Architectural Analysis of Microsoft Dynamics NAV

Tom Hvitved
hvitved@diku.dk
Department of Computer Science
University of Copenhagen
May 26, 2009

Abstract

This report describes our hands-on experience with the Enterprise Resource Planning (ERP) system Microsoft Dynamics NAV. Much literature exists on Microsoft Dynamics NAV, but none seem to have computer scientists as the main audience. We fill this gap by presenting the architecture of Microsoft Dynamics NAV using well-known concepts and terminology from computer science.

Our architecture analysis is object based, meaning that we present the components of NAV as classes in object oriented programming (OOP). During this analysis we address upgradability and performance problems. Our main observations are presented as hypotheses, the most important of which are (1) Lack of database joins – and in general database views – exacerbates an unnormalized database design which affects both performance and upgradability; (2) Lack of well-defined modules with well-defined interfaces affect upgradability negatively – introduction of joins is needed in order to redesign the software architecture in a modular fashion; and (3) Two key features of Microsoft Dynamics NAV, Sum Index Field Technology and reporting, have performance problems in the current implementation on Microsoft SQL Server.

We propose solutions for (1) - (3) which all take into account that Microsoft Dynamics NAV must be backwards compatible: The supply-chain of Microsoft Dynamics NAV is from Microsoft to customers via partners, who customize/modify the base product to meet the requirements of the customer. If customers are to upgrade to a new version of the ERP system, it is therefore crucial that the customizations can be ported as well – without having to implement the customizations from scratch. The proposed solutions for (3) are asymptotically faster than the current implementations.
## Contents

1 Introduction 1

2 Technical Architecture 2

2.1 Prerequisites and Conventions 3

2.2 The building blocks of NAV 3

2.3 Tables 4

  2.3.1 Interface 4

  2.3.2 Triggers 6

  2.3.3 Keys and Sum Index Field Technology 7

  2.3.4 Table relations 12

  2.3.5 Table relations formalized 17

  2.3.6 FlowFields and FlowFilters 18

  2.3.7 Iteration 22

2.4 Codeunits 24

  2.4.1 Interface 25

  2.4.2 C/AL 25

2.5 Forms 30

  2.5.1 Interface 30

2.6 Reports 31

  2.6.1 Interface 31

2.7 Runtime system (C/SIDE) 34

  2.7.1 Concurrency 34

  2.7.2 NAV 2009 35

3 NAV Design Patterns 36

3.1 Tables 36

  3.1.1 Master tables 36

  3.1.2 Templates 36

  3.1.3 Journals 37

  3.1.4 Ledgers 37

  3.1.5 References 38

  3.1.6 Registers 38

  3.1.7 Posted documents 39

  3.1.8 Setup tables 40

  3.1.9 Virtual tables 40

3.2 Codeunits 40

  3.2.1 Table independent libraries 40

  3.2.2 Table dependent libraries 41

  3.2.3 Posting routines 41

3.3 Forms 41

  3.3.1 Card forms 41

  3.3.2 Tabular/list forms 42
1 Introduction

This document summarizes the findings of an analysis of the Enterprise Resource Planning (ERP) system Microsoft Dynamics NAV (formerly Navision). NAV is an ERP system from Microsoft, targeting small- and medium-sized enterprises (SMEs). Microsoft Dynamics NAV has a substantial market share, and is distributed via Microsoft partners, who modify and customize the system to meet the requirements of the individual customers. The following numbers from the Microsoft Dynamics NAV web page [12] illustrate the success of NAV (and hence the relevance of this report):

- More than 57,000 customers (SMEs) worldwide.
- More than 2,700 certified partners worldwide.
- More than 1,500 registered add-ons (verticals).

One of the reasons for the success of Microsoft Dynamics NAV is high flexibility. Almost all functionality of NAV is written in a domain specific language (C/AL) which is customizable and extensible by NAV partners. C/AL is a relatively small language, making it easy for non-computer scientists to learn (typically NAV developers do not have a background in computer science).

The goal of this report is to document the present NAV architecture from a birds point of view, and come up with a redesign of the architecture which permits modularization, similar to what is known from Service Oriented Architecture (SOA) [3]. We aim at presenting the NAV architecture as concisely as possible; there exists much literature on NAV (see the list references for some), but none of them seem to have computer scientists as the main audience. We hope to fill this gap, by using well-known concepts (and terminology) from computer science such as semantics, invariants, relational databases, object oriented programming etc. Note however that due to the lack of formal documentation of NAV, the formalizations in this document describe our intuition, which may not be coherent with the reality (ultimately, the implementation/code of NAV is the formalization, which is however not very abstract).

Unfortunately we did not succeed with the second goal of reengineering (parts of) the NAV architecture, and we claim that this goal requires introduction of new functionality in Microsoft Dynamics NAV.

The rest of this document is structured as follows: First we give an overall introduction to NAV, which includes a thorough technical overview and a description of typical NAV design patterns. Then we proceed to a description of our attempt at modularizing NAV functionality, and why our approach did not work. We propose another method for modularizing NAV, and conclude with a list of topics for future work (some of which have already
 Throughout the document we present *hypotheses*, which are our key observations, and for some of the main problems encountered, we propose solutions.

2 Technical Architecture

In this section we present the technical architecture of NAV using concepts and terminology from computer science. In order to avoid a conflict of terminology, we will try to consistently differentiate NAV terminology (by underlining any first occurrences) and “normal computer science” terminology. Appendix A summarizes the key terms used in NAV and their correspondent in computer science terminology. The presentation in this section will focus on what the different key parts of NAV are, rather than how they are used. Figure 1 shows this distinction: We focus on the “horizontal” parts rather than the “vertical” parts (the vertical parts are described in course notes [14]).

![Figure 1](image.png)

Throughout our analysis we have been working with Microsoft Dynamics NAV W1 5.0 SP1, using Microsoft SQL Server 2005 as the underlying database system. NAV uses a classical two-tier architecture with a database system and a thick client running all business logic. We have deliberately chosen only to work with the SQL database system, as opposed to using the native NAV database system, since future releases of NAV will only support Microsoft SQL Server. For most parts of the analysis this choice makes no difference, we will however include the native database briefly when discussing Sum Index Field Technology in Section 2.3.3.

---

1Section 2.7.2 describes why the results of our analysis carry over to the recent version of NAV, Microsoft Dynamics NAV 2009.
2.1 Prerequisites and Conventions

In order to get full outcome from this section, the reader should be familiar with the following areas in computer science:

- Relational databases and SQL [2].
- Object oriented programming (OOP) [1].
- Denotational semantics [21].
- Algorithms and data structures [4].

Throughout the report we will use the following typographical conventions:

- NAV objects (tables, codeunits, forms and reports) use the following font: NAV object.
- Column names (fields) use the following font: Field.
- Code (C/AL) use the following font: Code.
- Keyboard strokes use the following font: Key.
- The first occurrence of an important NAV term is underlined.

2.2 The building blocks of NAV

As mentioned in the introduction, all business logic (C/AL code – more on C/AL in Section 2.4) runs entirely on the client. This however does not mean that the client is an “out of the box” atomic entity. Instead the client is a runtime system for executing various “building blocks” (called NAV objects – not to be mistaken with objects in OOP) containing business logic.

This means that the client itself does not implement the various components of the ERP system; rather the components are made up from NAV objects, which are then executed by the client. Provided that a proper license is used, it is possible to customize the components by altering the NAV objects they are made up of. This is a key feature of NAV: Almost all functionality is located in NAV objects that are modifiable, making it a highly customizable ERP system. The downside to this flexibility is that developers may alter C/AL code supplied by Microsoft, resulting in the need for a code merge when Microsoft releases a new version of the base code (this is known as the “update problem” in ERP systems [5]).

NAV objects can be divided into the following types:

- Tables
• Codeunits
• Forms
• Reports
• (Dataports), (XMLports) and (MenuSuites)

The following sections describe the first four object types, as these are the core object types of NAV. Our description will consist of an “outside in” and an “inside out” approach: The former is an object based description, where we conceptualize the interface each NAV object provides. In this setting each NAV object corresponds to a class in OOP, which can be instantiated in other classes (i.e. NAV objects in NAV terminology). The latter will describe how each entry in the class interface is presented in the internal representation/implementation (via both screen dumps and code fragments).

It should be mentioned that our analysis does not take into account the historical evolution of NAV: Many design choices are without a doubt due to the fact that NAV needs to be backwards compatible, as the customizations done by NAV partners need to be portable to new versions. We hence present the architecture as is.

Data- and XMLports deal with exporting data from NAV to clear text and XML respectively, and MenuSuites deal with the GUI only (no business logic), so these object types will not be discussed in this report. For a more thorough description of Dataports, XMLports and MenuSuites see [18].

2.3 Tables

2.3.1 Interface

Microsoft Dynamics NAV uses table objects to store data persistently. An NAV table corresponds to a class in OOP, with the methods and properties presented in Figure 2.

The methods and properties below the keyword Constant mean that the entries are shared by all instances of the table object and cannot be changed at runtime. In contrary, entries below Per instance can have different values per instance (e.g. a method may refer to instance variables).

Each NAV table has a name (1) and a signature (2). The signature describes the fields (columns in SQL terminology) and the type of each field (SimpleType is defined later, so for now just think “simple types”, such as strings, integers, etc.). In the implementation/internal representation each NAV table consists of exactly one table in the SQL database (with the exception of tables containing FlowFields and FlowFilters, more on these in Section 2.3.6).
Figure 2 The semantic model of a NAV table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Name</strong> ∈ String</td>
<td>(table name)</td>
</tr>
<tr>
<td>2. <strong>Σ</strong> : String → fin SimpleType × ( \mathcal{P}(\text{Property}) )</td>
<td>(signature/table schema)</td>
</tr>
<tr>
<td>3. <strong>Fields</strong> ( \overset{\text{def}}{=} \text{dom}\Sigma )</td>
<td>(non-empty primary key definition)</td>
</tr>
<tr>
<td>4. <strong>PrimaryKey</strong> ∈ <strong>Fields</strong> +</td>
<td>(table indexes and Sum Index Field definitions)</td>
</tr>
<tr>
<td>5. <strong>Indexes</strong> : <strong>Fields</strong> + ( \rightarrow \mathcal{P}(\text{Fields}) )</td>
<td>(table relations)</td>
</tr>
<tr>
<td>6. <strong>TableRelation</strong> : <strong>Fields</strong> ( \rightarrow \text{TableRelationExp} )</td>
<td>(FlowField definitions)</td>
</tr>
<tr>
<td>7. ( \Sigma_{\text{FlowField}} : \text{String} \rightarrow_{\text{fin}} \text{FlowFieldExp} )</td>
<td>(non-overlapping definitions)</td>
</tr>
<tr>
<td>8. ( \Sigma_{\text{FlowFilter}} : \text{String} \rightarrow_{\text{fin}} \text{SimpleType} )</td>
<td>(built-in methods)</td>
</tr>
<tr>
<td>9. ( \text{dom}\Sigma \cap \text{dom}\Sigma_{\text{FlowField}} \cap \text{dom}\Sigma_{\text{FlowFilter}} = \emptyset )</td>
<td>(OnInsert, OnDelete, etc.)</td>
</tr>
<tr>
<td>10. <strong>Vars</strong> : String ( \rightarrow_{\text{fin}} \text{Type} )</td>
<td>(user-defined instance variables)</td>
</tr>
<tr>
<td>11. <strong>Methods</strong> : String ( \rightarrow_{\text{fin}} \text{Procedure} )</td>
<td>(user-defined methods)</td>
</tr>
<tr>
<td>12. <strong>Mutators</strong></td>
<td>(built-in methods for updating state, e.g. set a FlowFilter)</td>
</tr>
<tr>
<td>13. <strong>Iterator</strong></td>
<td>(an iterator for traversing data in the table. Key methods: FIND, INSERT, MODIFY, DELETE)</td>
</tr>
</tbody>
</table>

Rows in a table are called [records](#) in NAV terminology. The term record is however overloaded, and sometimes (in particular in C/AL code) it refers to an entire table object, rather than a tuple of values in the table. We assume that this is the case since a table object is essentially an iterator (more on this in Section 2.3.7), and hence it always points to a particular column in the underlying SQL table.

Each field in a table may have associated properties, which differ depending on the type of the field: An example is the property **NotBlank**, which – when set – will not allow a record with a null-value in the field (i.e. the field “cannot be blank”). The set **Property** consists of all possible properties (cf. the co-domain of **Σ**).

Figure 3 shows the table **Customer**, which holds the information of customers in NAV (only the first 11 fields are included). The figure also shows some of the properties of the first field **No.**. Letting **Σ** denote the signa-
ture of the Customer table, we then have that e.g. \( \pi_1(\Sigma(No.)) = \text{Code}[20] \), \( \text{Editable} \in \pi_2(\Sigma(No.)) \) and \( \not\text{NotBlank} \not\in \pi_2(\Sigma(No.)) \).

What we have seen so far is very similar to SQL. NAV tables have, however, some features which cannot be modeled directly in SQL. We give here a description of some of those features, namely triggers, keys, table relations, FlowFields/FlowFilters and the use of an iterator.

### 2.3.2 Triggers

Each NAV table has a set of predefined triggers (Figure 2 (10)), which may contain C/AL code. Triggers in NAV correspond to normal methods in OOP, and not triggers as known from active databases [2]. The table below summarizes all table triggers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Scope</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnInsert</td>
<td>Table</td>
<td>Insertion of a new record</td>
</tr>
<tr>
<td>OnModify</td>
<td>Table</td>
<td>Modification of an existing record</td>
</tr>
<tr>
<td>OnDelete</td>
<td>Table</td>
<td>Deletion of a record</td>
</tr>
<tr>
<td>OnRename</td>
<td>Table</td>
<td>Modification of a field in an existing record, which is part of the tables primary key (Section 2.3.3)</td>
</tr>
<tr>
<td>OnValidate</td>
<td>Field</td>
<td>Data is entered into the field, or validation is requested</td>
</tr>
<tr>
<td>OnLookup</td>
<td>Field</td>
<td>The user does a lookup (F6) on the field</td>
</tr>
</tbody>
</table>

The scope “Table” means that there is one trigger per table, and the scope “Field” means one trigger per field.

