAV is highly customizable and has been used by companies in various different industrial segments, ranging from manufacturing and distribution, over construction, to governmental organizations.

Microsoft does not directly sell NAV to end customers. Instead, independent Value-Added Resellers (VAR) customize a general version of NAV to meet the specific requirements of businesses within a particular country or industrial segment. There are two distinguishing features of the NAV customization model, namely that (1) many as-

pect of NAV can be customized, including the client layer that controls the end-user experience, the business logic layer that implements common business routines, and the database layer, and that (2) many customizations are implemented as source code modifications [20]. (There are also customizations that are implemented as changes to metadata, such as additions of new fields to the database or modifications of exisiting fields. Since these customizations are conceptually simpler, they are often preferred over code modifications. In this report, we assume that the combination of such metadata customizations can be dealt with automatically. Therefore, we do not investigate the nature of these customizations in this report, but concentrate on source code modifications.)

Much of the success of NAV can be attributed to the flexibility that its customization mechanism provides. But in this mechanism also lies one of the greatest challenges of NAV: a significant burden (and cost) is imposed upon developers when migrating customizations applied to one general version of NAV to the next version of NAV. This situation is commonly referred to as the *upgrade problem* for NAV [24]. A simple instance is described in Figure 1.1 on the facing page. The current lack of built-in support for migrating customizations in NAV forces partners to either migrate manually or to merge textual exports of the business logic using untrusted third-party merge tools. Since both Microsoft, independent partners, and end customers modify and augment the behavior of NAV, all NAV developers in the development chain, from Microsoft via its partners to customers, are affected by the upgrade problem.

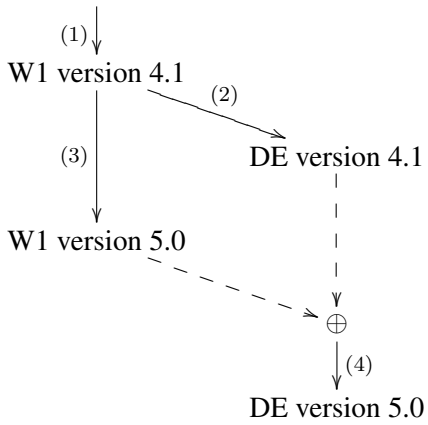
In this report, we let the term *(source code) modification* refer to one local change in source code from one version to another, as perceived by a particular model. A change can be either an addition, a deletion, or an update. The location of the change will be referred to as its *point of modification.* We let *customization* refer to a set of modifications that constitute a logical unit.

1.1.1 The NAV ecosystem

Several different partners and developers contribute to a final implementation of NAV. In the following list, these partners appear in same order as in a typical development chain.

Global Development and Localization (GDL): These are part of the Microsoft organization modifying the core NAV product (W1) to meet culture- and language-specific requirements of specific countries. Historically, most GDL partners started out as independent partners, but have since then been acquired by Microsoft.

Independent Software Vendors (ISVs): These are independent partners that typically extends the core NAV product with features (so-called *vertical solutions*) for a specific industrial segment (such as transportation, rental, or food and beverage industries) or for a specific task (such as data analysis or customer-relationship management).



Microsoft is responsible for products in the left part of the diagram. An independent Global Development and Localization (GDL) branch of Microsoft situated in Germany is responsible for products on the right. (In the diagram, time proceeds downwards.)

Microsoft develops version 4.1 of the core NAV product (called W1) and releases the source code of the business logic (1). The GDL partner customizes this product to the German market (2). These customizations involve culture- and language-specific modifications and adaptations to German law and conventions. All adaptations are implemented as modifications to the business logic source code.

After some time (typically two years), Microsoft releases the next major version of the NAV product (3). The GDL partner is now faced with the challenge of combining their customizations to the old version 4.1 with the new version 5.0 (4). Currently, the total cost of this migration (e.g., (4)) can be as much as 30% of the cost of the original adaptation (e.g., (2)).

(In order to simplify the description, this example illustrates the upgrade problem from the perspective of a GDL "partner." In reality, however, GDL (e.g., (2) and (4) in the diagram) is increasingly being performed by Microsoft. ISV partners address less artificial instances of the upgrade problem.)

Figure 1.1: An instance of the upgrade problem for Microsoft Dynamics NAV.

Value-Added Resellers (VARs): These are independent partners that typically combine a localized version of NAV with standalone modules into a product targeted either at a specific industrial segment or at one enterprise. VAR partners resell NAV products and provides consultancy services to customers.

Local Developers: The final system may be subject to further fine-tunings while running. The company who has bought the NAV product sometimes has the rights

and abilities to implement these fine-tunings itself.

There are more than 3 500 certified independent ISV and VAR partners worldwide, with an estimated total number of employees of more than 100 000 [19]. Together, these partners served more than 1 000 000 customers in 2006 [15]. A typical VAR partner employs 20–30 developers and consultants [19]. It is generally believed that few VAR partners are technically skilled IT professionals [3].

1.1.2 The NAV architecture

For a comprehensive architectural overview of NAV, consult Hvitved's forthcoming Ph.D. thesis [6]. For the rest of this technical report, however, the following description of NAV is sufficient.

NAV consists of a general framework that provides access to an extensible database of (business) "objects." This framework cannot be modified by partners. (Managing the different versions of the framework is thus not considered a problem.) The classes of business objects supported by NAV are listed in Figure 1.2. Reports, Forms, and Menusuites belong in the client layer, Codeunits belong in the business logic layer, and Dataports, XMLports, and Tables belong in the database layer. The particular purposes of the different classes are not relevant for this report.

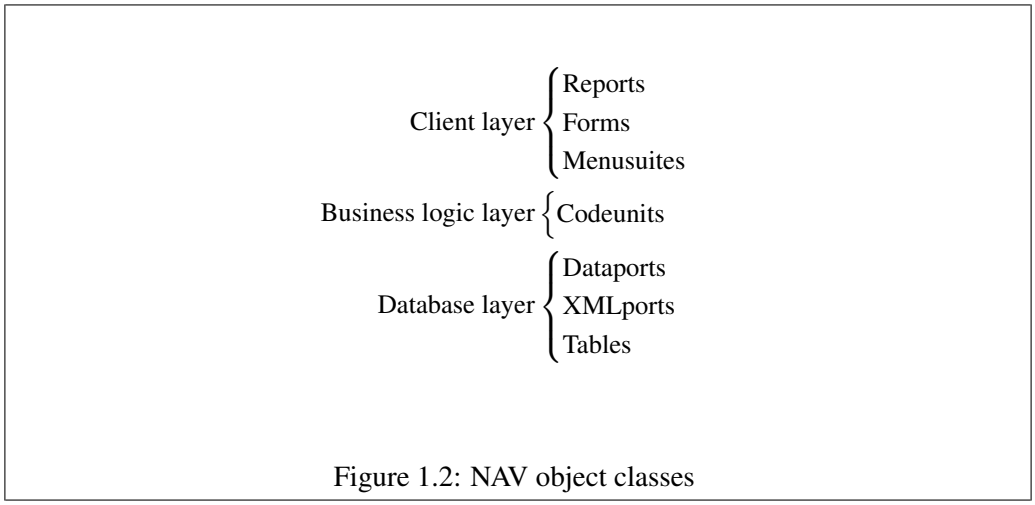


Figure 1.2: NAV object classes

Each business object has a name and a number. Numbers are unique within each class of object (but business objects of different classes may have the same number). Objects definitions are divided into *sections* of metadata and code that appear in the textual file format for the NAV application object types. The kind of sections allowed in an object

depends on its class. The sections supported by NAV are listed in Figure 1.3. Again, The particular purposes of the different sections are not relevant for this report.

CODE	CONTROLS	DATAITEMS	ELEMENTS
EVENTS	FIELDGROUPS	FIELDS	KEYS
MENUNODES	OBJECT-PROPERTIES	PROPERTIES	RDLDATA
REQUESTFORM	REQUESTPAGE		

Figure 1.3: NAV object sections

All business objects may have embedded code attached at various places. For example, a table contains code in triggers that are activated when entries in the table are inserted, modified, or deleted. Frequently used code may be structured into procedures. All objects may contain such procedures, but large (and commonly used) procedures are usually stored in Codeunit objects. Embedded code pieces are implemented in a procedural language called C/AL.

The location of an embedded code piece (or a scalar metadata value) in the business objects can be uniquely identified by a *path*, similar to a fully qualified identifier in a modular programming language. The individual elements in such a path may include the class of an object, the number of an object, the name of the section inside an object, the number of a field in a table, the name of a trigger, the name of a procedure, etc. Code paths denote leaves in a tree containing the modular structure of NAV. Some example paths are shown in Figure 1.4. The last three paths in this figure identify named procedures as part of a business object definition. The remaining paths identify triggers. Note that some legal paths (such as the intermediate paths `Table/14` and `Table/14/PROPERTIES` from Figure 1.4) do not identify code pieces.

```
Table/14/PROPERTIES/OnDelete
Table/14/FIELDS/5703/OnValidate
Form/99000959/CONTROLS/2/OnPush
Table/37/CODE/CheckWarehouse
Codeunit/80/CODE/DivideAmount
Codeunit/80/CODE/Increment
```

Figure 1.4: Example code paths

Version 5.0 of the uncustomized NAV (called W1), consists of 3 579 business objects implementing the business logic of the application containing a total of 11 509 procedures

and 18 551 code pieces in triggers. Version 5.0 of the GDL customizations add up to 600 objects, 1 800 procedures, and 4 500 triggers to W1.

Many customizations can be performed by modifying scalar properties in the database. One example could be modifying a numerical value for the "Discount Percentage" for a specific "Payment Term." Another example could be adding, deleting, or modifying fields in the table of Payment Terms. These changes are easy to identify. (An experimental version of a tool managing customizations of scalar properties has been developed by Microsoft Dynamics, for the AX ERP system.) However, most customizations require modifications to the business logic implemented by embedded C/AL code pieces.

All customization may be applied to a running version of NAV, via its integrated development environment C/SIDE. Once applied, it is generally impossible to roll back a modification. It is possible to export and import some or all of the business objects to and from text-based or XML-based formats. This ability is utilized by Microsoft and by some partners to maintain external repositories of modifications. In such settings, modifications are less often performed on a running version of NAV.

1.2 Detecting changes

The NAV upgrade problem is an example of *software merging:* Two independent branches or derivatives (in Figure 1.1, W1 version 5.0 and DE version 4.1) of a common ancestor (in Figure 1.1, W1 version 4.1) must be combined. The most successful approach to merging is probably those that merges the *differences* between one branch and the common ancestor into the second branch [13]. This is know as a *three-way merge*. Generally, the result of merging two branches is not uniquely defined. Consequently, three-way merging of software can not be fully automated. In particular, when the two branches modify the same feature in the common ancestor in different ways, a software merging system must chose one modification over the other. The decision of which branch should take priority must often be resolved by a programmer with insight in both branches, although (partially) automated methods for prioritizing exists [13].