To the authors’ knowledge, table triggers are only executed automatically when the table is accessed directly from the user interface (i.e. from forms, cf. Section 2.5). Thus if a record is inserted programatically (via C/AL code), then the OnInsert trigger will not be automatically fired! (it is possible to manually trigger the OnInsert trigger, however). The same goes for field triggers, where it is possible to execute the OnValidate code manually from the code, but it is not done automatically on insertion/modification.

Thus NAV table triggers are in fact GUI triggers, which makes it dangerous to use them for invariants/validation, as they are not always executed. We believe a more consistent use of triggers should be applied.

**Hypothesis 1**: The use of NAV table triggers for business logic (validation) can lead to inconsistent data, as NAV table triggers are not always executed. Solution: Encapsulate tables so they are only accessible via one interface – build database-style triggers (which cannot be escaped) into this encapsulation.
Besides from the built-in triggers, NAV tables can contain user-definable methods (12) and variables (11). User-definable variables can be used to instantiate other NAV objects (and the table object itself), but we will return to this in Section 2.4, as this is exactly the functionality of codeunits.

2.3.3 Keys and Sum Index Field Technology

NAV keys correspond to keys/indexes in database terminology. NAV keys play almost the same role as in SQL: performance enhancements and uniqueness constraints. Each table in NAV must have a primary key (Figure 2 (4)), which is translated directly into a primary key in the underlying SQL table. This automatically implies that no two records can be equal, as they must differ on at least one of the fields in the primary key. Figure 4 shows the NAV keys defined on the table G/L Entry.

**Figure 4** NAV Keys on the NAV table G/L Entry.

The first key (Entry no.) is the primary key. In Figure 2 this corresponds to PrimaryKey = (Entry no.). The other keys are secondary keys, which correspond to indexes in database terminology. The primary key is implicitly appended to all indexes. This means that for instance the second NAV key in Figure 4 is really an index on

(G/L Account No., Posting Date, Entry No.)

The column SumIndexFields in Figure 4 is where NAV keys differ from SQL keys/indexes: In order to calculate FlowFields fast (Section 2.3.6), NAV uses Sum Index Field Technology (SIFT). In Figure 4 Amount is defined as a Sum Index Field for the index (G/L Account No., Posting Date), which in our semantic model (Figure 2 (5)) means that

Amount ∈ Indexes(G/L Account No., Posting Date)

(Note that the order of the fields in the argument to Indexes matters). So what does it mean when in general F ∈ Indexes(F₁,...,Fₙ) on a table
First of all it means that the DBMS maintains an ordinary index on \((F_1, \ldots, F_n, F_{n+1}, \ldots, F_{n+m})\), where \(\text{PrimaryKey} = (F_{n+1}, \ldots, F_{n+m})\) (remember that the primary key is always appended to indexes). As known from database theory, such an index enables support for \((optimal)\) range queries of the form

\[
\sigma_{F_1=v_1 \land \ldots \land F_{i-1}=v_{i-1} \land F_i \in [v_i; v'_i]}(T)
\]

where \(i \leq n + m\), each \(v_i/v'_i\) is a value of type \(\pi_1(\Sigma(F_i))\) and we have used notation from relational algebra \([2]\). The condition \(F_i \in [v_i; v'_i]\) is equivalent to \(v_i \leq F_i \leq v'_i\), which implies that each type \(\pi_1(\Sigma(F_i))\) must be ordered; and indeed this is the case for all fields allowed in a NAV key – thus one cannot define an index using a field \(F\) if e.g. \(\pi_1(\Sigma(F)) = \text{BLOB}\).

But having such an index also enables support for computation of aggregated sums\(^2\)((\optimal) range sum queries) of the form

\[
\sum_{r \in \sigma_{F_1=v_1 \land \ldots \land F_{i-1}=v_{i-1} \land F_i \in [v_i; v'_i]}(T)} \pi_F(r)
\]

where again \(i \leq n + m\) and each \(v_i/v'_i\) is a value of type \(\pi_1(\Sigma(F_i))\).

We demonstrate, by example, how NAV indexes can be implemented using balanced search trees (e.g. B+ trees) – and in fact we believe this is in principle how they are implemented in the native NAV database. Consider the table \(G/L\) Entry from Figure 4 and assume that it contains the following records (throughout this example we omit the implicit appending of the primary key, as it makes no difference):

| G/L Account No. | Posting Date   | Amount | ...
|-----------------|----------------|--------|-----|
| \(r_1\)         | 1010           | 2008-05-01 | 100 | ...
| \(r_2\)         | 1030           | 2008-03-01 | 300 | ...
| \(r_3\)         | 1020           | 2008-07-01 | 600 | ...
| \(r_4\)         | 1010           | 2008-01-01 | 200 | ...
| \(r_5\)         | 1020           | 2008-12-01 | 100 | ...
| \(r_6\)         | 1030           | 2008-01-01 | 200 | ...
| \(r_7\)         | 1020           | 2008-08-01 | 500 | ...
| \(r_8\)         | 1020           | 2008-10-01 | 200 | ...

Then as in an ordinary database index, the DBMS maintains a balanced search tree (where the sorting is lexicographically on \(G/L\) Account No. and Posting Date), which may look as follows (here we use the invariant that each node in the left sub tree has a key which is \textit{strictly less}, and each node in the right sub tree has a key which is \textit{greater than or equal}):  

\(^2\)Actually SIFT indexes can also be used to compute other aggregated functions, but for now we only consider sums. Section 2.3.6 includes a full list of SIFT functions.
It is then a well-known result from algorithmics that we can search, insert and delete (and hence update, all preserving balancedness) in $O(\log n)$, where $n$ is the number of records in the table (cf. Chapter 13 in [4] on red-black trees, which are similar to B+ trees). This means that the range queries mentioned earlier can be performed in logarithmic time (i.e. logarithmic time – in the size of the table – for indexing plus linear time – in the size of output – for actually retrieving the records).

Now when Amount is defined as a Sum Index Field for the index above, we instrument each internal node in the search tree with the accumulated sum of all records beneath the node. The index tree above becomes:

It is shown in [4], Theorem 14.1, that maintaining augmented data in a balanced search tree which only depends on the value of the augmented data in the child nodes, introduces no asymptotic overhead on insertion, deletion and update. Since this is the case for aggregated sum (the sum in a node is exactly the sum of the aggregated sums in the child nodes), we get that the aggregated sums can be maintained in $O(\log n)$ as well.

Now if we want for instance to compute the sum of all postings on account 1020 from 2008-06-01 to 2008-11-01, then we need to perform two searches in the index tree (one for the lower bound and one for the upper). The informal algorithm is as follows:

---

And in fact this is the case for all aggregated SIFT functions in NAV, which is why we believe the original implementation used augmented B+ trees.
1. Do a binary search on the tree until we find the first node with key \( k \), such that \( k \in [(1020,2008-06-01);(1020,2008-11-01)] \) (this is the root node in the example).

2. Do a binary search on the left sub tree wrt. the lower bound, i.e. \((1020,2008-06-01)\).
   - Each time we go left, record the sum in the right sub tree (this happens once, where the value 1100 is recorded).
   - Each time we go right, record nothing (this happens once).
   - When a leaf is reached, record the value, if \( k \geq (1020,2008-06-01) \) (this does not happen since the leaf is \( r_1 \)).

3. Do a binary search on the right sub tree wrt. the upper bound, i.e. \((1020,2008-11-01)\).
   - Each time we go left, record nothing (this happens twice).
   - Each time we go right, record the sum in the left sub tree (this does not happen).
   - When a leaf is reached, record the value, if \( k \leq (1020,2008-11-01) \) (this happens since the leaf is \( r_8 \)).

4. Return the sum of all recorded values \( 1100 + 200 = 1300 \).

Since the index tree is always balanced, we are guaranteed that the range sum queries will be in \( O(\log n) \), where \( n \) is the number of records in the table.

**Hypothesis 2**: The native NAV database uses B+ trees \[2\] for indexes, where each node is augmented with accumulated data. This implies insertion/update and range (sum) queries complexity that is logarithmic in the number of records in the table.

The benefit of being able to compute total sums in logarithmic time comes at the cost of having to maintain intermediate results when updating tables with Sum Index Fields. A thing to note here is that it is up to the programmer to decide which Sum Index Fields to maintain (i.e. the programmer must specify the map \( Indexes \)), which may result in an unnecessary amount of indexes being maintained; if for instance a report computes very slow, it is easy to add indexes to make the computation go faster, however it is not \emph{a priori} evident when an index is maintained, but never utilized. This is analogous to deciding when to add an index to a SQL database.

**Hypothesis 3**: SIFT indexes in NAV are potential performance bottlenecks: When a new record is inserted, intermediate results need to be updated. Possible solution: Automatic generation of SIFT indexes – either from traces in the database or from static code analysis – ensuring that only those indexes which are needed are actually maintained.
As mentioned in Hypothesis 2 we believe the native database uses B+ trees with logarithmic complexity to maintain SIFT indexes. The implementation in Microsoft SQL Server uses materialized views [2] (also called indexed views): If \( F \in \text{Indexes}(F_1, \ldots, F_n) \) is defined on a table \( T \), a materialized view is created with the following (simplified) query:

\[
\begin{align*}
\text{SELECT} & \quad \text{COUNT(*)}, \quad \text{SUM(F)} \\
\text{FROM} & \quad T \\
\text{GROUP BY} & \quad F_1, \ldots, F_n
\end{align*}
\]

and an ordinary index is created on \((F_1, \ldots, F_n)\). Having the index means that the view can be updated in worst-case \( O(\log n) \) on insertion, deletion and modification in \( T \), where \( n \) is the number of rows in \( T \). The reason why this is possible is that \( \text{SUM} \) and \( \text{COUNT} \) are reversible, i.e. when a row is deleted we can subtract its value in \( F \) from the sum, and decrement the counter (as we shall see soon, this is not the case for all SIFT functions).

Thus maintaining the augmented sums can – as before – be done in \( O(\log n) \), assuming that Microsoft SQL Server utilizes the reversibility of \( \text{SUM} \) and \( \text{COUNT} \) to do so. However, range sum queries are now no longer in \( O(\log n) \), but worst-case \( O(n) \): If \( T \) contains \( n \) rows that are all distinct on \((F_1, \ldots, F_n)\), then the materialized view will contain \( n \) rows as well. So if we want the sum of all rows, then we need to add up \( n \) values!

**Hypothesis 4:** The implementation of Sum Index Field indexes in Microsoft SQL Server is non-optimal, as the accumulated sums are not directly attached to the B+ index trees. Maintenance of intermediate sums and counts can be done optimally (provided that Microsoft SQL Server utilizes the reversibility of \( \text{SUM} \) and \( \text{COUNT} \)), but calculation of range sum queries has worst-case linear complexity.

So a better solution for Sum Index Field Technology is to augment the B+ index trees. But even though Microsoft SQL Server is being used, it may not be possible to get access to the internal structure of the database indexes, and to augment these. An alternative solution is to emulate the augmented B+ tree in a table, where each row represents a node, with pointers to its child nodes. Adding an ordinary index on the pointer identifiers will then make it possible to look up nodes in \( O(\log n) \), providing an emulated index tree with complexity \( O(\log^2 n) \). So for updates this solution is sub optimal with respect to materialized views, but still asymptotically faster for calculating range sum queries. The question is, however, if the constant overhead will be too high for this solution to be useful.

In Section 2.3.6 we will see that it is also possible to compute minimum (and maximum) over a field in a table. As is the case for sum and count, minimum can be maintained in an augmented B+ tree in \( O(\log n) \). It is however not possible to maintain minimum in a materialized view in \( O(\log n) \), as minimum is not reversible! If it is known that \( \min(\pi_F(\sigma_{F_1=v_1} \land \ldots \land F_n=v_n(T))) = m \)
and a row with $F_i = v_i, i = 1, \ldots, n$ is deleted, then the new minimum cannot be computed without inspecting all rows with $F_i = v_i, i = 1, \ldots, n$.

We suspect this to be the reason why the materialized view is not equipped with minimum (and maximum) as well as sum and count. So if we want to compute minimum over a field, then SIFT will no longer help, and one would expect instead the obvious linear time algorithm:

$$\text{SELECT MIN}(F)\text{ FROM } T\text{ WHERE } F_1 = v_1, \ldots, F_i = v_i$$

for computing the “range minimum query” $\min_{r \in \sigma_{F_1 = v_1 \land \ldots \land F_i = v_i}(T)} \pi_F(r)$. But in fact the implementation is even worse: To find the minimum over $F$, the following algorithm is applied (as witnessed by inspecting a database trace):

1. Let $S = \sigma_{F_1 = v_1 \land \ldots \land F_i = v_i}(T)$.
2. Let $m = \pi_F(r)$, where $r$ is the first row in $S$.
3. Let $S = \sigma_{F < m \land F_1 = v_1 \land \ldots \land F_i = v_i}(T)$. If $S = \emptyset$ return $m$ else go to 2.

Thus in worst-case (when the rows in $T$ are sorted descending) the complexity is $O(n^2)$! To test this we constructed a table with one million rows, sorted descending. Using the obvious SQL statement from above, the query took under a second, while performing the query from NAV took 25 minutes, resulting in the order of one million SQL statements being executed! To make matters even worse, the selection of $S$ in the algorithm above uses a (seemingly random) sorting, which implies an actual complexity of $O(n^2 \log n)$ (provided that the fields being sorted on are not included in an index).