Tools that successfully detect differences between two documents dates back at least to the first implementation of the UNIX diff tool [5]. To make them scale, diff and its cousins use dynamic programming to solve variations of the *longest common subsequence* problem between the sequence of lines in the two documents [14, 17, 23]. UNIX diff works well as a subroutine in three-way merge tools [8]. As an example of its use, consider the programs in Figure 1.5. UNIX diff computes the following differences between the two "old" files in Figure 1.5(a) and Figure 1.5(b).

```
2a3
> } else {
```

This instruction states that the customized file (b) can be obtained by adding ("`a`") line three ("`3`") from the customized file after line two in the old file ("`2`"). If we ask UNIX

```
1   if (x == 0) {
2     print(y);
3     print(x);
4   }
```

(a) Old program

```
1   if (x == 0) {
2     print(y);
3   } else {
4     print(x);
5   }
```

(b) Customized old program

```
1   if (y < x) {
2     print(y);
3     print(x);
4   }
```

(c) New program

```
1   if (y < x) {
2     print(y);
3   } else {
4     print(x);
5   }
```

(d) Customized new program, derived automatically from (a), (b), and (c)

Figure 1.5: Simple software-merging scenario

diff to include the (1-line) *context* surrounding the modification, then the result is the following instruction in *unified diff format*.

```
--- a.txt       2009-06-30 11:43:44.531250000 +0200
+++ b.txt       2009-06-30 11:43:49.671875000 +0200
@@ -2,2 +2,3 @@
   print(y);
+} else {
   print(x);
```

These differences can be merged into the new file in Figure 1.5(c) automatically and without conflicts, using the GNU patch tool. The result is shown in Figure 1.5(d). (The same result can be achieved by passing files in part (b), (a), and (c) in Figure 1.5 to the GNU diff3 tool.) The scenario just described is common in software merging [13] where it is typically driven by a revision control system [22].

Diff-like tools that perform line-by-line comparison based on the longest common subsequence problem are less satisfying when the inputs are structured documents and, in particular, when the differences computed are subject to additional mechanical analysis. Any general attempt to analyze program structure, types, the flow of control or data, software metrics, etc. seems to require correctly structured programs fragments as input.

Neither of the two outputs from diff shown above lend themselves towards further mechanical analysis. It is possible to recover the structure and context of the modification by considering the original files. In this report, we suggest a different approach where the differences of two programs is represented as a *tree alignment* [7]. A tree alignment is itself a tree that contains both version being compared, as well as the locations where they differ. We argue that, for the purpose of analysis, tree alignments are superior to traditional sequence-based diffs.

1.3 Managing changes in NAV

The aim of the work documented in this report has been to address the following challenges.

Characterizing code modifications: If most code modifications match a small set of common patterns, then it might be possible to manage most code modifications using only a few techniques. Part of the work presented in this report has been aimed at providing insight into actual code modifications applied to NAV internally at Microsoft and externally by independent partners in order to detect such patterns.

Identifying patterns of code modifications seems difficult, if not impossible, without an in-depth understanding not only of the core version of NAV but also of a large set of derived versions. Since no single person has this overview, the "shapes" of potential code-modification patterns are currently unknown.

We have attempted to address this issue by studying a particular approach to managing code modifications and by identifying existing code pieces that fit within this approach.

Managing code modifications: Any successful customization mechanism for NAV must be compatible with all existing customized versions of NAV. Migrating existing code to a new customization mechanism should preserve as much of the original program as possible.

We have studied a customization mechanism that allows new code to be "attached" to existing procedures in a way that syntactically separates the new code from the existing procedure. The mechanism is similar to a simple variant of Aspect-Oriented Programming (AOP) [9] in which *join points* (the source locations in the existing program at which to add additional code) denote at most one source location and are limited to ends of procedures.

An existing code modification fits within this customization mechanisms when it amounts to adding a sequence of statements at the end of a single procedure. Due to the requirements of join points, variables local to the existing procedure can

be made available to the additional code by passing them "by reference" to the additional code.

To provide additional insight into actual code modifications, we have widened the search for common code-modification patterns to include any location where a piece of code is added, removed, or updated. Some of the code modifications detected can be handled using the AOP-like approach, whereas others cannot.

Two or more code modifications may logically belong to the same customization, even if they are applied at different procedures or modules. We have not addressed the challenge of grouping together such code modifications.

The following additional issue emerged during the present study but has not been addressed in this report.

Querying code modifications: The tools described in this report are aimed at *detecting* differences between two versions of NAV. However, they do not help in categorizing, characterizing, analyzing, or otherwise managing the differences. In particular, the tools do not identify any common pattern of code modifications, even if there are any. Part of the reason for this is, as discussed above, due to the lack of known patterns to search for. We have identified the need for tools that allow developers to get an understanding of a large code base (of which some parts may be derived versions of others); the use of such tools in the presence of Microsoft Dynamics NAV is future work.

1.4 Outline of the report

The rest of this report is organized as follows. In Chapter 2 we review the *less constrained tree-to-tree edit problem* and argue that solutions to this problem are good representations of differences between two pieces of source code. In Chapter 3, we describe an implementation of a tool (`astalign`) that uses Jiang, Wang, and Zhang's [7] algorithm to solve such less constrained tree-to-tree edit problems. This tool can be used to compute differences between any two trees (and not only source code) and it can either be invoked as a command-line tool or be used as a library by other applications. In Chapter 4 we use the `astalign` library to implement an application (`navdiff`) that compute differences between comparable code pieces from different version of Microsoft Dynamics NAV. (Both `astalign` and `navdiff` are implemented in C#.) In Chapter 5 we present a set of preliminary analyzes of differences between versions of Microsoft Dynamics NAV. In Chapter 6 we conclude.

Chapter 2

Tree alignments

The tree-alignment problem is one of several *tree edit problems* which identify minimal sets of differences between two hierarchically structured documents and that measure the minimal distance between two such documents in terms of how much they differ. In this chapter we present the notion of the tree alignment problem, which is a particular instance of the tree edit problem.

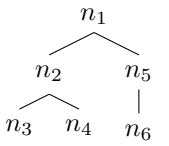
2.1 Preliminary definitions

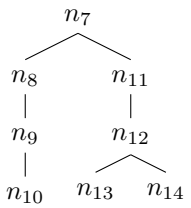
A rooted tree is a triple $\langle N, E, r\rangle$ consisting of a set N of nodes, a subset E of $N \times N$ of edges, and a distinguished root node $r \in N$. (Formally, E is an injective binary relation such that, for all $n \in N$, $(r, n) \in E^*$, where E^* is the reflexive, transitive closure of E.) A tree is *ordered* if, for each node $n \in N$, there exists a total ordering $\leq$ among its children $\{m \mid (n, m) \in E\}$. A tree is *labeled* if there is a mapping ℓ from nodes to a set of labels L. A *preorder traversal sequence* lists the nodes (or labels) of a tree in accordance with their preorder numbering.

2.2 Definition of tree alignments

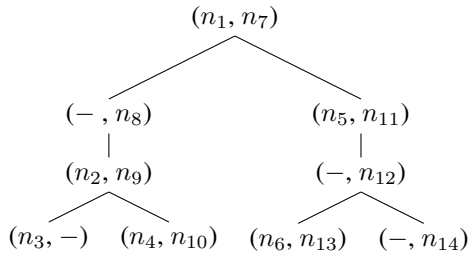
Let $T_1 = \langle N_1, E_1, r_1\rangle$ and $T_2 = \langle N_2, E_2, r_2\rangle$ be two ordered rooted trees. An *alignment* of T_1 and T_2 is a labeled, ordered, and rooted tree A whose labels are from $(N_1 \times N_2) \cup (N_1 \times \{-\}) \cup (\{-\} \times N_2)$ (where "$-$" is a special blank symbol) such that the preorder traversal sequence of A projected to the first (second, resp.) component can be obtained from the preorder traversal sequence of T_1 (T_2, resp.) by only inserting blanks. We require that the root of the alignment is labeled by (r_1, r_2). Intuitively, an alignment of T_1 and T_2 is a tree that *contains* both T_1 and T_2, in the sense that erasing the first (second, resp.) component of the pairs in the nodes of A and subsequently contracting paths involving "$-$" results in T_2 (T_1, resp.).

As an example, the set of possible alignments of the two trees

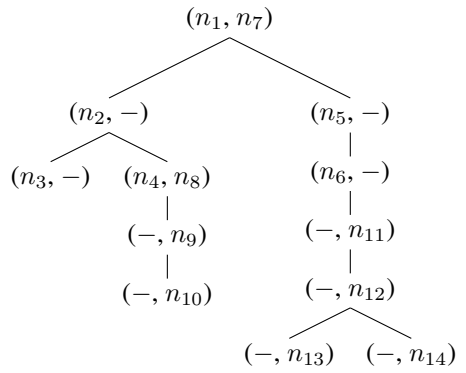
$T_1 =$ 

and $T_2 =$ 

with preorder traversal sequences n_1, n_2, n_3, n_4, n_5, n_6 and n_7, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{12}, n_{13}, n_{14} includes

$A =$ 

with preorder traversal sequence (n_1, n_7), $(-, n_8)$, (n_2, n_9), $(n_3, -)$, (n_4, n_{10}), (n_5, n_{11}), $(-, n_{12})$, (n_6, n_{13}), $(-, n_{14})$ and

$A' =$ 

with preorder traversal sequences (n_1, n_7), $(n_2, -)$, $(n_3, -)$, (n_4, n_8), $(-, n_9)$, $(-, n_{10})$, $(n_5, -)$, $(n_6, -)$, $(-, n_{11})$, $(-, n_{12})$, $(-, n_{13})$, $(-, n_{14})$.

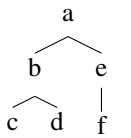
The nodes in an alignment are interpreted as operations that map the first tree into the second. A node (n_1, n_2) denotes an *update* of the label of node n_1 to the label of node n_2, a node $(-, n_2)$ denotes an *addition* of node n_2, and a node $(n_1, -)$ denotes a *deletion* of node n_1.

Nodes in an alignment are associated with a *cost* of performing the corresponding operations. A cost function over the nodes in the alignment may take the labels of the

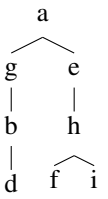
corresponding nodes in the original trees into account. We let c be a mapping from $(N_1 \times N_2) \cup (N_1 \times \{-\}) \cup (\{-\} \times N_2)$ to real-valued costs. The cost of an alignment is the sum of the costs of its nodes. An *optimal alignment* of two trees is an alignment with least cost. (There may be more than one optimal alignment.) Intuitively, the alignment with the least cost is the one that maps the first tree into the second using the cheapest set of operations. The purpose of a *tree alignment algorithm* is to find the optimal alignment of two trees.

Let $L = \{\text{a, b, c, } \ldots\}$ and $\ell(n_1) = \ell(n_7) = \text{a}$, $\ell(n_2) = \ell(n_9) = \text{b}$, $\ell(n_3) = \text{c}$, $\ell(n_4) = \ell(n_{10}) = \text{d}$, $\ell(n_5) = \ell(n_{11}) = \text{e}$, $\ell(n_6) = \ell(n_{13}) = \text{f}$, $\ell(n_8) = \text{g}$, $\ell(n_{12}) = \text{h}$, and $\ell(n_{14}) = \text{i}$. Then the two trees defined above can be drawn with their labels, as follows. (This notation is often preferred when labels uniquely define nodes.)