**Hypothesis 5**: Minimum and maximum have been removed from SIFT indexes in Microsoft Dynamics NAV 5.0 SP1, as they are not maintainable in a materialized view in $O(\log n)$. The replaced algorithm for minimum and maximum has complexity $O(n^2)$.

### 2.3.4 Table relations

NAV table relations are much like foreign key constraints in SQL. There are however some differences, which makes it impossible to model the NAV constraints directly in SQL. Table relations are defined on fields (cf. the domain of the function $\text{TableRelation}$ from Figure 2 (6)). The full grammar for table relation expressions is presented in Figure 3 ([: ] means optionality).

The most common (and simplest) type of relation has the form $\text{TargetTable}.\text{TargetField}$
where TargetTable is a table and TargetField is a column in TargetTable. When this relation is set on a field SourceField, i.e.

$$TableRelation(SourceField) = TargetTable.TargetField$$

it means that all records in SourceTable must have a value in the field SourceField which matches a record in TargetTable with the same value in the field TargetField (unless the value in the source field is null). More formally expressed, the table relation states that the following must hold:

$$\forall s \in SourceTable. \pi_{SourceField}(s) \neq \text{null} \Rightarrow \exists t \in TargetTable. \pi_{SourceField}(s) = \pi_{TargetField}(t)$$

This simple kind of relation is exactly what is known from SQL as a foreign key. There is however one important difference: Whereas foreign key constraints in SQL are invariants (i.e. the relations are guaranteed to hold at all times outside transactions), table relations in NAV are only checked when records are inserted or modified! Thus if $TableRelation(F) = rel$, then the predicate specified by rel (will be formalized soon) will only be checked when a new value is inserted in the field F of a (possibly new) record.

This means that it is possible for a record in SourceTable to refer to another record in TargetTable, which is later modified or deleted, resulting in a dangling reference. So how can a programmer make sure that a record does not refer data, which may potentially be deleted? The simplest solution is to copy the data from the target table; this guarantees local consistency, but it also results in an unnormalized database design, which is harder to maintain during upgrades.

**Hypothesis 6:** Table relations in NAV are not invariants, contrary to e.g. foreign key relations in SQL. This can result in inconsistent data, if
data is referred rather than copied. To ensure consistency the developer may therefore resolve to copy data, resulting in unnormalized database design.

Often the field identifier TargetField is left out in the simple relation, in which case the first field of TargetTable’s primary key is used. Figure 6 shows the table Customer which has a table relation to the table Currency set on the field Currency Code (i.e. TableRelation(Currency Code) = Currency). Since the primary key of Currency is (Code), the table relation is equivalent to TableRelation(Currency Code) = Currency.Code. In effect this means that if Currency Code is set on a customer, then a currency must be defined with the corresponding code (as one would expect).

Figure 6 A simple table relation for the field Currency Code in the NAV table Customer.

As mentioned above, simple table relations in NAV correspond to SQL foreign keys (modulo invariance), but NAV table relations are more general – in particular NAV table relations can be conditional with respect to the possible target records (via the WHERE clause) as well as the possible target table (via the IF construction).

We describe the three kinds of conditions in a WHERE clause by examples (in the next section we give a formal definition of NAV table relations). In all examples we consider the relation Currency.Code from Figure 6 (we include the field identifier Code to make the relation explicit):

First consider if we change the relation to

\[ \text{Currency.Code WHERE EMU Currency} = \text{CONST(Yes)} \]

Then the related record must have EMU Currency set to Yes, meaning that the currency must be the EMU currency. Second consider if we change the relation to

\[ \text{Currency.Code WHERE Last Date Modified} = \text{FILTER(01-01-09..01-02-09)} \]

Then the related record must have been last modified between 1st of January 2009 and 1st of February 2009. Finally consider if we change the relation to
Currency.Code WHERE Description = FIELD(Name)

Then the related record must have the same value set in the field Description as the value in the field Name of the Customer record (this makes really no sense, and is just meant as an example).

Thus in all three cases the left hand side of the equality in a WHERE clause is a field of the related table (the target table), and the right hand side is either a constant, a filter or a field in the source record. Note that the conditions of a WHERE clause in a NAV table relation can be expressed as a WHERE clause in a SQL statement, but SQL does not support foreign keys with WHERE clauses. Without having investigated it further, we believe one possibility of achieving integrity constraints with WHERE clauses directly in SQL could be to use assertions. It should however be investigated if this is possible in SQL in general, and in particular whether it is possible in Microsoft SQL Server.

So WHERE clauses can be used to put constraints on the records in the target table. But NAV table relations also have the ability to relate to different tables/fields, depending on conditions set on the source table. As can be seen from the grammar, the conditions only include equality with constants and filters – we give an example using both:

Consider again the table relation Currency.Code on the field Currency Code in table Customer. If we change the relation to

```
IF Name = CONST(Bob)
  Currency.Code
ELSE IF Last Date Modified = FILTER(01-01-09..01-02-09)
  Language.Code
```

Then the table relation can have three possible outcomes:

1. If the name of the customer is “Bob” then the relation on the field Currency Code is to the field Code of the table Currency (as before).
2. Otherwise, if the customer record was last modified in the period 1st of January 2009 to 1st of February 2009, the relation is to the field Code of the table Language.
3. Otherwise there are no table relations.

Conditional table relations is another example of relations that cannot be modeled in SQL using foreign keys. As a consequence, all consistency checks are performed directly in the client, rather than relying on Microsoft SQL Server.

One important thing to note in connection with table relations is that it is not possible to join tables in C/AL. One might expect such a facility since tables have relations, and NAV runs directly on Microsoft SQL Server, but this is not the case as NAV did not originally run on a relational database.
As one can imagine, the table relations of NAV can be quite powerful (compared to foreign key constraints in SQL). We claim however that most uses of conditional table relations could be avoided, if C/AL permitted joins: Consider the following (simplified) table relation on the field Bal. Account No. in the table G/L Entry:

\[
\begin{align*}
&\text{IF (Bal. Account Type} = \text{CONST(G/L Account))} \\
&\quad \text{G/L Account} \\
&\quad \text{ELSE IF (Bal. Account Type} = \text{CONST(Customer))} \\
&\quad \text{Customer} \\
&\quad \text{ELSE IF} \\
&\text{\ldots}
\end{align*}
\]

This relation states that if a G/L entry is balanced against a G/L (customer) account, then the account number in the field should be a G/L (customer) account. Figure 7 shows an example where the G/L Entry table contains two records: The first entry is balanced against a G/L account and the last entry is balanced against a customer account (the relations for each record are shown in parentheses).

**Figure 7** NAV table G/L Entry with two records.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G/L Account</td>
<td>1010 (→ G/L Account)</td>
</tr>
<tr>
<td>2</td>
<td>Customer</td>
<td>1020 (→ Customer)</td>
</tr>
</tbody>
</table>

It is possible to achieve the same effect using only static table relations, if we instead use mapping tables, i.e. one table for each branch in the conditional table relation (Figure 8).

**Figure 8** G/L account mapping table (top), customer account mapping table (bottom).

<table>
<thead>
<tr>
<th>G/L Entry no.</th>
<th>G/L Account no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (→ G/L Entry)</td>
<td>1010 (→ G/L Account)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G/L Entry no.</th>
<th>Customer Account no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 (→ G/L Entry)</td>
<td>1020 (→ Customer)</td>
</tr>
</tbody>
</table>

In this way conditional table relations can be replaced by simple table relations that are checkable directly in Microsoft SQL Server, rather than having the client to do the checks. Furthermore, we believe the complex table relations can be an indication of unnormalized database design: It is likely that the target of a (complex) table relation determines which other
fields of the source table should be active. It would hence be better to place these fields in the mapping table (as described above), to eliminate null-values in the source table.

**Hypothesis 7:** Table relations in NAV are unnecessarily complex: The use of conditionals can be a sign of unnormalized database design. Solution: Only allow foreign key relations (as invariants) and introduce joins (or in general views).

### 2.3.5 Table relations formalized

In this section we briefly present the (denotational) semantics of table relations, which may be easier to understand to some readers. Assume

\[
\text{TableRelation}(\text{SourceField}) = rel
\]

in the table SourceTable. Then each time a record \(s\) is either created or modified in SourceTable, the NAV client checks that the following holds:

\[
\pi_{\text{SourceField}}(s) \neq \text{null} \Rightarrow R[rel](\text{SourceField}, s)
\]

where \(R[rel](\text{SourceField}, s)\) is defined by structural induction on \(rel\):

\[
R[\text{TargetTable}.\text{TargetField}\ WHERE\ cs](\text{SourceField}, s) = \exists t \in \text{TargetTable}. \pi_{\text{TargetField}}(t) = \pi_{\text{SourceField}}(s) \land \bigwedge_{c \in cs} W[c](s, t)
\]

\[
R[\text{IF}\ cs\ rel\ ELSE\ rel'](\text{SourceField}, s)
= \begin{cases} R[rel](\text{SourceField}, s), & \text{if } cs \text{ rel} \text{ true} \\ R[rel'](\text{SourceField}, s), & \text{otherwise} \end{cases}
\]

\[
W[\text{TargetField} = \text{CONST}(C)](s, t) = \pi_{\text{TargetField}}(t) = C
\]

\[
W[\text{TargetField} = \text{FIELD}(\text{SourceField})](s, t)
= \begin{cases} \pi_{\text{TargetField}}(t) \in \text{FlowFilter}\ \pi_{\text{SourceField}}(s), & \text{if } \text{SourceField} \in \text{dom}(\Sigma_{\text{FlowFilter}}) \\ \pi_{\text{TargetField}}(t) = \pi_{\text{SourceField}}(s), & \text{otherwise} \end{cases}
\]

\[
W[\text{TargetField} = \text{FILTER}(F)](s, t) = \pi_{\text{TargetField}}(t) \in \text{Filter} F
\]

\[
C[\text{SourceField} = \text{CONST}(C)](s) = \pi_{\text{SourceField}}(s) = C
\]

\[
C[\text{SourceField} = \text{FILTER}(F)](s) = \pi_{\text{SourceField}}(s) \in \text{Filter} F
\]

For filters \(F\) we have used the informal notation \(\pi_{\text{Field}}(r) \in \text{Filter} F\) simply meaning that the value \(\pi_{\text{Field}}(r)\) is within the filter \(F\) (see 18 for a description of filter expressions). Furthermore we have used the notation \(\pi_{\text{TargetField}}(t) \in \text{FlowFilter} \ \pi_{\text{SourceField}}(s)\) meaning that the value of \(\pi_{\text{TargetField}}(t)\) must be within the value of the FlowFilter SourceField (more on FlowFilters
in the next section).

The ability to use FlowFilters in table relations is really somewhat strange: FlowFilters are not persistent in the database and will often have different values on different clients, which may be one of the reasons why NAV does not maintain table relations as invariants.

2.3.6 FlowFields and FlowFilters

FlowFields can be thought of as *virtual* fields that are computed/derived values, and thus not actual values in the table. The computation of a value in a FlowField may not only depend on the table in which it is defined, but also *exactly one* other table. We introduce FlowFields by an example:

**Figure 9** A FlowField in the table Customer.

Consider the field Balance in the table Customer (see Figure 9). It should be evident that the balance of a customer is derived data, based on the transactions for that customer. And indeed this is the case, since Balance is defined as a FlowField with the following calculation formula (we have simplified the actual formula):

\[
\phi = \text{Sum(Detailed Cust. Ledg. Entry.Amount)} \\
\text{WHERE (Customer No. = FIELD(No.))}
\]

This formula expresses that the value of Balance is the sum of all Amount fields in the Detailed Cust. Ledg. Entry table, for which the value of Customer No. equals the number of the customer we are currently considering. Cf. Figure 2(7) this corresponds to having \( \sum_{\text{FlowField(Balance)}} = \phi \).

So in this example we wish to calculate a sum for each record in the Customer table, which can potentially be time consuming, as we are iterating over the Detailed Cust. Ledg. Entry table. Fortunately this is where we can use Sum Index Fields, which we introduced in Section 2.3.3. In order to utilize SIFT in the example, we must define an index on the table Detailed Cust. Ledg. Entry – which includes the field Customer No. as the first field – such that Amount is defined as a Sum Index Field for that index. In our semantic model this means that

\[
\text{Amount} \in \text{Indexes}_{\text{Detailed Cust. Ledg. Entry}}(\text{Customer No.}, F_1, \ldots, F_n)
\]
for some fields $F_i$, and indeed such and index is already defined, as the reader can check.

In fact NAV will not allow the computation of a FlowField unless all fields used in the filtering are contained in an index with the field being summed over defined as a Sum Index Field. If the index is not defined, the NAV client will produce a runtime error.

**Hypothesis 8:** All fields used in the filter for a FlowField must be included in a NAV key with the field being aggregated over defined as a Sum Index Field. If this is not the case, the NAV client will produce a runtime error. It should be (partially) possible to verify this statically.\(^4\)

In our example we were interested in computing a sum, but FlowFields may compute other functions as well (all with the property that they can be maintained in a B+ tree with logarithmic update/insertion, as was the case for sum). The complete pseudo grammar for FlowField formulae is presented in Figure 10.