$T_1 =$

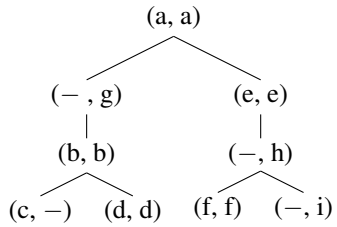


and $T_2 =$



If we assume that $c(n, n') = 0$ when $\ell(n) = \ell(n')$, $c(n, n') = 2$ when $\ell(n) \neq \ell(n')$ and both $c(-, n) = 1$ and $c(n, -) = 1$, then alignment A has cost 4 and alignment A' has cost 12. Alignment A is an optimal alignment of the trees T_1 and T_2 with respect to this cost function. With labels instead of nodes, A looks as follows.

$A =$



Notice that the number of nodes in an alignment of two trees with nodes N_1 and N_2 is between $\max(|N_1|, |N_2|)$ and $|N_1| + |N_2|$ and that both of the original trees can be reconstructed given the alignment only.

The notion of tree alignments and an algorithm to find optimal alignments was introduced by Jiang, Wang, and Zhang [7]. The time complexity of the algorithm is in $\mathrm{O}(|N_1| \times |N_2| \times (D_1 + D_2)^2)$ for trees with nodes N_1 and N_2 and where the degree D_i of tree T_i is the maximum number of children of any node in the tree.

2.3 Alternative definitions of tree alignments

There are several alternative definitions of tree alignments.

- Tree alignments may be characterized as the result of first adding blank nodes in the two trees until they have the same structure, and the overlaying the trees [1, 7].

- Tree alignments (and other tree-to-tree correction problems) can also be defined in terms of a relation between nodes in one tree and nodes in the other. Formally, the triple $\langle M, T_1, T_2 \rangle$ is an *ordered edit distance mapping* [1, 21] from $T_1 = \langle N_1, E_1, r_1 \rangle$ to $T_2 = \langle N_2, E_2, r_2 \rangle$ when M is a subset of $N_1 \times N_2$ such that for any $(n_1, m_1), (n_2, m_2) \in M$,

  1. $n_1 = n_2$ if and only if $m_1 = m_2$,
  2. $\mathrm{pre}(n_1) < \mathrm{pre}(n_2)$ if and only if $\mathrm{pre}(m_1) < \mathrm{pre}(m_2)$, where $\mathrm{pre}(n)$ is the preorder number associated with node n, and
  3. $n_2 \prec n_1$ if and only if $m_2 \prec m_1$, where $n \prec n'$ if and only if n' is an ancestor of n.

  Conditions 1–3 gives rise to a notion of (unconstrained) tree-to-tree edit problem [21]. An alignment corresponds to a *less constrained* edit [10, 12], which, in addition to conditions 1–3, require that for $(n_1, m_1), (n_2, m_2), (n_3, m_3) \in M$ such that none of n_1, n_2, and n_3 are ancestors of the others,

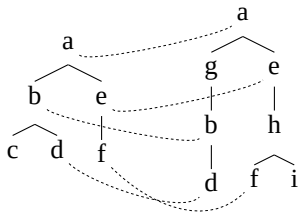
  4. $\mathrm{lca}(n_1, n_2) \preceq \mathrm{lca}(n_1, n_3) = \mathrm{lca}(n_2, n_3)$ if and only if $\mathrm{lca}(m_1, m_2) \preceq \mathrm{lca}(m_1, m_3) = \mathrm{lca}(m_2, m_3)$, where $\mathrm{lca}(n, n')$ denotes the *least common ancestor* of nodes n and n'.

  (Less constrained edits lift one of the restrictions of *constrained* edits [18, 25], a notion we do not consider further in this report.)

  As an example, alignment A above corresponds to the mapping

$$\langle T_1, T_2, \{(n_1, n_7), (n_2, n_9), (n_4, n_{10}), (n_5, n_{11}), (n_6, n_{13}\} \rangle$$

  which can be draw as the following diagram.



- Tree alignments are equivalent to *ordered sequences of edit operations* in which every addition precede all deletions [1, 7].

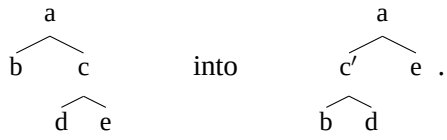
  Such an ordered script is an operational description of how one tree can be transformed into the other. In that sense, edit scripts correspond to the output of a traditional DIFF. An ordered edit script must include information about the location at which an operation is applied. An ordered edit script corresponding to alignment A above might be specified as follows.

  1. Add node n_8 labeled g under node n_1/n_7,
  2. add node n_{12} labeled h under node n_5/n_{11},
  3. add node n_{14} labeled i under node n_{12}, and then
  4. delete node n_3.

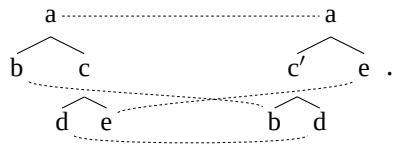
  Notice that if deletions were allowed to precede additions, then the ordered edit script

  1. Delete node c, and then
  2. add node c′ between node a and nodes b and d.

  would transform the tree



  The corresponding edit mapping is $\{(\mathrm{a},\mathrm{a}),(\mathrm{b},\mathrm{b}),(\mathrm{d},\mathrm{d}),(\mathrm{e},\mathrm{e})\}$ which we may display as the following diagram.



  An alignment would have to have a node (b,b) under a node $(-,\mathrm{c}')$ and a node (e,e) under a node $(\mathrm{c},-)$. But, according to the diagram, it should also have a node (d,d) under *both* nodes $(-,\mathrm{c}')$ and $(\mathrm{c},-)$, which is not possible.

2.4 Tree alignments as source code diffs

There are several variations in the design that must be addressed when using a tree alignment as a vehicle for detection changes in source code.

A programming language is associated with several different *notions of equivalences* among its programs. Listed in decreasing order of "strength", some of the most common notions of equivalences are as follows.

Lexical equivalences: The representation of two programs contains identical sequences of symbol.

Syntactical equivalences: The abstract syntax trees of two programs are identical.

α-equivalence: Syntactical equivalence modulo renaming of variables.

Semantic equivalence: Two programs have the same behavior.

UNIX diff is lexical, with lexical entities being the *lines* of the source programs. Detecting changes between programs is a generalization of deciding whether program are equivalent. Therefore, a particular change-detection tool employs a particular underlying notion of equivalence. Another implication of this is that identifying semantic changes between two programs is undecidable.

An equivalence relation can be "weakened" by composing it with a *normalization* phase that maps programs to canonical elements of a "weaker" equivalence class. For example α-equivalence can be emulated by syntactical equivalence by consistently renaming variables.

It seems evident that tree alignments should be based on the abstract syntax trees of two source program, rather than on their concrete syntax. Specialized notions of equality may also be considered. For syntactic equivalence, it might, for example, be relevant to consider two programs to be equal when their abstract syntax trees are identical modulo inconsequential reordering of declarations (such as the fields of a class) or statements.

Another factor that affects the detection of changes between documents is the *granularity* of changes. This is the extent of the smallest entities that can be modified separately. Typical examples are characters, words or tokens, or lines. When comparing source code, the most relevant entities are probably tokens, expressions, or statements but larger entities such as methods, classes, or modules might also be considered. For imperative programming languages such as derivatives of C or Pascal, comparing source programs line by line corresponds in many cases to comparing statement by statement. The success of this approach is demonstrated by the widespread use of UNIX diff in software repositories.

Modular programming language enable a particular optimization when detecting changes. If we let "module" denote any named block of source code (such as methods,

procedures, classes, or namespaces) and if we assume that modules are never renamed and that code is never moved between modules, then all changes are "intramodular" and can be detected by applying a change-detection tool only on modules of the same (fully qualified) name in the two programs being compared. This approach will greatly improve the performance of change-detection tools that use comparison algorithms that are non-linear. (It should be mentioned that the assumption that procedures are never renamed has been questioned [4], but that we find that this property holds in NAV.)

Chapter 3

Implementation of a tree alignment algorithm

This chapter describes `align.dll`, a general-purpose library implementing Jiang, Wang, and Zhang's tree alignment algorithm [7], and `astalign`, a stand-alone application demonstrating how the library may be used.

Two aspects of the library can be configured by supplying additional code: The costs of adding, deleting, updating, and copying nodes and labels (consult the interface `ICostProvider` below) and a set of actual actions to apply during a traversal of a generated alignment (see interface `IActionListener` below).

The library and the stand-alone application are both written in C#.

3.1 A tree alignment library

The tree-alignment library contains classes representing trees (`Tree`, `Node`), intervals over sequences of trees (`Interval`), and alignments (`Alignment`) and a main class (`Align`) that builds an internal representation of the alignment of two trees. In addition, two interfaces control how the library may be configured (`ICostProvider`, `IActionListener`).

3.1.1 Class `Node`

This inductively defined class provides an implementation of nodes of labeled ordered trees. Objects of this class are never created by the alignment algorithm. The labels of trees represented as `objects`. They are never generated or modified by the alignment algorithm.

Methods and properties of class `Node`:

- `public Node(object label, params Node[] children)`

(Constructor) Creates a node with a given label. The constructor may take child nodes as argument. Thus, trees can be constructed bottom-up. Further children may be added (to the right of existing children) using the `Add` method. Thus, trees represented as `Node`s can also be constructed top-down.

- `public void AddChild(Node n)`

  Adds a sub tree to the right of the right-most children of this `Node`.

- `public override string ToString()`

  Returns a written representation of the subtree under this `Node`.

- `public object Label`

  (Readonly property) Yields the label of this `Node`.

- `public int Size`

  (Readonly property) Yields the size of the subtree rooted under this `Node`.

- `public int Degree`

  (Readonly property) Yields the number of children of this node (its degree).

- `public int Depth`

  (Readonly property) Yields the depth of the subtree rooted under this `Node`.

- `public int OutDegree`

  (Readonly property) Yields the maximum degree among all subtrees rooted under this `Node`.

- `public IEnumerable<Node> Children`

  (Readonly property) Yields an enumeration of the children of this `Node`.

3.1.2 Class `Tree`

This class provides a sequentialized view of the nodes of a tree. In this view, nodes are indexed by their postorder number, starting from 1. This is the datastructure that the alignment algorithm operates on. The labels of trees represented as `objects`. They are never generated or modified by the alignment algorithm.

To construct a `Tree`, one must first construct a tree represented by a `Node` and then pass this representation to the constructor of the `Tree` class. (In earlier version of the library, the internal `Tree`s used by the algorithm were unsuitable for use in parsers. The distinction between class `Tree` and class `Node` has since been blurred.)

Methods and properties of class `Tree`:

- `public Tree(Node root)`

  (Constructor) Creates a new `Tree` given the root of a tree of `Node`s.

- `public object Label(int i)`

  Returns the label of node `i` (using postorder numbering).

- `public int NumberOfChildren(int i)`

  Returns the number of children of node `i` (using postorder numbering).

- `public int[] Children(int i)`

  Returns an array of the postorder numbers of children of node `i` (using postorder numbering).

- `public bool Equals(Tree other)`

  This method compares two trees. This is a fast way of filtering out pairs of trees that are identical, before applying the (rather expensive) alignment algorithm. (This method is never invoked by the alignment library.)

- `public void dump()`

  Prints a representation of this `Tree` to the standard output port.

- `public object Root`

  (Readonly property) Yields the label of the root of this `Tree`.

- `public int Size`

  (Readonly property) Yields the number of nodes in this `Tree`.

- `public int Depth`

  (Readonly property) Yields the depth of this `Tree`.