**Figure 10** BNF grammar for FlowField expressions.

```
FlowFieldExp ::= Function(TargetTable[.TargetField] WHERE WhereClause*)
Function ::= [\-] Exist | Count | [\-] Sum | [\-] Average | Min | Max | Lookup
WhereClause ::= TargetField = CONST(Constant)
| TargetField = FIELD(SourceField)
| TargetField = FIELD(UPPERLIMIT(SourceField))
| TargetField = FIELD(FILTER(SourceField))
| TargetField = FIELD(UPPERLIMIT(FILTER(SourceField))))
| TargetField = FILTER(Filter)
```

The formal semantics of FlowFields is as follows: Assume we are defining `SourceField` as a FlowField on the table `SourceTable` with the FlowField formula $\phi$ (i.e. $\Sigma_{\text{FlowField}}(\text{SourceField}) = \phi$ in `SourceTable`). Then the value of $\pi_{\text{SourceField}}(s)$ for a given $s \in \text{SourceTable}$ is $[[\phi]](s)$:

```
\[
[[/(\text{TargetTable}\.\text{TargetField} \text{ WHERE } cs)]/(s)]
= \mathcal{F}\{f\{((\pi_{\text{TargetField}}(t) | t \in \text{TargetTable} \land \land_{c \in cs} W[e](s, t))\}
\]
\mathcal{F}[\text{Exist}](S) = S \neq \emptyset
\mathcal{F}[\text{Count}](S) = |S|
\mathcal{F}[\text{Sum}](S) = \sum_{s \in S} s
```

\(^4\)The reason that we can only partially verify the existence of NAV keys is because NAV keys can be disabled at runtime. Thus the existence of a NAV key does not necessarily imply that it is active.
\[ F[\text{Average}](S) = \frac{F[\text{Sum}](S)}{F[\text{Count}](S)} \]
\[ F[\text{Min}](S) = \min(S) \]
\[ F[\text{Max}](S) = \max(S) \]
\[ F[\text{Lookup}](S) = \text{First element of } S \]

\[ \forall [\text{TargetField} = \text{CONST}(C)](s, t) = \pi_{\text{TargetField}}(t) = C \]
\[ \forall [\text{TargetField} = \text{FIELD}(\text{SourceField})](s, t) \]
\[ = \begin{cases} 
\pi_{\text{TargetField}}(t) & \in \text{FlowFilter} \text{ for } \pi_{\text{SourceField}}(t), \text{ if } \text{SourceField} \in \text{dom}(\Sigma_{\text{FlowFilter}}) \\
\pi_{\text{TargetField}}(t) & = \pi_{\text{SourceField}}(s), \text{ otherwise} 
\end{cases} \]
\[ \forall [\text{TargetField} = \text{FIELD}(\text{UPPERLIMIT}(\text{SourceField})))](s, t) = \]
\[ = \pi_{\text{TargetField}}(t) \leq \max\{x \mid x \in \text{FlowFilter} \pi_{\text{SourceField}}(s)\} \]
\[ \forall [\text{TargetField} = \text{FIELD}(\text{FILTER}(\text{SourceField})))](s, t) = \]
\[ = \pi_{\text{TargetField}}(t) \leq \max\{x \mid x \in \text{Filter} \pi_{\text{SourceField}}(s)\} \]
\[ \forall [\text{TargetField} = \text{FILTER}(F)](s, t) = \pi_{\text{TargetField}}(t) \in \text{Filter } F \]

The optional “-” allowed in front of some of the functions (which we have left out for simplicity) is negation (with respect to integers and booleans respectively).

FlowFilters are – like FlowFields – virtual. But whereas FlowFields compute a value for each record in the table, FlowFilters are used as parameters in the calculation of FlowFields (which is already evident from the semantics of FlowFields).

Again we illustrate the concept by an example: Consider the NAV table Currency, which defines the different currencies in the company. This table contains a FlowField Customer Balance, which computes the total balance of all customers working in the given currency. If we were only interested in computing the total balance for all customers, the following FlowField formula would be sufficient:

\[ \text{Sum(Detailed Cust. Ledg. Entry.Amount WHERE Currency Code = FIELD(Code))} \]

If, however, we only wish to compute the balance for a subset of customers, we need somehow to restrict the ledger entries in the WHERE clause, and preferably this restriction should be parametrized: This is exactly what FlowFilters do. Returning to the example, one defines a FlowFilter Customer Filter which has the same type as the destination field we want to filter on. In this example we want to filter on the field Customer No., thus Customer Filter must have type Code[20] (see Figure[11]). In our semantic model, Figure[2](8), we hence have \( \Sigma_{\text{FlowFilter}}(\text{Customer Filter}) = \text{Code}[20] \).

\(^5\)S being a set it does not really make sense to talk about “the first element of S”. When we write this anyway it is because the selection is with respect to a given ordering of S – typically the order specified by the primary key on the underlying table.
The FlowFilter Customer Filter can now be used in the formula from earlier to restrict the set of customers, from which to compute the total balance (we call this formula ψ):

\[
\text{Sum(} \text{Detailed Cust. Ledg. Entry.} \text{Amount}
\]
\[
\text{WHERE (Customer No. = FIELD(Customer Filter),}
\]
\[
\text{Currency Code = FIELD(Code))}
\]

Using the semantics we see that ψ denotes the following function:

\[
[\psi](s) = \sum_{t \in \text{Detailed Cust. Ledg. Entry, where } P(s,t)} \pi \text{Amount}(t)
\]

where

\[
P(s, t) = \pi \text{Customer No.}(t) \in \text{FlowFilter } \pi \text{Customer Filter}(s) \land
\]
\[
\pi \text{Currency Code}(t) = \pi \text{Code}(s)
\]

which is what we would expect.

Even though the FlowFilter from Figure 11 is presented as a field in the table, it has actually nothing to do with normal fields: It is not part of the underlying SQL table schema, and when a user sets a FlowFilter (via a mutator, cf. Figure 2 (13)) it is never stored in the database. Also note that FlowFilters are properties on tables and not on records, thus the value of a FlowFilter is the same for all records (i.e. in the predicate P above, the value \( \pi \text{Customer Filter}(s) \) is the same for all \( s \in \text{Currency} \)).

Figure 12 illustrates this way of thinking about FlowFilters, where the FlowField formula from the last example (ψ) is used to compute customer balance based on the Customer Filter FlowFilter (in this example set to the range 5..10). The reader is encouraged to check that the FlowField values are correct with respect to the semantics of ψ.

One thing that strikes us is the mixing of data definitions (fields) and derived data (FlowFields and FlowFilters) in one NAV object. This makes
Figure 12 Detailed Cust. Ledg. Entry (top) and Currency (bottom).

<table>
<thead>
<tr>
<th>Customer No.</th>
<th>Currency Code</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁ = 1</td>
<td>DKK</td>
<td>10</td>
</tr>
<tr>
<td>t₂ = 5</td>
<td>DKK</td>
<td>7</td>
</tr>
<tr>
<td>t₃ = 10</td>
<td>EUR</td>
<td>2</td>
</tr>
<tr>
<td>t₄ = 7</td>
<td>EUR</td>
<td>6</td>
</tr>
<tr>
<td>t₅ = 13</td>
<td>EUR</td>
<td>10</td>
</tr>
<tr>
<td>t₆ = 6</td>
<td>DKK</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer Filter</th>
<th>Code</th>
<th>Customer Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁ = 5..10</td>
<td>DKK</td>
<td>$\psi(s₁) = \sum {\pi\text{Amount}(t₂), \pi\text{Amount}(t₆)} = 7 + 4 = 11$</td>
</tr>
<tr>
<td>s₂ = 5..10</td>
<td>EUR</td>
<td>$\psi(s₂) = \sum {\pi\text{Amount}(t₃), \pi\text{Amount}(t₄)} = 2 + 6 = 8$</td>
</tr>
<tr>
<td>s₃ = 5..10</td>
<td>USD</td>
<td>$\psi(s₃) = \sum 0 = 0$</td>
</tr>
</tbody>
</table>

Table definitions complex, and we believe it also makes maintenance harder. An alternative approach is to define raw data definitions in one NAV object (which is more like a SQL table), and derived data in another. The derived NAV object may then not only extract data from one table, but possibly multiple tables, making it essentially a database view.

**Hypothesis 9:** Mixing raw data (fields) and derived data (FlowFields) in the same NAV object makes table definitions complex and harder to maintain. Instead: Separate raw data (i.e. SQL table) from derived data (views).

### 2.3.7 Iteration

The final entry in the class interface for NAV tables (Figure 2 (14)) is an iterator. The iterator of a NAV table is a cursor-based implementation of the well-known Iterator Design Pattern [6]. When the cursor of an iterator points to a record in the underlying SQL table, the fields of the record can be accessed, and the record can be modified/deleted.

The most important methods in connection with iteration are: FIND, NEXT, INSERT, MODIFY and DELETE. FIND takes as parameter a filter expression (i.e. a predicate on records, see [15]) and sets the cursor to the first record within the filter. NEXT moves the cursor to the next record within the current filter, and returns an indication of how many steps the cursor has moved (zero means “no records left”, and the number of steps is with respect to an ordering of the records based on a NAV key[6]). This is illustrated in

---

It is possible to change the NAV key using the SETCURRENTKEY method [15]. The
Figure 13 also illustrates how NAV tables maintain a local in-memory copy of the current SQL column. Any (local) changes to this record will not have persistent effect until the changes are committed, either via MODIFY, INSERT or DELETE\(^7\) (see Figure 14, where “time moves vertically”).

We conclude this Section with a brief example on how to use iterators (and in particular how not to use them). Consider a table with \(\Sigma(\text{Name}) = \text{TEXT}[n]\), for some suitable \(n \in \mathbb{N}\). If we want to duplicate all entries in this table and add “duplicated” to the name, then the following piece of code will not always work (\(t\) is an instance of the table we want to duplicate):

```plaintext
// Get all records
Repeat
    t.Name := 'Duplicated: ' + t.Name; // Rename
    t.INSERT; // Duplicate current record
UNTIL t.NEXT = 0 // Stop when no records left
```

The reason why this piece of code is incorrect is because the invocation of INSERT will automatically move the cursor to the newly created record (cf. Figure 14), and thus skip all records in between. To solve this problem, we need two instances of the table (one for iteration and one for insertion), yielding the following correct code:

```plaintext
// Get all records
REPEAT
    t2.TRANSFERFIELDS(t); // Copy in-memory fields
    t2.Name := 'Duplicated: ' + t2.Name; // Rename
    t2.INSERT; // Duplicate current record
UNTIL t2.NEXT = 0 // Stop when no records left
```

fact that iterators always use a NAV key means in particular that the iteration order is deterministic, unlike the SQL standard.

\(^7\)Followed by an implicit or explicit COMMIT, more on this in Section 2.7.1.

---

**Figure 13** FIND and NEXT operation. NAV table instance (left) and underlying SQL table (right).
2.4 Codeunits

In this section we will look more into how business logic is defined in Microsoft Dynamics NAV. A **codeunit** is a collections of methods (procedures in NAV terminology) written in C/AL that are combined in one NAV object, making the logic therein available to other NAV objects. Strictly speaking, codeunits could be removed from NAV, since it is possible to define C/AL procedures on e.g. tables and forms as well, but the idea is to separate business logic from data definition and user interface.
2.4.1 Interface

Like NAV tables, codeunits correspond to classes in OOP that can be instantiated. The class interface for codeunits is given in Figure 15 where the only constant entry is the name of the codeunit.

Figure 15 The semantic model of a NAV codeunit.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Per instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Name ∈ String</td>
<td>(codeunit name)</td>
</tr>
<tr>
<td>2. Built-in method</td>
<td>(OnRun)</td>
</tr>
<tr>
<td>3. Vars : String → in Type</td>
<td>(user-defined instance variables)</td>
</tr>
<tr>
<td>4. Methods : String → in Procedure</td>
<td>(user-defined methods)</td>
</tr>
</tbody>
</table>

Codeunits have one built-in trigger: OnRun. This means that codeunits can be executed (e.g. as the result of pushing a button on a form), but from a conceptual point of view the OnRun trigger is nothing but a parameterless method.

Apart from procedures (Methods), codeunits may contain global variables (Vars), which span all procedures within the codeunit. The body of a procedure may be any C/AL statement – we describe C/AL in the following section.

2.4.2 C/AL

C/AL is an imperative, statically typed programming language, similar to Pascal [9]. Expressions in C/AL evaluate to values and statements are computations with effects – in particular modifications of (persistent) store. The BNF grammar for expressions (Exp) and statements (Statement) are given in Figure 16 and Figure 17 respectively – the grammar is simplified in some places to make it more uniform.

We will not explain the semantics of expressions in detail, rather we will explain some of the constructs informally. The first non-selfexplanatory expression is member access:

\[ \text{Exp}_1.\text{Exp}_2 \]

Here \( \text{Exp}_1 \) must evaluate to an object instance (e.g. a table), and \( \text{Exp}_2 \) must be a member of that objects interface. If e.g. \( \text{Exp}_1 \) is a variable pointing to an instance of table \( T \) and \( \text{Exp}_2 = F \in \text{dom}(\Sigma_T) \), then \( \text{Exp}_1.\text{Exp}_2 \) evaluates to the value \( \pi_F(r) \), where \( r \) is the in-memory copy of the current record (cf. Section 2.3.7).