- `public int OutDegree`

  (Readonly property) Yields the degree of this `Tree`.

3.1.3 Class `Interval`

This static class provides a bijective mapping from

$$\{(i,j) \mid 1 \leq i \leq j < max\}$$

to integers in the range $[1; \frac{1}{2}max(max+1)]$. In the alignment algorithm, pairs (i,j) denote intervals over the children of a node with max children. The class `Interval` enables such intervals to stored in a one-dimensional array and be indexed by integers.

Methods and properties of class `Interval`:

- `public static int Index(int max, int i, int j)`

  This method provides the mapping from $\{(i,j) \mid 1 \leq i \leq j < max\}$ to $[1; \frac{1}{2}max(max+1)]$. This method has the property that

  $$\texttt{index}(max, i, j+1) = \texttt{index}(max, i, j) + 1$$

  for $1 \leq i \leq j < max$. (The alignment algorithm relies on this property. If the `Index` method is ever modified, this property must be preserved.)

- `public static int Size(int max)`

  This method returns $\frac{1}{2}max(max+1)$.

3.1.4 Class `Alignment`

Object of this class are representations of the alignment of two trees. Such an alignment is itself a tree where each node contains a pair of an old node (or `null`) and a new node (or `null`). If the old node part is `null`, then this alignment node designates an addition; and dually, if the new node part is `null`, then this alignment node designates a delete. (Either the old node or the new node must be non-`null`.) Objects of this class are generated by the alignment algorithm.

Methods and properties of class `Alignment`:

- `public Alignment(object` $\texttt{n}_1$`, object` $\texttt{n}_2$`, Alignment[] ch)`

  Constructs an `Alignment` node with a given sequence of children. Either $\texttt{n}_1$ or $\texttt{n}_2$ may be `null`, but not both. This method is used by the library to construct an `Alignment`.

- `public object info`

  This is an unused field. It may be used during traversals of the alignment to hold synthesized attributes. This field is never accessed by the library.

- `public object OldLabel`

  (Readonly property) The label (or `null`) from the old tree associated with this `Alignment` node. Notice that this property yields a label, not a `Node`.

- `public object NewLabel`

  (Readonly property) The label (or `null`) from the new tree associated with this `Alignment` node. Notice that this property yields a label, not a `Node`.

- `public int Degree`

  (Readonly property) Yields the number of children of this alignment node.

- `public Alignment this[int i]`

  (Readonly property) Yields the ith children (starting from 0) of this alignment node.

3.1.5 Class `Align`

This class performs the alignment of two `Trees`.

Methods and properties of class `Alignment`:

- `public Align(ICostProvider costs, Tree` t_1`, Tree` t_2`)`

  (Constructor) Aligns the `Trees` t_1 and t_2 with respect to the given costs.

- `public void Execute(IActionListener listener)`

  Traverse the alignment. For each alignment node visited, invoke the corresponding method in the supplied `IActionListener`.

- `public Alignment Alignment()`

  Construct an `Alignment` tree.

3.1.6 Interface `ICostProvider`

This interface contains the signatures for cost functions.

- `int CostOfDelete(object l)`

  This method should return the cost of deleting an old node with label `l`.

- `int CostOfInsert(object l)`

  This method should return the cost of adding a new node with label `l`.

- `int CostOfCopy(object l)`

  This method should return the cost of copying, without modifying, a node with label `l` from the old to the new tree. The alignment algorithm decides if two labels are identical by invoking the `object.Equals` method. Custom labels with custom `Equals` predicates may be defined as subclasses of `object`.

- `int CostOfUpdate(object` l_1`, object` l_1`)`

  This method should return the cost of updating a node by changing its label from l_1 to l_2.

3.1.7 Interface `IActionListener`

This interface defines the signatures of the operations used when 'executing edit scripts' associated with an alignment. There are four groups of methods, corresponding to the four edit operations. Each group consists of one method invoked when entering the subtree rooted under the alignment node and another invoked when leaving the subtree rooted under the alignment node.

- `void BeginCopy(object` l_1`, object` l_2`)`
- `void EndCopy(object` l_1`, object` l_2`)`

  These method will be invoked when a node has been copied from the old tree to the new without modification.

- `void BeginUpdate(object` l_1`, object` l_2`)`
- `void EndUpdate(object` l_1`, object` l_2`)`

  These method will be invoked when a node has been changed. (This is like copying, but with modification to the node label.)

- `void BeginInsert(object l)`
- `void EndInsert(object l)`

  These method will be invoked when a node has been added into the new tree.

- `void BeginDelete(object l)`
- `void EndDelete(object l)`

  These method will be invoked when a node has been deleted from the old tree.

3.1.8 Example usage

The following class implements cost functions corresponding to those used to align the trees in Section 2.

```
class CostProvider : Align.ICostProvider {
  public int CostOfCopy(object l) { return 0; }
  public int CostOfUpdate(object l1, object l2) { return 2; }
  public int CostOfInsert(object l) { return 1; }
  public int CostOfDelete(object l) { return 1; }
}
```

The code snippet below then constructs two trees (represented as `Nodes`), aligns the two trees with respect to the cost functions, and displays the alignment.

```
...
Align.Node n1 =
  new Align.Node("a",
    new Align.Node("b",
      new Align.Node("c"),
      new Align.Node("d")),
    new Align.Node("e",
      new Align.Node("f")));

Align.Node n2 =
  new Align.Node("a",
    new Align.Node("g",
      new Align.Node("b",
        new Align.Node("d"))),
    new Align.Node("e",
      new Align.Node("h",
        new Align.Node("f"),
        new Align.Node("i"))));

Align.Align a =
  new Align.Align(new CostProvider(),
                  new Align.Tree(n1),
                  new Align.Tree(n2));

Display(0, a.Alignment());
```

The method `Display` writes a representation of the alignment tree to the console. It uses indentation to distinguish nodes at different levels in the alignment tree. This method can be implemented as follows.

```
static void Display(int depth, Align.Alignment a) {
  for (int i = 0; i < depth; i++)
    System.Console.Write("  ");

  System.Console.WriteLine("({0}, {1})",
    (a.OldLabel == null) ? "-" : (string) a.OldLabel,
    (a.NewLabel == null) ? "-" : (string) a.NewLabel);

  for (int i = 0; i < a.Degree; i++)
    Display(depth + 1, a[i]);
}
```

Running the code snippet above results in the following output. (Compare with alignment A in Section 2.)

```
(a, a)
  (-, g)
    (b, b)
      (c, -)
      (d, d)
  (e, e)
    (-, h)
      (f, f)
      (-, i)
```

3.2 A standalone tree alignment application

Experiments with tree alignments can be conducted using `astalign`, a stand-alone application not unlike the example program from above.

`astalign` accepts the following options specifying constant costs for adding, deleting, updating, and copying nodes: "`-i` v", "`-d` v", "`-u` v", and "`-c` v". Here, the vs must be positive integers. They default to 1, 1, 1, and 0 respectively.

The input to `astalign` are two trees written in fully parenthesized notation (parentheses around leaves may be omitted) and whose labels are strings of non-whitespace characters. The input can be described by the following grammar.

$$
\begin{aligned}
\textit{tree} &\rightarrow \textit{label} \\
\textit{tree} &\rightarrow \text{`(' } \textit{label}\ \textit{tree}^{*}\ \text{`)'} \\
\textit{label} &\rightarrow \textit{non-empty sequence of non-whitespace characters}
\end{aligned}
$$

3.2.1 Example usage

The trees from Section 2 are aligned in the following session. (The parts in bold-face font is typed in. The remaining parts are output by the application.)

```
> astalign.exe -u 2
(a (b c d) (e f))
(a (g (b (d))) (e (h (f) (i))))
COST OF ALIGNMENT: 4

EDIT SCRIPT:
```

```
COPY a
  INSERT g
    COPY b
      DELETE c
      COPY d
  COPY e
    INSERT h
      COPY f
      INSERT i

ALIGNMENT TREE:
(a, a)
  (-, g)
    (b, b)
      (c, -)
      (d, d)
  (e, e)
    (-, h)
      (f, f)
      (-, i)
```

(Compare again with alignment A in Section 2. And notice that, for illustrative purpose, the first tree is typed in without parentheses around leaves but the second with parentheses around leaves.)

Chapter 4

Implementation of a change-detection tool for NAV

This chapter describes `navdiff`, a tool for comparing two versions of Microsoft Dynamics NAV. The tool uses the alignment library presented in Chapter 3 to compare code pieces from the two versions.

4.1 Design

The purpose of `navdiff` is to assist in detecting and characterizing differences between two versions of NAV. The tool is not limited to comparing one version of NAV with one of its derivatives, but may be applied to any two versions of NAV.

4.1.1 Input

All business objects from a version of NAV can be output in a textual format known as an *export*. It is convenient to let the change-detection tool operate on this format using standard parsing methodology. (An alternative is to integrate the tool with the existing integrated development environment.) In an attempt to reduce the amount of memory required when comparing two version of NAV, the tool avoids parsing (and storing) the entire exports of the two version being compared. This is enabled by a one-pass preprocessing phase that *chops* one export (corresponding to the *old* version of NAV) into smaller files containing one (or more) procedures or triggers. (If one old version must to be compared to several new versions, it need only be chopped once.) The procedures and triggers of the other export (corresponding to the *new* version of NAV) need not be chopped: A one-pass traversal of the new export parses procedures and triggers and constructs their abstract syntax trees one by one. Once constructed, the abstract syntax of a new procedure or trigger is compared with the corresponding old abstract syntax tree stored in a file. After comparison, both abstract syntax trees can

be discarded. The total amount of memory required when comparing two versions of NAV is then proportional to the amount of memory required by the alignment tool when operating on the largest procedure.

To facilitate this scenario, `navdiff` defines an interface `ICodeProvider` of by classes that can supply abstract syntax trees. This interface is implemented by class `FileCodeProvider` that reads abstract syntax trees from chopped files and by class `BufferedCodeProvider` that reads abstract syntax trees from an export.

4.1.2 Output format

The purpose of `navdiff` is not to assist in managing, examining, or characterizing the differences that are detected. (But see section 4.1.3 below.) Instead it dumps alignments as presented in Section 2 to a relational database. A diagram describing the table schemas is shown in Figure 4.1 on the next page. The content of the tables are summarized below.

Alignments: An alignment represents the result of aligning two code pieces with the same path from different versions of NAV. A row in the `Alignments` table contains the names of the old and new version of NAV being compared, the code path (key), and the root note (key) of the alignment tree.

CodePaths: A code path represents the fully qualified path identifying a code piece within NAV. A row in the `CodePaths` table contains a unique ID (key), a (possibly `null`) ID of a parent code path, and a name.

AlignNodes: An alignment node represents a node in an alignment of two code pieces. A row in the `CodePaths` table contain a unique ID (key), a (possibly `null`) ID of the parent alignment node, a (possibly `null`) ID of the left sibling, and (possibly `null`) IDs of old and new labels (called lines). (For experimental purposes, a row also contains a string representing the path from the root of this alignment node to the current node. This is a derived value.)

A `null` old label means that the alignment node represents an addition. A `null` new label means that the alignment node represents a deletion. At most one of the old or the new label may be `null`.

An alignment node has a parent alignment node if and only if there are no alignments referring to it.

Lines: A line represents a fragment of characters from a code piece. A row in the `Lines` table contains a unique ID (key), a line number of the occurrence of the fragment in NAV export, and an ID of the actual content.