One important thing to note is that \( \text{Exp}_2 \) may not be any arbitrary expression: If e.g. \( \text{Exp}_1 \) is a table variable, then the value of \( \text{Exp}_2 \) must be known at compile time! This means that one cannot access fields dynamically, which makes it possible to statically type check accesses to the
Figure 16 BNF grammar for expressions.

| Exp      | ::= | Constant               | Constants (strings, integers, floating points, booleans, etc.) |
|          |     | Identifier             | Identifiers (variable names, system calls, etc.) |
|          |     | UnaryExp               | Unary expression |
|          |     | BinaryExp              | Binary expression |
|          |     | NaryExp                | nary expression |
| UnaryExp | ::= | (Exp)                  | Grouping |
|          |     | NOT Exp                | Negation |
|          |     | - Exp                  | Unary minus |
| BinaryExp | ::= | Exp ⊕ Exp              | Arithmetic ($\oplus \in \{+, -, *, /, \text{MOD}\}$) |
|          |     | Exp ⊙ Exp              | Boolean combinators ($\odot \in \{\text{AND}, \text{OR}, \text{XOR}\}$) |
|          |     | Exp ≺ Exp              | Relations ($\prec \in \{<, =, <\leq, >, >\geq, <>\}$) |
|          |     | Exp . Exp              | Member access |
|          |     | Exp .. Exp             | Range |
|          |     | Exp :: Exp             | Scope |
| NaryExp  | ::= | Exp(Exp₁, ..., Expₙ)   | Function call |
|          |     | Exp[Exp₁, ..., Expₙ]   | Array reference |
|          |     | Exp \in [Exp₁, ..., Expₙ] | Membership test |

database (remember that the signature $\Sigma$ of each table is known at compile time).

This restriction means that it is for instance not possible to write a procedure which takes as input any table with a field named $\text{Id}$ of type $T$, and outputs $\pi_{\text{Id}}(r)$, where $r$ is the first record in the table (this can almost be achieved using variables of type $\text{RecordRef}$ – more on this later – however, then no statical type checking is done).

The next non-selfexplanatory expression is range:

$$\text{Exp}_1 \ldots \text{Exp}_2$$

Here $\text{Exp}_1$ and $\text{Exp}_2$ must evaluate to values of the same type, and the type must be ordered. If $v_i$ is the value of $\text{Exp}_i$ then $\text{Exp}_1 \ldots \text{Exp}_2$ simply denotes the range $[v_1; v_2]$ (which is why the type must be ordered). Ranges can e.g. be used when setting FlowFilters.

Next we consider scopes:

$$\text{Exp}_1 :: \text{Exp}_2$$

Scopes are used for fields of type $\text{Option}$, which are enumeration types in the database. Consider a table signature $\Sigma$ for a table $T$. If $\pi_1(\Sigma(F)) =$
Option and (OptionString, ’a,b,c’) ∈ π₂(Σ(F)), then F is defined as an enumeration with possible values a, b and c. Letting x denote an instance of table T, we can then access the enumeration values of F via x.F::a, x.F::b and x.F::c. This can then be used in a test for the actual enumeration value in F for the current in-memory record via:

IF x.F = x.F::a THEN
   <Code for case a>
ELSE IF x.F = x.F::b then
   <Code for case b>
ELSE
   <Code for case c>

Function calls in C/AL have the form:

\[ \text{Exp}(\text{Exp}_1, \ldots, \text{Exp}_n) \]

C/AL uses call-by-value function calls, and parameters can either be passed by reference or as values (how each parameter is passed is specified in the functions formal parameters definition). Each expression \(\text{Exp}_1, \ldots, \text{Exp}_n\) must evaluate to a simply-typed value (e.g. integer or bool) or be a table/codeunit instance, thus it is not possible to pass e.g. a function as a parameter (functions are not first class values). The value of \(\text{Exp}\) must be known at compile time in order to statically type check function calls. Functions can be called recursively.

C/AL includes fixed size, multidimensional arrays (i.e. the size must be known at compile time). Array access has the form

\[ \text{Exp}[\text{Exp}_1, \ldots, \text{Exp}_n] \]

which makes it possible to retrieve and store values in the array.

The semantics of statements should be somewhat clear, so we will not get into detail (again we refer to [13] for an informal description). Having defined expressions and statements informally, we can now define procedures:

\[
\text{Procedure} \ ::= \ [\text{LOCAL}] \ \text{PROCEDURE} \ Identifier([\text{VAR}] \ \text{Var}_1, \ldots, [\text{VAR}] \ \text{Var}_n) [: \text{Type}];
\text{Var} \ :: \ Identifier : \ Type
\]

The optional keyword LOCAL makes a procedure available only to the NAV object in which it is defined (private method in OOP terminology). The optional keyword VAR in the formal parameter list specifies whether the parameter should be passed by reference (VAR set) or as a value. The optional type identifier specifies the return type of the function. Variables \(\text{Var}_1', \ldots, \text{Var}_k'\) are local variables, which are instantiated each time the procedure is invoked.
Figure 17 BNF grammar for statements.

| Statement ::= Empty statement  |
| BEGIN Statement END | Block  |
| Statement; Statement | Sequence  |
| Exp ::= Exp | Assignment  |
| Exp | Expression  |
| IF Exp THEN Statement [ELSE Statement] | Conditional  |
| CASE Exp OF Exp11,...,Exp1k ; Statement1 ;  |
| ...  |
| Expn1,...,Expnkn ; Statementn | Case split  |
| WHILE Exp DO Statement | Loop  |
| REPEAT Statement UNTIL Exp | Loop  |
| FOR Exp ::= Exp TO/DOWNTO Exp DO Statement | Bounded loop  |
| WITH Exp DO Statement | Open table  |
| EXIT(Exp) | Function return  |

All variables in C/AL are explicitly typed, which makes type checking easy, and as mentioned earlier, this makes it possible to type check database access. The BNF grammar for types can be seen in Figure 18. We have included a small description for some of the types – for a (more or less) complete description of types see [15].

We mentioned earlier that it is not possible to access fields dynamically; This is however not entirely true. The type RecordRef above basically means that a variable points to any record of any table. It is then possible to iterate over the fields of this record, but then the type system can make no guarantees with respect to correct access.

Hypothesis 10: The strict type annotations of table instance variables imply that code written for a table cannot be used for other tables. If similar code is needed for two different tables, then the programmer must either duplicate the code, or merge the two tables. Solution: Introduce polymorphism. Rather than having strict type annotations, for instance “x : Record 18”, one could instead use type inference, or explicit subtyping (i.e. “x : Record(F : T)” – meaning x must point to any table with Σ(F) = T).

The last thing to note in connection with the type system of C/AL is the type Variant. Variant is a dynamic type, which means that it can be instantiated with values of different types (see [15] for a list of allowed types). This means that runtime type errors can occur, and thus C/AL is not strongly typed (as was already witnessed via the RecordRef type).

Apart from the conceptual grammar given in this section, C/AL contains tons of built-in system calls (for instance GUI-messaging functionality). We
**Figure 18 BNF grammar for types.**

<table>
<thead>
<tr>
<th>Type</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>SimpleType</code></td>
</tr>
</tbody>
</table>

**SimpleType** ::=  
- `Integer`  
- `Text[n]` A string of maximum length n  
- `Code[n]` A code of maximum length n  
- `Char`  
- `Decimal`  
- `Option` Enumeration type  
- `Boolean`  
- `Date`  
- `Time`  
- `DateTime` Binary data of maximum length n  
- `BLOB` Large binary data  
- `DateFormula` Date formulae  
- `TableFilter` Unknown  
- `BigInteger`  
- `Duration` Difference between two DateTime's  
- `GUID` Global Unique Identifier  
- `RecordID` Id (name) of a NAV table

**ComplexType** ::=  
- `Action` Unknown  
- `RecordRef` Reference to *any* record  
- `Dialog` Dialog window  
- `Variant` Dynamic type (!)  
- `InStream`  
- `OutStream`  
- `FieldRef` Reference to *any* field  
- `KeyRef` Reference to a key definition in *any* table  
- `File`  
- `[TEMPORARY] Record n` A (temporary) instance of table n  
- `Codeunit n` An instance of codeunit n  
- `Form n` An instance of form n  
- `Report n` An instance of report n  
- `XMLport n` An instance of XMLport n  
- `Automation identifier` Automation object (see [15], Chapter 19)  
- `OCX identifier` OCX object (see [15], Chapter 19)  
- `ARRAY [n1, ..., nk] OF Type` Multidimensional, fixed size array  
- `'enum1, ..., enumn'` (Constant) enumeration type  
- `TextConst String` Multilanguage string

have deliberately left these out in this document, and we refer to [15] and [18] for a more complete description. A complete grammar for C/AL (including complete grammars for tables, codeunits, forms and reports) can be found in Appendix B. The grammar has been constructed by reverse engineering in collaboration with Michael Nissen (DIKU) – to our surprise no official grammar existed before our attempt! The grammar is specified in F#’s parser generator language [19]. The tedious work on constructing the parser has opened up for a wide range of interesting topics for future work (Section 5).
2.5 Forms

In this section we briefly discuss forms in NAV. A form in NAV is much like a Windows Form: A form contains various controls for displaying and altering data. Like tables and codeunits, forms have triggers (but unlike tables these are “proper” triggers) – some of which have form scope (i.e. one per form) and some of which have control scope. Triggers can be used to activate C/AL code (e.g. when the user pushes a specific button), and like tables and codeunits, forms can contain user-defined methods and variables as well.

2.5.1 Interface

The class interface for a NAV form is presented in Figure 19.

![Figure 19 The semantic model of a NAV form.](image)

<table>
<thead>
<tr>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Name ∈ String (form name)</td>
</tr>
<tr>
<td>2. SourceTable ∈ String ∪ {unbound} (source table)</td>
</tr>
<tr>
<td>3. c : N →fin Control × P(Property) (controls)</td>
</tr>
<tr>
<td>Per instance</td>
</tr>
<tr>
<td>4. Built-in methods (OnOpenForm, OnNextRecord, etc.)</td>
</tr>
<tr>
<td>5. Vars : String →fin Type (user-defined instance variables)</td>
</tr>
<tr>
<td>6. Methods : String →fin Procedure (user-defined methods)</td>
</tr>
</tbody>
</table>

All forms have a name (1) and can be bound to at most one table (2). Having a form bound to a table means that each instance of the form implicitly has an instance of the bound table, from which data can be shown. Besides from making the table instance visible in the source code of the form, the runtime system automatically generates GUI access to filters, and automatically invokes table triggers (this is why the “triggers” of Section 2.3.2 are really GUI triggers).

Figure 20 shows the NAV form Customer Card, which shows information about a given customer. The form is bound to the table Customer, which means that one can iterate through all customers using \[ \text{Pg Up} \] and \[ \text{Pg Dn} \]. But it also makes it possible to change customer data, by changing the value in one of the controls – and this will automatically invoke the appropriate NAV trigger(s) on the Customer table. If we e.g. change the value in the text box Name, then the runtime system will automatically invoke the OnModify trigger for the field Name. Similarly, if we change the value in the text box No., then the OnRename trigger will automatically be invoked, as No. is part of the primary key on the Customer table.

Having a form bound to a table does not automatically imply that all fields of the table are presented on the form. All visible controls on the form must (in principle) be added manually, and it is then possible to hook up e.g. a text box to a field of the table. In Figure 20 we have for instance
that $\pi_1(C(2)) = \text{TextBox}$ and $(\text{SourceExpr, No.}) \in \pi_2(C(2))$, meaning that the control with identifier 2 is a TextBox, and it is bound to the field No. of the underlying table, Customer.

It is possible to auto generate the controls for a new form, based on the table it is bound to. But when an underlying table is updated, there is no means of automatically updating the existing form(s) bound to the table.

**Hypothesis 11:** Forms can only be bound to a single table, which exacerbates the use of unnormalized database design: If a developer wishes to show the compound data from two tables in a form, then the easiest solution is to simply merge the two tables into a (Cartesian) product table. Solution: Generalize source tables to source views instead.

### 2.6 Reports

The last NAV object type we will describe in this section is reports. We will not discuss layout issues, instead we will focus on how data retrieved and calculated.

#### 2.6.1 Interface

The semantic model of a NAV report is presented in Figure 21.

Like forms, reports are bound to tables. But unlike forms, reports can in general be bound to several tables, which are called data items. Data items thus define the data from which reports are calculated. When there are no dependencies between the data items, nor any post-processing of the retrieved data, then the output from a report is simply

$$\bigcup_{n \in \text{dom(DataItem)}} \sigma(\pi_1(\text{DataItem}(n)))$$
Figure 21 The semantic model of a NAV report.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Per instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Name ∈ String</td>
<td>3. Built-in methods</td>
</tr>
<tr>
<td>2. DataItem : N →fin String × P(Property)</td>
<td>(OnPreReport, OnAfterGetRecord, etc.)</td>
</tr>
<tr>
<td></td>
<td>4. Vars : String →fin Type</td>
</tr>
<tr>
<td></td>
<td>5. Methods : String →fin Procedure</td>
</tr>
</tbody>
</table>

(here we use the informal notation $\sigma(\pi_1(DataItem(n)))$ denoting all records in the table with the name $\pi_1(DataItem(n))$).

It is possible to nest data items, which means that a data item can depend on exactly one other data item. We illustrate this by an example: Consider the NAV report VAT Statement in Figure 22.

Figure 22 NAV report VAT Statement.

This report has one root data item, VAT Statement Name, and one nested data item, VAT Statement Line. In the semantic model this means that

$$\pi_1(DataItem(1)) = \text{VAT Statement Name}$$
$$\pi_1(DataItem(2)) = \text{VAT Statement Line}$$

Furthermore, the second data item is nested under the first data item, which in the semantic model means that

$$(\text{DataItemLinkReference}, 1) \in \pi_2(DataItem(2))$$

i.e. the second data item links to the first data item. This means that the VAT Statement Line table is iterated (inner loop) for each record in the VAT Statement Name (outer loop). However, we do not iterate through all records in VAT Statement Line, instead we only iterate through those records that meet certain join conditions.

In the example above, the join condition is that for each record $n$ in the VAT Statement Name table, we iterate through those records $l$ in the VAT Statement Line table that satisfy

$$\pi\text{Statement Template Name}(l) = \pi\text{Statement Template Name}(n)$$
$$\pi\text{Statement Name}(l) = \pi\text{Name}(n)$$
(this is specified in the DataItemLink property, cf. Figure 22). But this means that we wish to process exactly the records (again using notation from relational algebra)

\[ \text{VAT Statement Name} \bowtie_P \text{VAT Statement Line} \]

where \( P \) is the predicate

Statement Template Name = Statement Template Name \( \land \) Name = Statement Name

In general the DataItemLink property can only depend on the immediate parent data item, thus all data in NAV reports correspond (in principle) to pairwise joined tables.

In the example above, one would imagine that the following query would be sent to Microsoft SQL Server to generate the data set:

```
SELECT * FROM [VAT Statement Name], [VAT Statement Line]
INNER JOIN [VAT Statement Line] ON
[VAT Statement Name].[Statement Template Name] = [VAT Statement Line].[Statement Template Name] AND [VAT Statement Name].Name = [VAT Statement Line].[Statement Name]
```

But after doing a database trace, it turned out that all joins were performed manually via nested loops, instead of utilizing joins and automatic query optimization.

**Hypothesis 12**: The implementation of reports in Microsoft SQL Server does not utilize joins and the performance gains possible if the joined fields are indexed. Instead data is retrieved via nested looping making it asymptotically slower.

As mentioned earlier, it is possible to do post-processing of the data retrieved via data items. This is done via triggers, which can have either report, data item or record scope. For example, all data items have an OnAfterGetRecord trigger, which is executed after the retrieval of a record from the corresponding table. Since this trigger can contain arbitrary C/AL code, one could for instance access another table (not specified in the DataItem map), and include data from this table in the report. However, we do not believe this is often used, as all data relevant to a report should really be defined as data items – all other parameters used in the report should contain static values. This suggests that C/AL is perhaps too expressive for post processing, and maybe a less expressive language should be used ([16] is an example of a declarative, domain specific language for ERP reporting).

**Hypothesis 13**: The underlying data set used in a NAV report can be expressed directly as a statement in SQL using joins. However, the post-
processing can contain arbitrary C/AL code, making the report infeasible for direct conversion to e.g. SQL.

Hypothesis 14: NAV developers are already familiar with table joins from reports. The same user interface could be used to introduce views in NAV, making it more feasible to use normalized database design (the developer would not need to hand-write table joins).

2.7 Runtime system (C/SIDE)

C/SIDE is the runtime system of NAV. By runtime system we mean the client, since it is the client that is running all business logic (cf. the introduction). Having described the “building blocks” of NAV as classes in OOP, the runtime system corresponds to e.g. the virtual machine of Java, since it executes instances of compiled NAV objects.

More specifically, the C/SIDE client interprets compiled C/AL code for codeunits, generates GUI for forms and reports, and links NAV tables to their underlying SQL tables.

2.7.1 Concurrency

Microsoft Dynamics NAV is – as mentioned in Section 1 – a multiuser system, thus it has to deal with concurrency. We shortly describe the facilities in NAV used to avoid concurrency problems (again we will only consider the case where NAV is running on Microsoft SQL Server).

The first thing we describe is how NAV handles lost updates. Consider the following example: Two processes (object instances) wish to update the same field in the same record, by incrementing the field by 100 and 200 respectively. When processed in serial, the result will be a total incrementation of 300. However, when processed in parallel, the total result can be an incrementation of either 100, 200 or 300 – the two first cases are due to lost updates. NAV handles this by adding time stamps to all(!) records in the database. Each time the record is updated, the time stamp is updated as well, and the update is only carried out if the time stamp has not been updated in the meantime. This scenario is described in Figure 23, where the right process will not be able to commit its update, as the left process has made an intermediate update (we have used integers for time stamps).

In other words, Microsoft Dynamics NAV uses optimistic concurrency control [2]. If an update fails due to an incorrect version, then all changes to the database in the current transaction are not committed. C/SIDE has built-in support for transactions: each execution of C/AL code from the user interface starts with an implicit “start write transaction”, and the transaction is either committed explicitly via COMMIT or implicitly upon completion of the execution (when control is given back to the user in the
Figure 23 Time stamps are used to prevent lost updates.

user interface). This means that either all updates take place, or no updates take place. Thus if e.g. the right process in Figure 23 had made another update prior to the failing MODIFY without first COMMITing the update, then that update would not be executed.

Transactions are implemented by acquiring locks, however it is unclear to us exactly what is locked and when it is locked. [15] has some information on this, but it is still very unclear what locking principles are used.

Question: Microsoft Dynamics NAV does not scale well beyond 250 users. Is this due to the specific concurrency control protocol used in the database? If so, is it then because of the method used for locking or because of the unnormalized database design (decomposition of tables could minimize the data that actually needs to be locked)?

2.7.2 NAV 2009

As mentioned in the beginning of this document, we have used Microsoft Dynamics NAV 5.0 SP1 throughout our analysis. A new version of NAV – Microsoft Dynamics NAV 2009 – has however recently been released. So will our analysis make any sense in the new setting?

One of the new things in NAV 2009 is the use of a three-tier architecture, i.e. all business logic is moved from the client to running centrally on an application server. It is, however, the same C/AL code that is being executed as in the old version of NAV. So even though NAV 2009 introduces some new features (such as a “Role-tailored client” and SQL reports) it still uses the exact same database scheme and business logic (and data source for reports).

We therefor claim that the observations made in this document are still valid for the new version of NAV.
3 NAV Design Patterns

In the last section we described what the different NAV objects are. In this section we wish to describe how these object are (typically) used (i.e. NAV design patterns). We classify different kinds of tables, codeunits, forms and reports – note however that these are only design guidelines; In principle the programmer can design NAV objects in any way desirable, using the possibilities described in the previous section. The division of NAV objects described in this section is based on a similar division in [18].

3.1 Tables

3.1.1 Master tables

Master tables contain primary/master data, such as customers (table 18) and items (table 27). Rows in a master table can be thought of as entities, and since NAV lacks joins, these tables will often contain many columns and be sparse (many null values). The Items table for instance contains 175 columns! Master tables typically use card forms as primary input (will be explained later).

3.1.2 Templates

Templates are used in connection with journals (below). A row in a template table characterizes a set of rows in a journal, by describing common control information such as type of journal, numbering series, account numbers, etc. An example of a template table is table 80 (Gen. Journal Template). This table divides general journals into classes, such as “assets”, “payments”, etc. Templates can be thought of as configuration, as they are not often changed.
3.1.3 Journals

Journals contain *unposted* data, i.e. information that may possibly change before it is posted to the ledger. Each row in a journal is associated with a row in a template table (above). Table 81 (Gen. Journal Line) is an example of a journal table, which contains all lines not yet posted to the general ledger. Journals typically use tabular forms as input (will be explained later).

3.1.4 Ledgers

Ledger tables contain *posted* data, i.e. events that cannot – and in fact, by law, should not – be deleted (in some cases, however, they can be *reversed*). Data in ledger tables is considered “accounting data”, i.e. data which is used for auditing. An example of a ledger table is G/L Entry (table 17). Data cannot be inserted manually in a ledger table, instead journal lines are posted via a *posting routine* to become ledger lines. The posting routine then makes sure that all entries are balanced against dual entries, maintaining the invariants of *double-entry bookkeeping* [20][1][1].
3.1.5 References

Reference tables are somewhat like master tables, only data is always referred rather than copied. An example of a reference table is Post Code (table 225), which contains a list of postal codes. If this table for instance contains the record (DK-2300, Copenhagen South), and an address refers to the code DK-2300 (which is the primary key), then any changes to the postal code will automatically be reflected in the address.

Opposed to this, data from the customer table is copied, for instance when a sales order is created. This is because the address on a sales order must be that of the time when the order was established, and not the present address (which may be different).

3.1.6 Registers

Register tables are used to group entries in ledger tables that belong to the same posting. Whereas ledger entries contain posting dates, registers contain the actual date the entries were posted (which need not be the same). Registers are used for auditing – an example is table 45 (G/L Register), which groups entries in the G/L Entry table.
3.1.7 Posted documents

Posted documents are – like registers – used to group ledger entries. But whereas registers serve auditing purposes, posted documents are only used for easier reference to the posted data (and can thus in principle be deleted). Posted documents are examples of the “history problem”. By history problem is meant the fact that some data needs to be copied for historical reasons, e.g. the address of a customer or the name of an item on an invoice, as explained earlier.

**Hypothesis 15**: The history problem is handled in NAV by explicit copying of data. This implies that a change in the data definition of historic data must be changed in all places where it is copied (since the data definition/schema is copied as well). An alternative solution is to use versioning, where historical data refers to a given version of data rather than copying it – avoiding propagation of changes when upgrading.

An example of a table used for posted documents is Sales Invoice Line (table 113), which describes items that were sold. Ledger entries are linked to their originating documents via a document number, making it possible to navigate from a ledger entry on a sales account to the sales invoice from which it originated.
3.1.8 Setup tables

Setup tables are special cases of reference tables, which only contain a single row. Data in this row is configuration – typically used to configure a single functional area (Financial Management, Sales & Marketing, etc.). Configuration data is typically used as control information in C/AL code – for instance, table 98 (*General Ledger Setup*) contains information on how to perform rounding, which is used in posting routines.

![General Ledger Setup Table](image)

3.1.9 Virtual tables

The last table type is virtual tables. Virtual tables are not real tables (in particular, they are not linked to an SQL table), but the interface to virtual tables is similar to that of normal tables. Virtual tables can be used e.g. to access data from the file system – an example is the virtual table *File*, which gives access to a physical file.

Another example is the table *Date*, which contains “all dates”. This is useful e.g. when defining a report on a table T, where rows should be grouped by date. One then defines *Date* as a data item (cf. Section 2.6) and T as a nested data item, joining the two items on the date field of T. A more elegant solution would of course be to have grouping (as in SQL), but this is not the case for NAV.

3.2 Codeunits

3.2.1 Table independent libraries

The first codeunit pattern we will describe is what we call a “table independent library”. What we mean by this is a codeunit that provides functionality, which is not bound to a particular table object. However, it may depend on information from setup tables (Section 3.1.8), which is data that is seldom changed. In principle one could imagine that a table independent library could be specialized with respect to a given configuration, which would minimize code, and be more efficient by eliminating runtime checks.
Examples of table independent libraries include codeunit 365 (Format Address), codeunit 397 (Mail) and codeunit 412 (Common Dialog Management).

### 3.2.2 Table dependent libraries

The second codeunit pattern is the dual of table independent libraries, namely “table dependent libraries”. Table dependent libraries are somewhat similar to bound forms (Section 2.5) in that they provide functionality for a set of tables which are logically related. What we mean by this is tables that typically belong to the same functional area (e.g. accounting, orders, etc.).

Examples include codeunit 3 (G/L Account-Indent, using tables G/L Account and IC G/L Account), codeunit 60 (Sales-Calc. Discount, using e.g. tables Cust. Invoice Disc., Sales Header and Sales Line) and codeunit 220 (Resource-Find Cost, using table Resource Cost).

### 3.2.3 Posting routines

The final pattern we have identified is posting routines, which is a special case of table dependent libraries. Posting routines are typically divided into three codeunits:

1. Check line
2. Post line
3. Post lines

A posting routine takes a set of journals (Section 3.1.3) and posts them as ledger entries (Section 3.1.4), and group them together in one register (Section 3.1.6). This is done by calling (3), which for each unposted line calls (1) to check that the line is valid for posting, and if so calls (2) to do the actual posting.

Codeunit 11 (Gen. Jnl.-Check Line), codeunit 12 (Gen. Jnl.-Post Line) and codeunit 13 (Gen. Jnl.-Post Batch) make up a posting routine for posting general journals.

### 3.3 Forms

#### 3.3.1 Card forms

A card form is used to display a single record from the table to which it is bound. Card forms typically show data from master tables or setup tables. Form 21 (Customer Card, cf. Figure 20) is an example of a card form, which is bound to the Customer table.
3.3.2 Tabular/list forms

Tabular/list forms are used to show a list of records from the table to which it is bound. Typically one has a list form for a table exactly when there is also a card form for that table. Form 22 (Customer List) is an example of a list form.

![Customer List Form](image)

3.3.3 Header/detail forms

Header/detail forms are somewhat different from card and list forms since they can show data from two tables. The form is still bound to a single table T1 (the header information), but it contains a sub form (a list), which is bound to another table T2 (the detail information). This list contains those rows of T2 that are related to the current row of T1, i.e. there is a one-to-many relation between T1 and T2.

Form 42 (Sales Order) is an example of a header/detail form, which is bound to the table Sales Header. The detailed information is sales lines, which are retrieved from the Sales Line table (the one-to-many mapping is on the field No. of Sales Header and Document No. on Sales Line).

![Sales Order Form](image)
3.4 Reports

In our description of reports in Section 2.6 we were only interested in the data aspects of a report, and not the visualization of data. In this section we will follow that approach, and make a division only on how/what data is retrieved, and not in what ways it can be visualized.

In [18] reports are divided into “lists”, “documents”, “transactions”, “tests” and “postings”. In our opinion this distinction is entirely on the visualization level, and data is retrieved in the exact same way in all cases (as mentioned in Section 2.6 the data used in reports is in fact always a selection with joins).

There is however one important distinction in reports, namely those that are read-only, and those that are processing-only. Read-only are the most normal reports, and one can argue that these are in fact the only “proper” reports, in the sense that a report should be a side-effect free function on data, and in particular not modify data. An example of a read-only report is report 111 (Customer - Top 10 List), which calculates a list if customers who bought most.

Processing-only reports are used to update data, and will not produce an output, and as mentioned we find the “report” terminology misleading.

**Hypothesis 16:** Processing-only reports exist because reports have the ability to join tables (via data items), making it easier to traverse a set of tables, rather than having to hand-write join procedures in a codeunit. However, processing-only reports have the same performance problems as mentioned in Hypothesis 12.