Content: Contains chunks of characters from NAV export files.

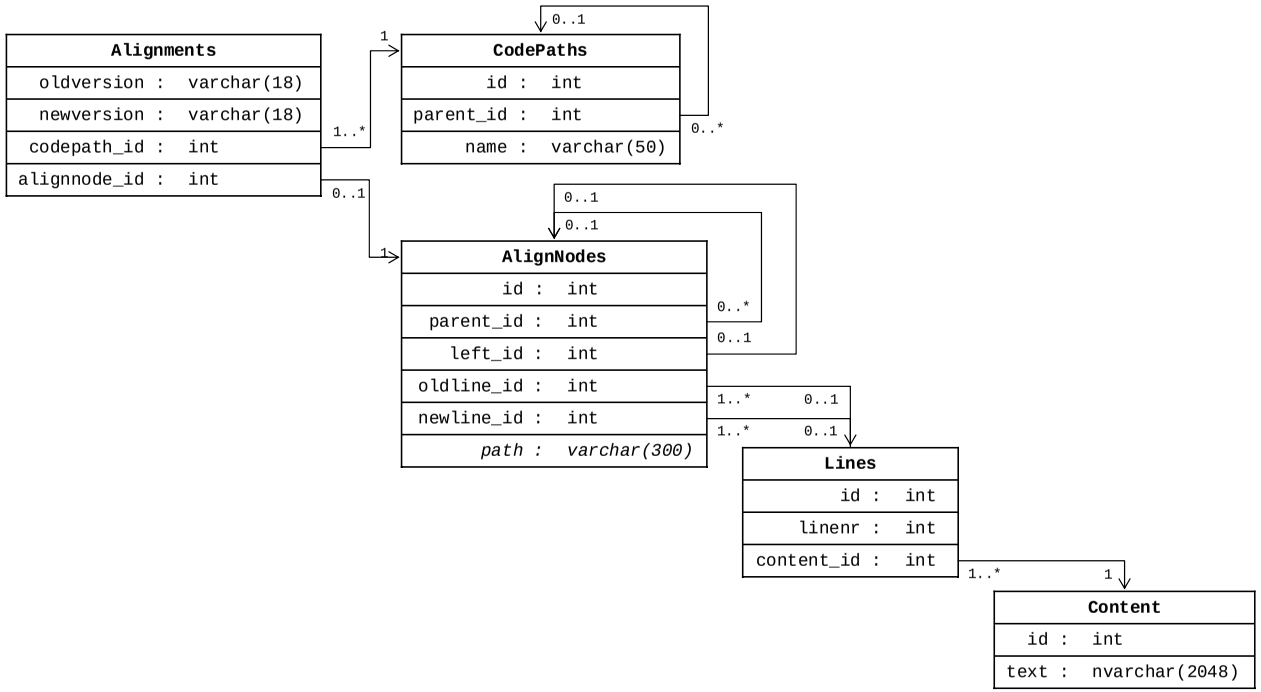


Figure 4.1: Schema describing the output of `navdiff`

These tables allows differences between several different pairs of versions of NAV to be stored in one database. This is essential for the identification of program points that are subject to modification in several derived versions of NAV.

These tables encode tree-like structures (code paths and alignments) using the *Adjacency List Model* where a child contains an ID of its parent [2]. Although this is a natural representation of hierarchies in relational databases, some queries involving such a representation cannot be expressed as pure SQL queries. In particular, due to the lack of general recursion techniques in pure SQL queries, simple problems such as finding the size of an alignment, finding the root of an alignment, or finding code pieces that contains modifications cannot be expressed. Limited forms of recursive queries are supported by the `connect by` clause of Oracle's SQL extension and by the Common Table Expressions of Transact SQL [16].

There are at least two alternative representations that support standard (non-recursive) SQL queries over hierarchical data [2]. The first, the *Path Enumeration Model* or *Materialized Path Model*, stores in each node the path from the root of the hierarchy to that node, as a string of either edge labels or (if siblings labels are always distinct) node labels, separated by a path separator. In such a model, queries involve predicates over these paths. Many natural queries can be expressed using predicates that involve matching *regular expression* against paths. Unfortunately, general regular expressions matching is not supported by standard SQL. To successfully use the limited matching capabilities of SQL, it seems necessary, for example, to uniformly store edge or node labels using a fixed number of characters in the path string.

The second alternative, the *Nested Sets Model*, associates with each node an interval represented by two numbers such that (1) the interval associated with any descendant of a node is a subinterval of the interval associated with the node itself and (2) siblings have disjoint intervals. Queries over a nested set model use predicates that involve simple numerical comparisons.

The additional information supporting the Path Enumeration Model and the Nested Set Model can be generated by recursively defined stored procedures.

4.1.3 Alternative output format

Since the output to a relational database is experimental, `navdiff` can also output differences in a text-based format. More precisely,

1. if `navdiff` detects a consecutive sequence of statements that have been deleted, then it outputs one line listing the line number in the original version (typically W1) of the first of the deleted statements;

2. if `navdiff` detects a consecutive sequence of statements that have been added, then it outputs one line listing the line number in the original version of the immediately preceding line that was not added; and

3. if `navdiff` detects a statement that has been updated, then it outputs one line listing the line number in the original version of the updated statement.

Notice that lines that are added do not have a line number in the original version. `navdiff` numbers lines relative to the exported file, not relative to the surrounding code piece. Therefore all line numbers are global. As an example, here is an excerpt from comparing two versions of NAV, with the (global) line numbers in the left column.

```
...
0514237 node added (new line = 0584420)
0514237 node deleted
0530147 trees added (new line = 0600719)
0531183 node deleted
0531284 nodes updated (new line = 0601906)
0531301 nodes updated (new line = 0601923)
0531332 nodes updated (new line = 0601954)
0531355 nodes updated (new line = 0601977)
0534582 trees added (new line = 0605290)
...
```

As indicated by the first two lines of this excerpt, some updates are characterized as an addition followed by a deletion. (Thus, the meaning of such a pair of operations is not to add a line and then delete it afterwards.)

4.2 Implementation

The sections below present the most important classes and methods of the tool. A diagram describing the most important relationsships between these classes is presented in Figure 4.2 on the following page.

4.2.1 Class `CodePath`

Objects of this class represents paths identifying code fragments (procedures or triggers) in a version of NAV. These paths can be compared efficiently, so they may serve as keys in tables, sets, and dictionaries. A codepath is either the root, or a parent codepath augmented with a short name:

Root codepaths are represented as the `null` reference but are nevertheless real codepaths. (In other word, some legal codepaths are `null`, rather than references to objects.) Therefore, almost all operations on codepaths are done via static member functions.

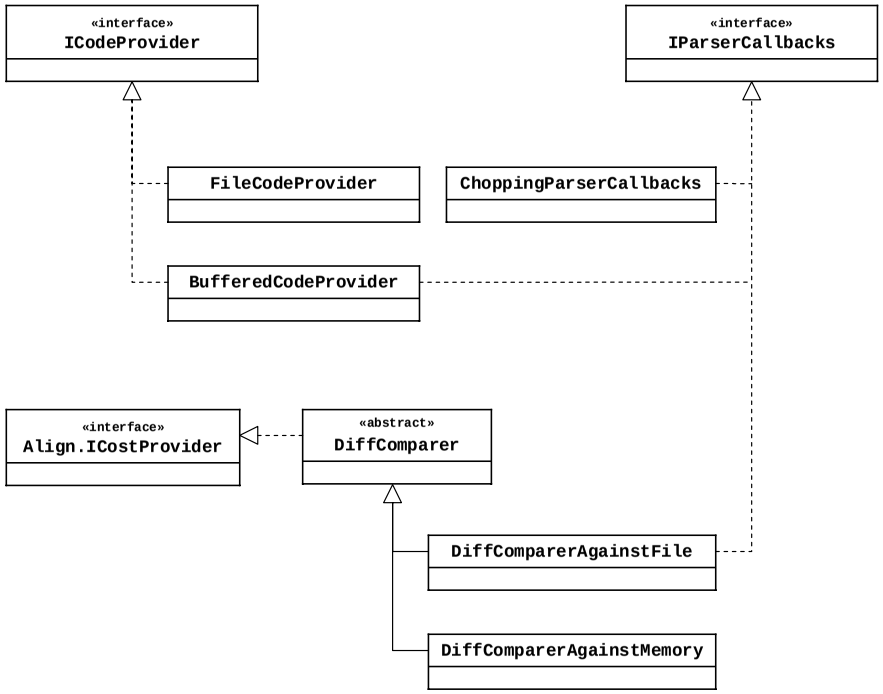


Figure 4.2: Common classes of `navdiff`

Before using this class (or between two comparisons), invoke the method `Reset()`. The class property `Roots` and the instance property `Children` can be used to recursively retrieve all codepaths constructed (since the last `Reset()`).

Methods and properties of class `CodePath`:

- `public static IEnumerable<CodePath> Roots`

  (Readonly property) Yields an enumeration of the root `CodePath`s installed since the last `Reset`.

- `public static void Reset()`

  Empties the set of installed root `CodePath`s.

- `public CodePath Parent`

  (Readonly property) Yields the parent element (itself a `CodePath`) of this `CodePath`.

- `public string Name`

  (Readonly property) Yields the short name of this (non-root) `CodePath`. This is the name of the leaf identified by the `CodePath`.

- `public int Length`

  (Readonly property) Yields the length of this `CodePath`.

- `public IEnumerable<CodePath> Children`

  (Readonly property) Yields an enumeration of the children of this `CodePath`.

- `public static CodePath MakeCodePath(params string[] names)`

  Constructs a `CodePath` from an array of names.

- `public static CodePath Add(CodePath p, string name)`

  Adds a leaf element to an existing (but possibly `null`) `CodePath`. The `CodePath` terminating at the leaf is returned.

- `public static string[] PathComponents(CodePath p)`

  Yields an array of the path elements of a given `CodePath`.

4.2.2 Interface `ICodeProvider`

This interface is implemented by classes that can provide access to code fragments (procedures or triggers) from a version of NAV. Each code fragment is identified by its `CodePath`.

Methods specified by interface `ICodeProvider`:

- `bool Contains(CodePath path)`

  This method must return `true` if and only if the given `CodePath` exists in this code provider.

- `Align.Tree this[CodePath path]`

  (Property) This property must yield the abstract syntax tree associated with the given `CodePath`.

- `IEnumerator<KeyValuePair<CodePath, Align.Tree>> GetEnumerator()`

  This method must yield an enumerator producing all `CodePath`s (and their associated abstract syntax tree) provided by this `ICodeProvider`.

4.2.3 Class `FileCodeProvider`

This class implements interface `ICodeProvider`.

Objects of this class provides random access to chopped versions of NAV file hierarchies without loading the entire NAV version into memory.

4.2.4 Class `BufferedCodeProvider`

This class implements interfaces `IParserCallbacks` and `ICodeProvider`.

Objects of this class provides random access to code in a NAV version by parsing the entire version into memory. It is not recommended to use this class on large NAV databases. Instead use a `FileCodeProvider` on a chopped NAV version or use `DiffCompareAgainstFile` to traverse an NAV databases.

4.2.5 Class `DiffComparer`

This class is abstract and implements interface `Align.ICostProvider`. This class serves as superclass for classes that can compare two version of NAV.