Report 794 (Adjust Item Costs/Prices) is an example of a processing-only report, which updates the price of a set of items with respect to an adjustment factor.
4 Architectural Redesign

In this last section we describe our attempt at transforming the existing NAV architecture into a modularized design, and why that approach did not work. We conclude with proposing another method, which relies on expanding the existing functionality described in Section 2.

One might ask why we are at all interested in having a modularized design, and what exactly is meant by such a design. Today all NAV functionality is spread across tables, codeunits, forms and reports – as described in Section 2 – with no logical grouping of functionality (e.g. customers, currency management, VAT management, etc.). This means that for instance VAT functionality is spread across multiple NAV objects, and it is not possible to see exactly which NAV objects are involved without inspecting the code. This is the first reason why modularization is desirable; simply to make it easier for developers to find the source code for a particular functionality, which will ultimately lower the time needed for customizations/upgrades and thus lower total cost of ownership (TCO).

Another aspect is updates. Having a modularized design in which all modules have a formally defined interface (e.g. a class interface), it should be possible to update such a module – preserving the interface – and then plug in the updated version in the ERP system, without having to worry about the other modules. And to make the update even more reliable, we suggest using behavioral interfaces (e.g. session types), which not only specify the legal module invocations, but also the order in which methods should be invoked.

But in fact we believe modularization is not just desirable, we believe it is necessary. Extending and maintaining a big system such as Microsoft Dynamics NAV means that a lot of developers must work on the source code simultaneously. Without a modularized design one can easily imagine the need for simultaneous updates of the same NAV objects, which means that the updates need to be merged (and re-tested). As the tightly-coupled system grows even more complex, this procedure will become even more troublesome, if not impossible. Using a modularized design it should be easier to make non-overlapping updates by having developers working only on disjoint modules (which are in a sense self-contained).

**Hypothesis 17:** Modularization of the NAV architecture is desirable with respect to maintenance and upgradability, and a necessity when NAV is to be extended with even more features in the future.

Our approach was based on grouping existing NAV objects into modules. The objects within a module were allowed to have a high level of interdependency, but should be somewhat more loosely coupled from the other modules. In the analysis we needed a tool for computing dependency graphs, i.e. a graphical visualization of dependencies between NAV objects.
Since no such tool existed, we built one ourselves, which included object references (in C/AL), table relations and source tables (forms) – but omitting e.g. FlowFields (the tool was created before the development of the C/AL parser, which is needed to include FlowField relations). The tool was restricted to tables, codeunits and forms as well.

Figure 24 shows how the tool is used to compute all dependencies for the Customer Card form (forms are displayed in orange, codeunits in black and tables in green).

Besides from rendering dependencies between NAV objects, the tool has support for **components**, which are collections of NAV objects (and components). The dependencies of a component is the union of the dependencies of the objects in the component. For instance one can define a “General ledger setup” component (in XML-style notation) via:

```xml
<Component>
  <Name>General ledger setup</Name>
  <Id>1</Id>
  <NAVObjects>
    <NAVObject id="118" type="Form" name="General Ledger Setup" />
    <NAVObject id="98" type="Table" name="General Ledger Setup" />
    <NAVObject id="426" type="Codeunit" name="Payment Tolerance Management" />
  </NAVObjects>
</Component>
```

Using this facility we tried to modularize the architecture via components, but it turned out that the interdependency levels were in general far to high to make a reasonable modularization. For instance the Customer table has around 50 dependencies and around 150 dependants. On average the tool reported around ten dependencies per object – and again only taking into account a subset of dependency types. Thus we believe modularization via simply collecting existing NAV objects in components is not feasible.

So another method is needed in order to modularize the functionality of Microsoft Dynamics NAV. The problem with our approach is that NAV objects often contain functionality/data that is not logically related: Consider
for instance the **Customer** table. This table contains customer data, but it also contains a lot of other data which is not directly customer related. An example is shipping information (ten fields), which is used to describe various information on how to ship goods to the customer. A better approach is to factorize the shipping information into another table (which must be part of the “shipping module”), and then add a relation between customers and shipping information.

The problem with decomposition of tables is that it is not possible to join tables in NAV (as mentioned in Section 2.3). Thus the introduction of joins (or in general views) into NAV is crucial if such decomposition should be made possible. We mentioned earlier the desire for a normalized database design. Besides from ensuring data integrity, this also makes customization/upgrading easier, as changes to table definitions need not be propagated to other copies of the same definitions.

Another problem is **backwards compatibility**. Since NAV is customized by partners, it must be backwards compatible (or at least not hard to update existing customizations to the new architecture). This suggests introduction of a new object type, **view**, which can be used to resemble the old **Customer** table, but such that data is retrieved from multiple underlying tables. This also makes it possible to move all FlowField and FlowFilter definitions from tables to views, such that tables only contain proper data definitions, and not derived data. It should then be possible to use views in all places where a table was previously used – in particular reports could use a view as data set rather than data items.

In a similar fashion, codeunits need to be modularized as well. This, however, does not imply the need for new object types, as one existing codeunit should be divided into logically different codeunits, which need no relation to each other. For backwards compatibility it is possible to create a resembling codeunit, which calls the functionality in the divided units. This kind of modularization is also referred to as **code refactorization**, for which tools exist. An example is [7], which may also be useful in decomposition of tables.

Finally we must consider forms. The only thing needed is to accommodate the changes made in tables and codeunits. Changes to codeunits is again an instance of code refactorization, and decomposition of tables should be accommodated by allowing views (the new object type) to be the underlying data source, rather than just tables.

Now the question is to what extent this decomposition/refactorization can be done automatically. We do not believe it is possible to fully mechanize the procedure, but we believe the process will be tool-supported. The first thing that is needed is a more fine-grained dependency analysis; rather than registering object dependencies (as is the case for our “NAV Object Analyzer” tool), we need to register dependencies on procedure level (codeunits) and field level (tables). In order to do this analysis, the C/AL parser
from Appendix B is needed.

For tables it would furthermore be interesting to divide dependencies into read-only and read/write, for it is likely that read/write dependencies are logically more related than those that are read-only. This again involves the C/AL parser and an access pattern analysis (INSERT, MODIFY and DELETE, cf. Section [2.3.7]).

A final interesting experiment is to apply a tree decomposition algorithm [17] to the fine-grained dependency graph, in order to get a first approximation of a modularized design. With such an analysis we will also get an idea of the dependency complexity, which in the graph theoretic setting is known as the tree width.

5 Conclusion & Future Work

We have presented a detailed, high-level description of the ERP system Microsoft Dynamics NAV. Our presentation is different from existing literature on NAV, as it targets mainly computer scientist. The presentation is given in an object-based fashion, making it useful as a starting point if C/AL is to be ported to an object oriented language.

We have presented some problems regarding performance and upgradability, and given concrete ideas for how these problems can be solved. Our solutions take into account that any modifications of the base product must be backwards compatible. The main contributions of this report are:

- The current implementation of Sum Index Field Technology in Microsoft SQL Server is non-optimal. SIFT is considered one of the main technologies (i.e. selling points) in Microsoft Dynamics NAV, but we believe the competitive advantage has been lost. We propose an asymptotically faster solution.

- Table relations are rather complex, and may result in unintended dangling references. We propose using simple foreign key constraints that are checked automatically by the SQL server (and which prevent dangling references).

- The design of NAV exacerbates an unnormalized database design, witnessed by the strict type annotation in C/AL, lack of database joins and tailoring of forms to single tables. We propose introduction of polymorphism/type inference (or explicit subtyping) to C/AL, and introduction of joins/views.

- The unnormalized database design of NAV makes it hard to modularize functionality into “logically related” components. Introducing views and joins (above) will enable easier decomposition of tables, and provide means for backwards compatibility.
• Data retrieval in reports is sub-optimal. Rather than processing data by means of coded joins (nested looping) the query optimizer of Microsoft SQL Server should be utilized.

5.1 Future Work

What is perhaps most interesting about our work in this report, is the possibilities that it opens for future work. In a sense, this report provides a lot of the “hard work” on understanding what is going on in NAV, and in particular in the construction of a context-free grammar (which did not exist prior to this report). Below we list some of the interesting topics for future work (some of which have already been initiated), furthermore we categorize these into “immediate” changes and “deep” changes:

• Dependency analysis (immediate). Extension of the “NAV Object Analyzer” tool to include more fine-grained dependencies. Dependencies at field-level (with read/write analysis) for table objects is useful in the process of decomposing tables. Relies on the C/AL parser.

• Clustering analysis/modularization of NAV (deep). Apply a tool like the above in modularizing a realistic part of NAV (for instance accounting). A proof-of-concept would be interesting for evaluating the possibility – and complexity – of a complete system modularization.

• Views (deep). Investigate what is needed in order to extend NAV with a view object type. In particular, it should be investigated how views with update can be handled (read-only views almost come for free). Updatable views are needed if views are to be used for backwards compatibility.

• Reimplementation of SIFT (immediate). Investigate possibilities of implementing SIFT using the augmented search tree algorithm described in this report. We label this change “immediate”, as it will not affect the users or developers of NAV directly (though they should be able to see a speedup!).

• Reimplementation of data retrieval in reports (immediate). Investigate possibilities of retrieving data from Microsoft SQL Server using SQL joins, rather than nested looping. The same comment about “immediate” above applies here.

• Extend C/AL with polymorphism/type inference or explicit subtyping (deep). We already described how the strict type annotations in C/AL prevent code to be used on similar table objects. An interesting

---

8By immediate we mean changes that will not require extensive adaption by NAV users and NAV partners.
project – which should be carried out before extending C/AL – is to classify/identify the various kinds of code duplication in C/AL. It is our thesis that most of the code duplication in NAV is due to lack of polymorphism – and not just because of bad programming style. This thesis should be investigated.

- Translation of C/AL to an object oriented programming language, for instance C# (deep). We have provided a high-level class interface for each NAV object type. With this description as basis, it would be interesting to investigate possibilities of translating the (old-fashioned) C/AL code to a more up-to-date programming language like for instance C#.

- Investigate possible concurrency- and locking issues in NAV (deep). As described in the question on page 35, NAV does not scale well beyond 250 simultaneous users. Supposedly this is due to locking of tables, so it would be interesting to investigate how locking is done (and when). According to [10], the maintenance of SIFT indexes plays a big role in database locking – hence this investigation may overlap with the SIFT reimplementation mentioned earlier.

Acknowledgements

I am grateful to Fritz Henglein for fruitful discussions and proof reading of the report. I thank Peter Bünger, Thomas Jensen, Jesper Kiehn, and Hans Kierulf of Microsoft Dynamics, Vedbæk, Denmark for their inputs to the report, and for helping me understand the more exotic parts of Microsoft Dynamics NAV.

References


A Glossary

The table below summarizes a list of the most important terminology used in Microsoft Dynamics NAV. For each term we put a reference to the page where it is introduced, and a small description.

<table>
<thead>
<tr>
<th>NAV terminology</th>
<th>Page</th>
<th>Description/CS terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>3</td>
<td>Class (OOP)</td>
</tr>
<tr>
<td>Table</td>
<td>4</td>
<td>Class (OOP – database layer)</td>
</tr>
<tr>
<td>Field</td>
<td>4</td>
<td>Column (relational databases)</td>
</tr>
<tr>
<td>Trigger</td>
<td>6</td>
<td>Method (OOP)</td>
</tr>
<tr>
<td>Record</td>
<td>5</td>
<td>Row (relational databases)</td>
</tr>
<tr>
<td>Key</td>
<td>7</td>
<td>Key or index</td>
</tr>
<tr>
<td>Primary key</td>
<td>7</td>
<td>Primary key (relational databases)</td>
</tr>
<tr>
<td>Secondary key</td>
<td>7</td>
<td>Index (relational databases) with aggregated data (Sum Index Field Technology)</td>
</tr>
<tr>
<td>Table relation</td>
<td>12</td>
<td>Generalized foreign key constraints (relational databases), but not invariants</td>
</tr>
<tr>
<td>FlowField</td>
<td>18</td>
<td>Virtual column in a table (typically containing aggregated value from some other table)</td>
</tr>
<tr>
<td>FlowFilter</td>
<td>20</td>
<td>Parameter used in the calculation of FlowFields (instantiated at runtime)</td>
</tr>
<tr>
<td>Codeunit</td>
<td>24</td>
<td>Class (OOP – business logic layer)</td>
</tr>
<tr>
<td>Procedure</td>
<td>24</td>
<td>Method (OOP)</td>
</tr>
<tr>
<td>C/AL</td>
<td>24</td>
<td>The domain specific language of NAV</td>
</tr>
<tr>
<td>Form</td>
<td>30</td>
<td>The GUI through which the end-user interacts with NAV. Forms are typically bound to a table, from which data is presented.</td>
</tr>
<tr>
<td>Report</td>
<td>31</td>
<td>Normally; derived data from (pairwise joined) tables. Sometimes; update of data in tables (batch run)</td>
</tr>
<tr>
<td>Data item</td>
<td>31</td>
<td>Table (relational databases)</td>
</tr>
<tr>
<td>C/SIDE</td>
<td>34</td>
<td>The NAV client</td>
</tr>
</tbody>
</table>
B C/AL Parser

B.1 Abstract Syntax Tree

```plaintext
// Expressions
and 'a exp = |
\| Int of int
\| Real of string
\| Date of string
\| Time of string
\| String of string
\| Ident of 'a identifier
\| True
\| False
\| Opr of string * 'a exp list
\| Call of 'a identifier * 'a exp list
and 'a exp = 'a exp * 'a

// Statements