Subclasses of this class must implement the method `Comparer()` so that it traverses all code in both versions, invokes methods `Enter()` and `Leave()` when entering and leaving nodes in the NAV hierarchy, and invokes method `CompareTrees()` on code from both versions. This traversal must be a post-order traversal. That is, it must respect the tree structure of the NAV hierarchy and it must process children before parents. There are two subclasses of this class,

`DiffComparerAgainstFile` and

`DiffComparerAgainstMemory`.

Before using objects of this class, the `ActionListener` property must be set to an appropriate value.

This class implements a specific cost function.

4.2.6 Class `DiffComparerAgainstFile`

This class extends class `DiffComparer` and implements interface `IParserCallbacks`.

Objects of this class can compare two version of NAV, one of which comes from a raw text file (that is, a file exported by NAV) and the other which is represented by an `ICodeProvider`.

Methods and properties of class `DiffComparerAgainstFile`:

- `DiffComparerAgainstFile(ICodeProvider old_code_provider, string new_code_file)`
- `DiffComparerAgainstFile(string old_code_file, ICodeProvider new_code_provider)`

4.2.7 Class `DiffComparerAgainstMemory`

This class extends class `DiffComparer`.

Objects of this class can compare two version of NAV, both of which are represented as `ICodeProvider`s.

4.2.8 Class `ChoppingParserCallbacks`

This class implements interface `IParserCallbacks`.

Objects of this class are used when chopping an NAV export into smaller files. When constructing a `ChoppingParserCallbacks`, the "depth" d at which to split exports is specified. A procedure P at code path $M_1/\cdots/M_k$ will be stored in file M_d under directory $M_1/\cdots/M_{d-1}$ as illustrated as follows.

$$\underbrace{M_1/M_2/\cdots/M_{d-1}}_{\text{Directory}}/\underbrace{M_d}_{\text{File}}/\underbrace{M_{d+1}/\cdots/M_k}_{\text{File internal}}$$

Several procedures, namely those whose paths share the prefix $M_1/\cdots/M_{d-1}/M_d$, will be stored in the same file.

4.3 Command-line options recognized by `navdiff`

The following is a list of the most important options recognized by `navdiff`.

`--chop` *input-file output-file*

This option instructs `navdiff` to chop (or split) an exported NAV file into smaller pieces.

`--chop-level` *level*

This options sets the level at which an exported NAV file should be chopped. Its default (2) results in one directory for each kind of business object (Report, Form, Menusuite, Codeunit, Dataport, XMLport, Table) with each directory containing one file for each business object of that kind.

`--db-initialize` *user-name database-name*

Initializes a SQLServer database *database-name* with user name *user-name*.

`--db-delete` *user-name database-name*

Removes a SQLServer database *database-name* with user name *user-name*.

`--db` *user-name database-name old-version new-version*

Instructs `navdiff` to dump diffs to SQLServer database *database-name* with user name *user-name*. The old and new version will be identified in the database by the strings *old-version* and *new-version*. (This option is only relevant with option `--compare`.)

`--compare` or `-c`

This option instructs `navdiff` to compare two versions of NAV.

`--old` *type old-file*

Sets the type and location of the old version participating in the comparison. *Type* should be (a prefix of) either `chopped` (if the old version is chopped) or `raw` (if the old version is supplied as exported from NAV. *Old-file* is the directory of the chopped version or the name of the exported NAV file.

`--new` *type new-file*

Similar to `--old`, but for the new file.

`--size-threshold` *value*

This option sets a threshold preventing comparison of pairs of code pieces if the product of their sizes exceeds *value*.

`--compare-declarations`

By default, `navdiff` does not compare variable declarations. Include this flag to enable comparing variable declarations.

`--test-buffered-code-provider` *input-file*

For debugging purposes.

`--test-file-code-provider` *input-file*

For debugging purposes.

`--write-codepaths`

For debugging purposes.

4.3.1 Example usage

A typical use of `navdiff`

1. Chop the old exported file `w1-50sp1.export` to output directory `w1`:

   ```
   > navdiff --chop w1-50sp1.export w1
   ```

2. Initialize database `diffdb` with user `mir`:

   ```
   > navdiff --db-initialize mir diffdb
   ```

3. Compare old chopped version to a new (customized) version:

   ```
   > navdiff --db mir diffdb w1 dk-50sp1.export
   ```

Step 3 can be repeated for any number of new versions that should be compared to the same old version. All differences detected will be stored in database `diffdb`. They must be further analyzed to find common customization patterns. This task is not handled by `navdiff`.

Chapter 5

Preliminary analyzes of NAV

This chapter describes preliminary analyzes of structural properties of individual versions of NAV (in Section 5.1) and structural differences between different versions of NAV conducted using the `navdiff` tool (in Section 5.2).

As inputs, we have used W1 version 5.0 SP 1 and 39 country-specific derivatives of this version (i.e., versions customized by GDL).

5.1 Properties of individual NAV version

5.1.1 Number of code pieces in W1

Most code pieces in NAV are small. In the following, we let the size of a code piece be the number of nodes in the syntax tree used for comparison by the `navdiff` tool. This corresponds roughly to the number of statement or to the number of lines in the code piece. (The numbers presented below are generated by the analysis mode of `navdiff` tool described in Section 4.1.3.)

NAV W1 version 5.0 SP 1 contains 30 060 code pieces. Figure 5.1 on page 49 shows the number of code pieces in this version whose size are within certain ranges. (The peak in column 10-19 is a result of the presentation of the data. This column accounts for 10 different code sizes, whereas the columns to its left each account for only one code size.) In Figure 5.2 these values have been accumulated across ranges; it shows the number of code pieces whose size are less than certain limits. (Figure 5.2 shows the definite integral of Figure 5.1.) 50% of all code pieces (15 837) contain less than 6 statements and more than 80% (25 495) contain less than 20 statements.

5.1.2 Size of code pieces in W1

NAV W1 version 5.0 SP 1 contains 352 747 statements. Figure 5.3 on page 50 show the number of statements within code pieces whose size are within certain ranges. In

Figure 5.3 these values have been accumulated across ranges; it shows the number of statements within code pieces whose size are less than certain limits. 12% of all statements (43 320) are within code pieces that contain less than 6 statements, 25% (87 470) are within code pieces that contain less than 20 statements, and 80% (284 843) are within code pieces that contain less than 90 statements.

5.1.3 Number of business objects in W1

NAV W1 version 5.0 SP 1 contains 3 579 business objects in total. 897 are Tables, 1 498 are Forms, 584 are Reports, 6 are Dataports, 561 are Codeunits, 32 are XMLports, and 1 is a Menusuite. (See Appendix A.1.)

5.2 Difference between NAV versions

5.2.1 Number of code pieces added by GDLs

On the average, the 39 derivatives of W1 version 5.0 SP that we have investigated add about 1 900 code pieces. One GDL add as few as 63 code pieces whereas others add more than 5 000. Numbers are summarized in Figure 5.5 on page 51. (See Appendix A.1.)

5.2.2 Number of business objects added by GDLs

On the average, the 39 derivatives of W1 version 5.0 SP that we have investigated add about 200 business objects. One GDL add as few as 20 business objects, whereas others add more than 600. Numbers are summarized in Figure 5.6 on page 51. (See Appendix A.2.)

5.2.3 Number of modification points in GDLs

A modification point is a location (in W1) where modifications are detected. (Notice that this characterization includes modifications that are not logically a part of a customizations. Such modifications may be caused by bug fixes, performance enhancements, optimizations, code refactorings, etc.)

On the average, the 39 derivatives of W1 version 5.0 SP that we have investigated contain about 750 modification points. One GDL contain as few as 77 modification points, whereas others contain more than 1 700. Numbers are summarized in Figure 5.7 on page 53. (See Appendix A.3.)

5.2.4 Hotspots

A hotspot is a modification point that is modified by “many” derived versions.

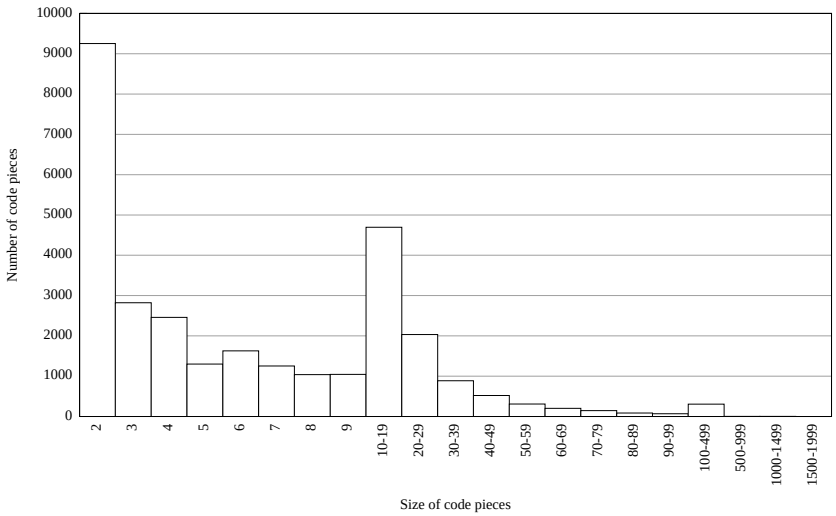


Figure 5.1: Number of code pieces of specific size ranges

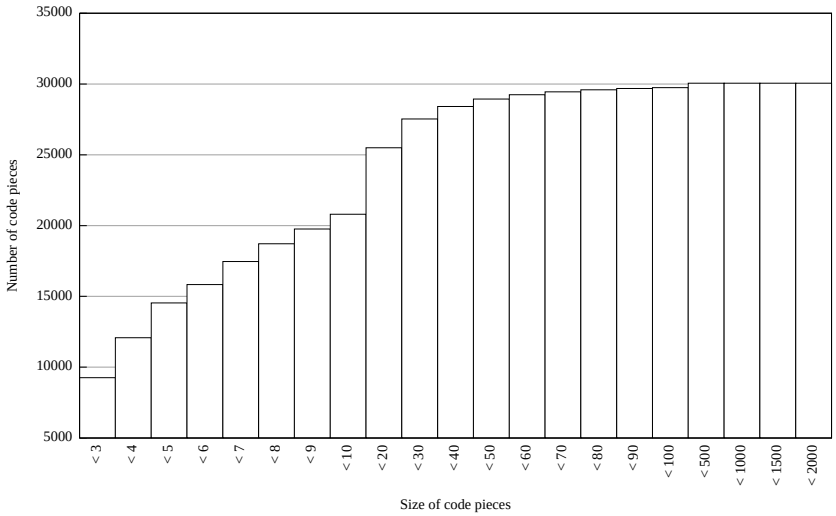


Figure 5.2: Number of code pieces less than specific sizes

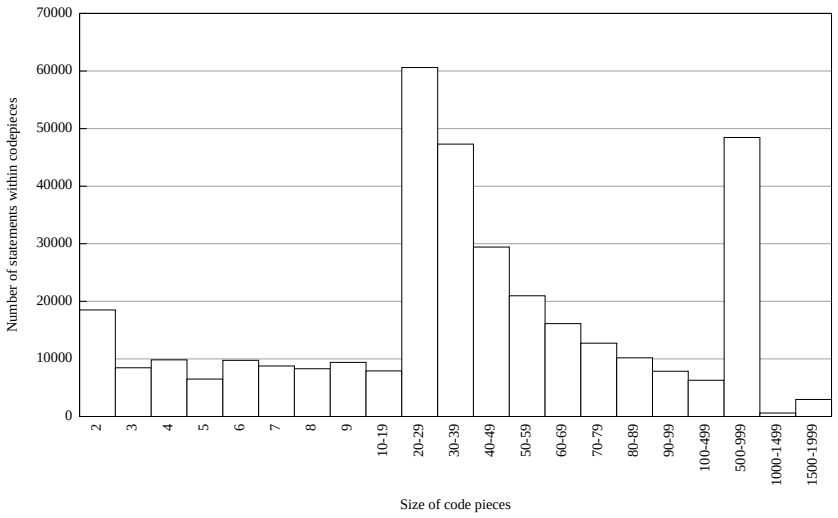


Figure 5.3: Number of statements within code pieces of specific size ranges

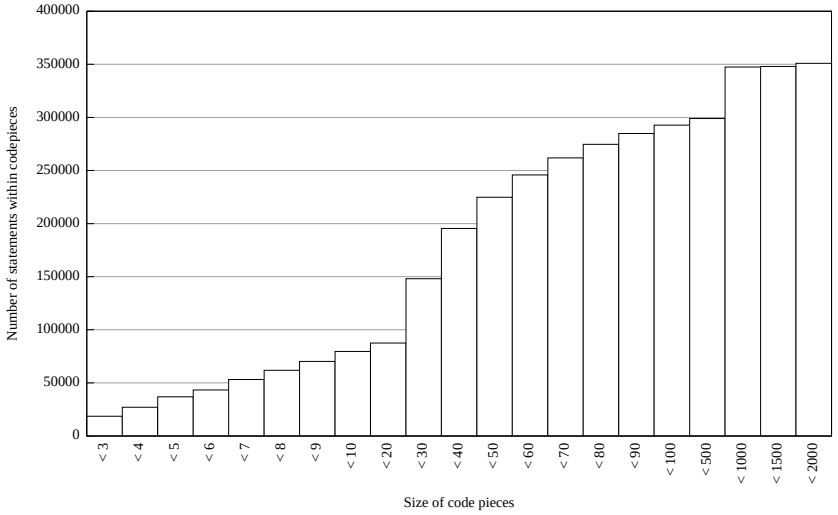


Figure 5.4: Number of statements within code pieces less than specific sizes

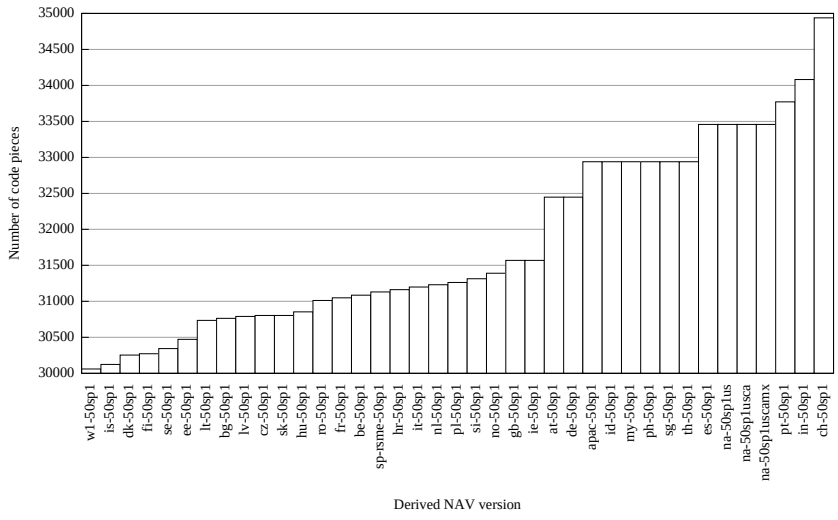


Figure 5.5: Number of code pieces pr. (derived) NAV version

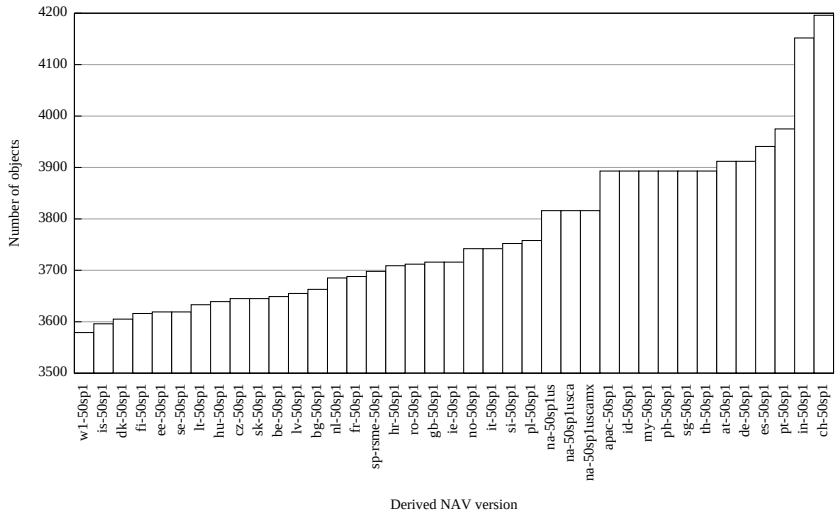


Figure 5.6: Number of business objects pr. (derived) NAV version

In total, the 39 derivatives of W1 version 5.0 SP that we have investigated modify 8 873 modification points. Most of these modification points are "cold": 45% (4 009) are modified by only one GDL, another 16% (1 468) by two GDLs, and yet another 11% (1 018) by three GDLs.

Few modification points are "hot": 31 modification points are modified by 34 GDLs and 2 are modified by 35 GDLs. Numbers are summarized in Figure 5.8 on the facing page. (See Appendix A.4.)

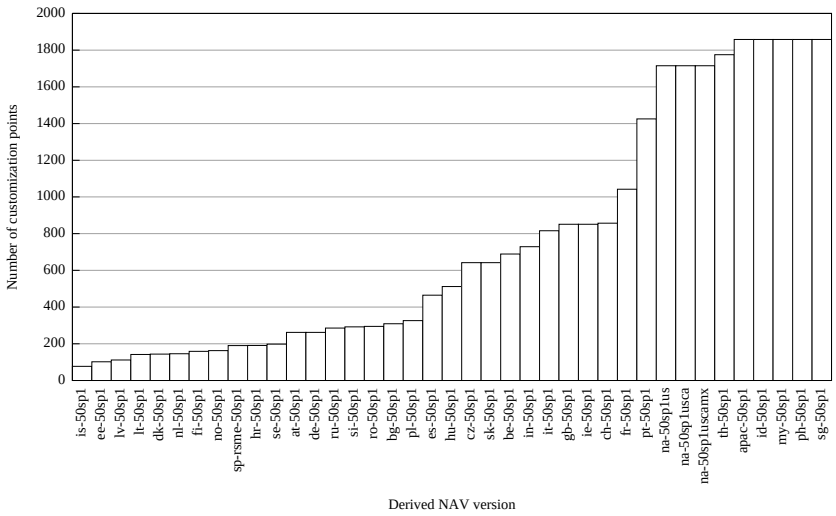


Figure 5.7: Number of modification point pr. derived NAV version

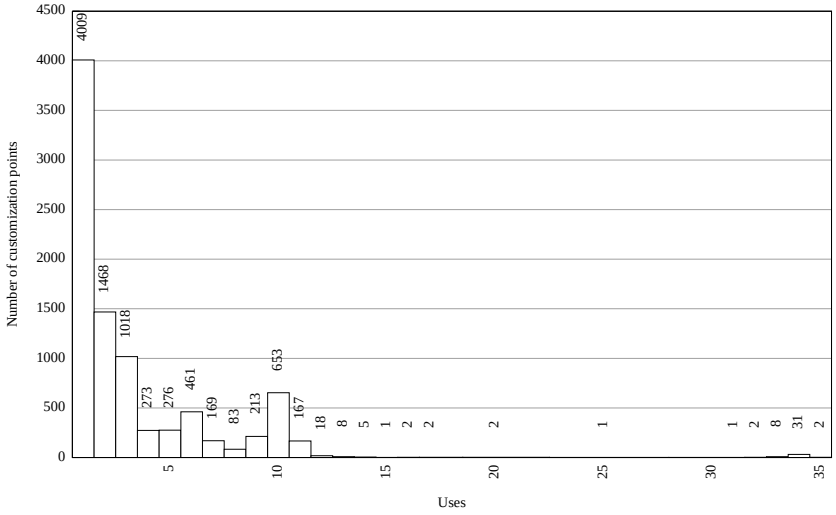


Figure 5.8: Hotspots over 39 derived versions of NAV

Chapter 6

Conclusions

We have presented a tool `navdiff` for detecting differences between different versions of the Microsoft Dynamic NAV ERP system. We have used these differences as a measure of the code modifications that independent partners apply to NAV to adapt it to the needs of specific countries, industrial segments, or enterprises.

`Navdiff` characterizes code modifications as either additions, deletions, or updates to one or more statements in a procedure. Formally, the differences between two versions of NAV are represented by one tree alignment for each procedure that exists in both versions being compared. Tree alignments recognize a particular set of modifications that we have found natural in the presence of source code.

We have used the output from `navdiff` to identify differences between the core version of NAV and 39 derived versions customized for specific countries and languages (GDLs). The purpose of this experiment was both to investigate the usefulness of the tool and to perform an actual analysis of the GDLs indicating whether the customizations applied by the GDLs follow a set of simple patterns. We conclude that tree alignments are appropriate for representing differences between programs. We also conclude that the GDL customizations of NAV do not follow simple common patterns of customizations. It is an open question whether NAV customizations follow any patterns that can be exploited to design simple and effective customization mechanisms for NAV. Standard query-language technologies seem insufficiently flexible to help address this answers. We therefore find future investigations in less traditional query languages in the domain of programs and program differences a promising research area.

Bibliography

[1] Phillip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337:217–239, 2005.

[2] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.

[3] Yvonne Dittrich and Sebastien Vaucouleur. Customization and upgrading of ERP systems: An empirical perspective. Technical report TR–2008–105, IT University of Copenhagen, Copenhagen, Denmark, March 2008.

[4] Michael Godfrey, Xinyi Dong, Cory Kapser, and Lije Zou. Four interesting ways in which history can teach us about software. In *In Proceedings of the International Workshop on Mining Software Repositories*, 2004. Available from `http://msr.uwaterloo.ca/papers/Godfrey.pdf`.

[5] James W. Hunt and M. Doug McIlroy. An algorithm for differential file comparison. Computer Science Technical report 41, Bell Laboratories, July 1976.

[6] Tom Hvitved. PhD thesis, Department of Computer Sciences, University of Copenhagen, Copenhagen, Denmark. Forthcoming.

[7] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of trees — an alternative to tree edit. *Theoretical Computer Science*, 143:137–148, 1995.

[8] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of Diff3. In V. Arvind and Sanjiva Prasad, editors, *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, number 4855 in Lecture Notes in Computer Science, pages 485–496, New Delhi, India, December 2007.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference*

on Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[10] Tetsuji Kuboyama, Kilho Shin, Tetsuhiro Miyahara, and Hiroshi Yasuda. A theoretical analysis of alignment and edit problems for trees. In Mario Coppo, Elena Lodi, and G. Michele Pinna, editors, *ICTCS*, volume 3701 of *Lecture Notes in Computer Science*, pages 323–337. Springer-Verlag, 2005.

[11] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[12] Chin Lung Lu, Zhen-Yao Su, and Chuan Yi Tang. A new measure of edit distance between labeled trees. In Jie Wang, editor, *Computing and Combinatorics, Proceedings of the 7th Annual International Conference*, number 2108 in Lecture Notes in Computer Science, pages 338–348, Guilin, China, August 2001.

[13] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[14] Eugene W. Meyers. An O(ND) difference algortihm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[15] Microsoft. Microsoft Dynamics NAV passes millionth user milestone while launching breakthrough deployment tools, November 2008. Available at `http://www.microsoft.com/presspass/press/2006/mar06/03-28Convergence2006NAVPR.mspx` on May 6, 2009.

[16] Microsoft. Recursive queries using common table expressions, August 2009. Available at `http://msdn.microsoft.com/en-us/library/ms186243.aspx` on September 18, 2009.

[17] Webb Millers and Eugene W. Meyers. A file comparison program. *Software — Practice and Experience*, 15(11):1025–1040, 1985.

[18] Thorsten Richter. A new measure of the distance between ordered trees and its applications. Technical report 85166-cs, Department of Computer Science, University of Bonn, Bonn, Germany, 1997.

[19] Anders B. Spatzek. Engagement between academia and Dynamics partners. Presented at the Academic Preconference at Microsoft Dynamics Convergence, Copenhagen, Denmark, November 2008. Available at `http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7808`.

[20] David Studebaker. *Programming Microsoft Dynamics NAV*. Packt Publishing, 2007.

[21] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26:422–433, 1979.

[22] Walter F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[23] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[24] Sebastien Voucouleur. *Upgradable Software Product Customization by Code Query*. PhD thesis, The IT University of Copenhagen, Copenhagen, Denmark, July 2009.

[25] Kaizhong Zhang. Algorithms for the constrained editing problem between ordered labeled trees and related problems. *Pattern Recognition*, 28:463–474, 1995.

Appendix A

Raw analyzes

This appendix contains the raw result from the analyzes of the structural properties of NAV and of the differences between W1 and the corresponding GDL versions.

For each set of data, a script generating the data is presented. These scripts are written for the Bash shell and make use of standard Unix tools (grep, sort, uniq, ls, wc, awk).

A.1 Number of codepieces pr. version

```
Total   Triggers        Procedures
30060   18551           11509           w1-50sp1.export
30123   18604           11519           is-50sp1.export
30253   18654           11599           dk-50sp1.export
30272   18700           11572           fi-50sp1.export
30344   18722           11622           se-50sp1.export
30471   18849           11622           ee-50sp1.export
30736   19102           11634           lt-50sp1.export
30763   19007           11756           bg-50sp1.export
30791   19187           11604           lv-50sp1.export
30804   19077           11727           cz-50sp1.export
30804   19077           11727           sk-50sp1.export
30854   19138           11716           hu-50sp1.export
31012   19277           11735           ro-50sp1.export
31048   19178           11870           fr-50sp1.export
31085   19340           11745           be-50sp1.export
31130   19344           11786           sp-rsme-50sp1.export
31162   19374           11788           hr-50sp1.export
31198   19408           11790           it-50sp1.export
31231   19198           12033           nl-50sp1.export
31262   19448           11814           pl-50sp1.export
31313   19474           11839           si-50sp1.export
31390   19553           11837           no-50sp1.export
31569   19729           11840           gb-50sp1.export
31569   19729           11840           ie-50sp1.export
32447   20191           12256           at-50sp1.export
32447   20191           12256           de-50sp1.export
32940   20609           12331           apac-50sp1.export
32940   20609           12331           id-50sp1.export
```

```
32940   20609           12331           my-50sp1.export
32940   20609           12331           ph-50sp1.export
32940   20609           12331           sg-50sp1.export
32940   20609           12331           th-50sp1.export
33457   20929           12528           es-50sp1.export
33458   21011           12447           na-50sp1us.export
33458   21011           12447           na-50sp1usca.export
33458   21011           12447           na-50sp1uscamx.export
33772   21259           12513           pt-50sp1.export
34082   21298           12784           in-50sp1.export
34938   22097           12841           ch-50sp1.export
36435   23109           13326           ru-50sp1.export
```

Generating Bash script

This script assumes that all version of NAV are stored in files whose name contains the substring "`50sp1`" and that they are formatted as the textual exports generated by NAV.

```
echo -e "Total\tTriggers\tProcedures";
for x in `ls -1 *50sp1*.export`; do
  echo -ne "`grep -c \"\(=BEGIN\|=VAR\|^[ ]*PROCEDURE\|^[ ]*LOCAL PROCEDURE\)\" $x`\t"
  echo -ne "`grep -c \"\(=BEGIN\|=VAR\)\" $x`\t\t"
  echo -ne "`grep -c \"\(^[ ]*PROCEDURE\|^[ ]*LOCAL PROCEDURE\)\" $x`\t\t"
  /bin/echo $x;
done | sort
```

A.2 Number of objects pr. version

```
Total   Table   Form    Report  Datapt. Codeun. XMLport MenuSt.
3579    897     1498    584     6       561     32      1       w1-50sp1.export
3596    901     1503    591     6       561     32      2       is-50sp1.export
3605    897     1498    591     6       579     32      2       dk-50sp1.export
3616    907     1512    593     6       564     32      2       fi-50sp1.export
3619    902     1503    611     6       563     32      2       ee-50sp1.export
3619    907     1511    592     8       567     32      2       se-50sp1.export
3633    901     1504    622     8       564     32      2       lt-50sp1.export
3639    910     1523    600     8       564     32      2       hu-50sp1.export
3645    911     1526    599     8       565     34      2       cz-50sp1.export
3645    911     1526    599     8       565     34      2       sk-50sp1.export
3649    913     1516    608     9       569     32      2       be-50sp1.export
3655    906     1513    630     6       566     32      2       lv-50sp1.export
3663    919     1530    605     7       568     32      2       bg-50sp1.export
3685    922     1535    607     13      574     32      2       nl-50sp1.export
3688    917     1536    617     12      570     33      3       fr-50sp1.export
3698    923     1542    617     14      568     32      2       sp-rsme-50sp1.export
3709    927     1548    618     14      568     32      2       hr-50sp1.export
3712    925     1541    637     7       568     32      2       ro-50sp1.export
3716    925     1543    628     6       580     32      2       gb-50sp1.export
3716    925     1543    628     6       580     32      2       ie-50sp1.export
3742    928     1550    641     13      576     32      2       no-50sp1.export
3742    947     1563    621     8       569     32      2       it-50sp1.export
3752    937     1565    621     24      569     34      2       si-50sp1.export
3758    939     1567    635     6       577     32      2       pl-50sp1.export
3816    919     1558    712     7       586     32      2       na-50sp1us.export
```

```
3816    919     1558    712     7       586     32      2       na-50sp1usca.export
3816    919     1558    712     7       586     32      2       na-50sp1uscamx.export
3893    969     1607    674     9       600     32      2       apac-50sp1.export
3893    969     1607    674     9       600     32      2       id-50sp1.export
3893    969     1607    674     9       600     32      2       my-50sp1.export
3893    969     1607    674     9       600     32      2       ph-50sp1.export
3893    969     1607    674     9       600     32      2       sg-50sp1.export
3893    969     1607    674     9       600     32      2       th-50sp1.export
3912    973     1635    654     8       607     32      3       at-50sp1.export
3912    973     1635    654     8       607     32      3       de-50sp1.export
3941    970     1647    686     7       596     33      2       es-50sp1.export
3975    981     1653    700     6       599     34      2       pt-50sp1.export
4152    1073    1750    706     6       583     32      2       in-50sp1.export
4196    1023    1698    799     9       628     36      3       ch-50sp1.export
4231    1036    1769    758     9       625     32      2       ru-50sp1.export
```

Generating Bash script

This script makes the same assumptions as the previous one.

```
echo -e "Total\tTable\tForm\tReport\tDatapt.\tCodeun.\tXMLport\tMenuSt.";
for x in `ls -1 *50sp1*.export`; do
  echo -ne "`grep -c \"^OBJECT\" $x`\t"
  echo -ne "`grep -c \"^OBJECT Table\" $x`\t"
  echo -ne "`grep -c \"^OBJECT Form\" $x`\t"
  echo -ne "`grep -c \"^OBJECT Report\" $x`\t"
  echo -ne "`grep -c \"^OBJECT Dataport\" $x`\t"
  echo -ne "`grep -c \"^OBJECT Codeunit\" $x`\t"
  echo -ne "`grep -c \"^OBJECT XMLport\" $x`\t"
  echo -ne "`grep -c \"^OBJECT MenuSuite\" $x`\t"
  echo $x;
done | sort
```

A.3 Number of modification points pr. derived version

```
Modifications
77              is-50sp1.spots
102             ee-50sp1.spots
112             lv-50sp1.spots
142             lt-50sp1.spots
144             dk-50sp1.spots
146             nl-50sp1.spots
159             fi-50sp1.spots
163             no-50sp1.spots
190             sp-rsme-50sp1.spots
191             hr-50sp1.spots
198             se-50sp1.spots
262             at-50sp1.spots
262             de-50sp1.spots
286             ru-50sp1.spots
292             si-50sp1.spots
295             ro-50sp1.spots
309             bg-50sp1.spots
326             pl-50sp1.spots
```

```
465             es-50sp1.spots
512             hu-50sp1.spots
642             cz-50sp1.spots
642             sk-50sp1.spots
689             be-50sp1.spots
729             in-50sp1.spots
816             it-50sp1.spots
851             gb-50sp1.spots
851             ie-50sp1.spots
857             ch-50sp1.spots
1042            fr-50sp1.spots
1425            pt-50sp1.spots
1715            na-50sp1us.spots
1715            na-50sp1usca.spots
1715            na-50sp1uscamx.spots
1775            th-50sp1.spots
1858            apac-50sp1.spots
1858            id-50sp1.spots
1858            my-50sp1.spots
1858            ph-50sp1.spots
1858            sg-50sp1.spots
```

Generating Bash script

The input to the following script is a set of files whose name ends in "`.spots`" listing the differences between derived versions and W1, as detected by `navdiff`. The format is described in Section 4.1.2.

Notice that additions followed by deletions with respect to the same line are counted as only one modification point in the table above.

```
echo "Modifications";
for x in `ls -1 *50sp1*.spots`; do
  awk '{print $1}' $x
  | sort | uniq | echo -e "`wc -l`\t\t$x";
done | sort -n
```

A.4 Hotspots

```
Cst.pts Count
   4009 1
   1468 2
   1018 3
    273 4
    276 5
    461 6
    169 7
     83 8
    213 9
    653 10
    167 11
     18 12
      8 13
      5 14
```

```
      1 15
      2 16
      2 17
      2 20
      1 25
      1 31
      2 32
      8 33
     31 34
      2 35
```

Generating Bash script

This script makes the same assumptions as the previous one.

```
echo "Cst.pts Count";
for x in `ls -1 *50sp1*`; do
  awk '{print $1}' $x | sort | uniq;
done
| sort | uniq -c | awk '{print $1}'
| sort | uniq -c | sort -n --key=2
```