```

```plaintext
module CALAST

```
null
// Properties allowed on a field
type 'a fprop =
  | FIPTrig of string * 'a procedurebody
  | FIPAutoSearchField of string
  | FIPAutoFormatExpr of 'a exp
  | FIPAutoFormatType of int
  | FIPIncrement of bool
  | FIPBlankNumbers of string
  | FIPBlankZero of bool
  | FIPCalcFormula of string
  | FIPCaptionClass of 'a exp
  | FIPCaptionML of string
  | FIPClosingDates of bool
  | FIPCompressed of bool
  | FIPDateFormula of bool
  | FIPDecimalPlaces of string
  | FIPDescription of string
  | FIPEditable of bool
  | FIPFieldClass of string
  | FIPInitValue of string
  | FIPMaxValue of string
  | FIPMinValue of string
  | FIPNotBlank of bool
  | FIPNumeric of bool
  | FIPOptionCaptionML of string
  | FIPOptionString of string
  | FIPSubType of string
  | FIPTableIDExpr of 'a exp
  | FIPTableRelation of bool
  | FIPTestTableRelation of bool
  | FIPValidateTableRelation of bool
  | FIPValuesAllowed of string

and 'a fprop = 'a fprop * 'a

type 'a fieldproperties = 'a fprop list

type 'a field = string * 'a fieldproperties

type 'a fields = 'a field list

// Properties allowed on a table
type 'a tprop =
  | TAPTrig of string * 'a procedurebody
  | TAPPerm of 'a perm list
  | TAPCaptionML of string
  | TAPDataCaptionFields of string
  | TAPDataPerCompany of bool
  | TAPDrillDownFormID of int
  | TAPLinkedObject of bool
  | TAPLookupFormID of int
  | TAPPasteIsValid of bool

and 'a tprop = 'a tprop * 'a

type 'a tableproperties = 'a tprop list

type 'a tablebody = 'a objproperties * 'a tableproperties * 'a fields * 'a keys * 'a code

// NAV table object
type 'a table = string * string * 'a tablebody

// Properties allowed on a menu item
type 'a mprop =
  | MIPTrig of string * 'a procedurebody
  | MIPCaptionML of string
  | MIPEllipsis of bool
  | MIPEnabled of bool
  | MIPID of int
  | MIPMenuItemType of string
  | MIPMenuLevel of int
  | MIPPushAction of 'a exp
  | MIPRunFormLink of string
  | MIPRunFormLinkType of string
  | MIPRunFormOnRec of bool
  | MIPRunFormView of string
  | MIPRunObject of string
  | MIPShortCutKey of string
  | MIPUpdateOnAction of bool
  | MIPVisible of bool

and 'a mprop = 'a mprop * 'a

type 'a mproperties = 'a mprop list

type 'a menuitem = 'a mproperties

type 'a menuitems = 'a menuitem list

// Properties allowed on a control
type 'a cprop =
  | MIPTrig of string * 'a procedurebody
  | MIPCaptionML of string
  | MIPEnabled of bool
  | MIPID of int
  | MIPMenuitemType of string
  | MIPMenuLevel of int
  | MIPPushAction of 'a exp
  | MIPRunFormLink of string
  | MIPRunFormLinkType of string
  | MIPRunFormOnRec of bool
  | MIPRunFormView of string
  | MIPRunObject of string
  | MIPShortCutKey of string
  | MIPUpdateOnAction of bool
  | MIPVisible of bool

and 'a cprop = 'a cprop * 'a

type 'a cproperties = 'a cprop list

// Properties allowed on a menu item
type 'a menuitem = 'a mproperties

type 'a menuitems = 'a menuitem list

// Properties allowed on a control
type 'a cprop =
331  | COPToolTipML of string
332  | COPTopLineOnly of bool
333  | COPUpdateOnAction of bool
334  | COPValidateTableRelation of bool
335  | COPValuesAllowed of string
336  | COPVertAlign of string
337  | COPVertGlue of string
338  | COPVisible of bool
339  and 'a cprop = 'a cprop ' * 'a
type 'a controlproperties = 'a cprop list
340  type 'a controlheader = int * string * string * int * int * int
type 'a control = 'a controlheader * 'a controlproperties
type 'a controls = 'a control list
341
342 // Properties allowed on a form
type 'a foprop =
  | FOPTrig of string * 'a procedurebody
  | FOPPerm of 'a perm list
  | FOPActiveControlOnOpen of int
343  | FOPAutoPosition of string
344  | FOPAutoSplitKey of bool
345  | FOPBorderStyle of string
346  | FOPCalcFields of string
347  | FOPCaptionBar of string
348  | FOPCaptionML of string
349  | FOPDataCaptionExpr of 'a exp
350  | FOPDataCaptionFields of string
351  | FOPDefaultInsert of bool
352  | FOPDeleteAllowed of bool
353  | FOPEditable of bool
354  | FOPHeight of int
355  | FOPInsertAllowed of bool
356  | FOPLogHeight of int
357  | FOPLogWidth of int
358  | FOPLookupModes of bool
359  | FOPMaximizable of bool
360  | FOPMinimizable of bool
361  | FOPModifyAllowed of bool
362  | FOPMultipleNewLines of bool
363  | FOPPopulateAllFields of bool
364  | FOPSaveControlInfo of bool
365  | FOPSaveFooterSize of bool
366  | FOPSaveTableView of bool
367  | FOPSaveValues of bool
368  | FOPSizeable of bool
369  | FOPSourceTable of int
370  | FOPSourceTablePlacement of string
371  | FOPSourceTableTemporary of bool
372  | FOPSourceTableView of string
373  | FOPTableBoxID of int
374  | FOPTimerInterval of int
375  | FOPUpdateOnActivate of bool
376  | FOPWidth of int
377  and 'a foprop = 'a foprop ' * 'a
378  type 'a formproperties = 'a foprop list
379  type 'a formbody = 'a objproperties ' * 'a formproperties ' * 'a controls ' * 'a code
380  // NAV form object
type 'a form = string ' * string ' * 'a formbody
381
382 // Properties allowed on a request form
type 'a rfprop =
  | RFPTrig of string ' * 'a procedurebody
383  | RFPPerm of 'a perm list
384  | RFPCaptionML of string
385  | RFPHeight of int
386  | RFPLookupModes of bool
387  | RFPWidth of int
388  | RFPSaveValues of bool
389  | RFPSaveControlInfo of bool
390  | RFPSaveFooterSize of bool
391  | RFPSaveTableView of bool
392  | RFPSaveValues of bool
393  | RFPSizeable of bool
394  | RFPSourceTable of int
395  | RFPSourceTablePlacement of string
396  | RFPSourceTableTemporary of bool
397  | RFPSaveControlInfo of bool
398  | RFPSaveFooterSize of bool
399  | RFPSaveTableView of bool
400  | RFPSaveValues of bool
401  and 'a rfprop = 'a rfprop ' * 'a
402  type 'a requestformproperties = 'a rfprop list
403  type 'a requestform = 'a requestformproperties ' * 'a controls
404
405 // Properties allowed on a section control
type 'a scprop =
  | SCPAutoCalcField of bool
406  | SCPAutoFormatExpr of 'a exp
407  | SCPAutoFormatType of int
408  | SCPBackColor of int
409
57
// Properties allowed on a section
type 'a seprop = 'a seprop * 'a

// Properties allowed on a sectioncontrol

type 'a sectioncontrolproperties = 'a scprop list

// Properties allowed on a dataitem
type 'a diprop = 'a diprop * 'a

type 'a dataitemproperties = 'a diprop list

// Properties allowed on a report

type 'a reprop = 'a reprop * 'a

// Properties allowed on a sectioncontrolheader

type 'a sectioncontrolheader = int * string * string * int * int * int

type 'a sectioncontrol = 'a sectioncontrolheader * 'a sectioncontrolproperties

type 'a sectioncontrols = 'a sectioncontrol list

// Properties allowed on a dataitem
type 'a dataitem = 'a dataitemproperties * 'a sections

type 'a dataitems = 'a dataitem list

// Properties allowed on a report
type 'a report = 'a reprop * 'a
REPProcessingOnly of bool
REPShowPrintStatus of bool
REPUseReqForm of bool
REPUseSystemPrinter of bool

and 'a reprop = 'a reprop * 'a

REPProcessingOnly of bool
REPShowPrintStatus of bool
REPUseReqForm of bool
REPUseSystemPrinter of bool

and 'a reprop = 'a reprop * 'a

type 'a reportproperties = 'a reprop list

type 'a report = string * string * 'a reportbody

// NAV report object
type 'a report = string * string * 'a reportbody

type 'a report = string * string * 'a reportbody

let getPos (pos : Lexing.Position) =
    at line " + pos.pos_lnum.ToString() +

let dummy_pos = { new Lexing.Position
    with pos_fname = "< dummy >";
    and pos_lnum = 0;
    and pos_bol = 0;
    and pos_cnum = 0;
}

// C/AL system call keywords

module SysCallKeywords

open Microsoft.FSharp.Collections

(* C/AL system calls (reserved keywords) *)

let Words = [
    "ABS"; "ACTIVATE"; "ACTIVE"; "ADOLINK"; "ADOTEXT";
    "APPLICATIONPATH"; "ARRAYLEN"; "ASCENDING"; "BREAK";
    "CALCDATE"; "CALCFIELD"; "CALCFIELDS"; "CALCSUM";
    "CAPTION"; "CHANGECOMPANY"; "CHECKLICENSEREFILE";
    "CLASS"; "CLEAR";
    "CLEARALL"; "CLEARLASTERRORMARK"; "CLEARKARDS";
    "CLOSE"; "CLOSEDATE";
    "CODECOVERAGELOAD"; "COMPAREIN"; "COMMIT";
    "COMPANYNAME"; "COMPRESARRAY";
    "COLORS"; "CONSISTENT"; "CONTENT"; "CONVERTSTRING";
    "COPY";
    "COPYOBJECT"; "COPYTEXT";
    "COPYFILTER"; "COPYFILTERS"; "COPYLINKS";
    "COPYSTR";
    "COPYSTREAM"; "COUNT"; "COUNTAPPROX"; "CREATE";
    "CREATEDATETIME";
    "CREATEGUID"; "CREATEINSTREAM"; "CREATESOUTSTREAM";
    "CREATETEMPFILE";
    "CREATETOTALS"; "CURRENTDATETIME"; "CURRENTKEY";
    "CURRENTRETRYINDEX";
    "CURRENTTRANSACTIONTYPE"; "DATE2DOW"; "DATETODOW";
    "DATETOTODOW"; "DECIMALPLACEMAX"; "DECIMALPLACEMIN";
    "DELCHR";
    "DELETE"; "DELETEALL"; "DELETELINK"; "DELETELINKS";
    "DELESTR";
    "DDEDATE"; "DOWNLOAD"; "DOWNLOADFROMSTREAM";
    "DTPAGE"; "DT2DATE"; "DT2TIME";
    "DUPLOG"; "DUPPAGE"; "EDITPAGE"; "ENABLE";
    "ENVIRO";
    "EOF"; "ENAE"; "ENBA"; "EVALUATE"; "EXISTS";
    "EXPORT"; "FIELD"; "FIELDACTIVE"; "FIELDCAPTION";
    "FIELDCOUNT";
    "FIELDERROR"; "FILEEXIST"; "FIELDINDEX"; "FILENAME";
    "FILTERGROUP"; "FIND"; "FINDFIRST"; "FINDLAST";
    "FINDESS"; "FORM"; "FORMAT"; "GET"; "GETFILTER";
    "GETFILTERS"; "GETLASTERRORTEXT"; "GETPOSITION";
    "GETRANGECOMMA";
    "GETTEXT"; "GETRECORD"; "GETRESPONSE"; "GETSTAMP";
    "GETSUBTEXT";
    "GETTABLE"; "GETVIEW"; "GLOBALCACHET";
    "GUIALLOWED"; "HASFILTER";
    "HASLINKS"; "HAVEVALUE"; "HEIGHT"; "HIDELINK"; "HIDE";
    "HIDECHR";
    "ISACTION"; "ISAUTOMATION"; "ISBINARY"; "ISBOOLEAN";
    "ISCODE";
    "ISCLEAR"; "ISCODEDESK"; "ISDATE"; "ISDATEFORMULA";
    "ISDECIMAL"; "ISEMPTY"; "ISFILE"; "ISINSTREAM";
    "ISLINKED"; "ISOPTION"; "ISOUTSTREAM"; "ISRECORD";
    "ISVISIBLE";
    "ISTRACK"; "ISTEXT"; "ISTIME"; "ITRANSACTIONTYPE";
    "KEYCODE"; "KEYGROUPDISABLE";
    "KEYGROUPENABLE"; "KEYGROUPENABLED";
    "KEYINDEX"; "LANGUAGE"; "LEM";
    "LENGTH"; "LOCKABLE"; "LOGETIMEOUT"; "LOGHEIGHT";
    "LOOKUPMODE"; "LOOKUPCORE"; "MARK"; "MARKEDONLY";
    "MAINTENANCEDORP"; "MODIFY"; "MODIFYALL";
    "NAME"; "RENAME"; "RENAMEPERMRECORD"; "NEXT"; "NORMALDATE";
    "NUMBER"; "OBJECTID"; "OPEN"; "OPTIONCAPTION";
    "OPTIONSSTRING";
let GetWordsAsHashSet () =
    let set = new HashSet < string >()
    let _ = List . map (function s -> set . Add (s)) Words
    set

B.3 Lexer Definition

// Lexer for NAV objects (C/AL) //

let getLoc lexbuf = Lexing . lexeme_start_p lexbuf
let sysKeyWords = SysCallKeywords . GetWordsAsHashSet ()
let isSysCall s = sysKeyWords . Contains s

let keyword = function

| "AND" -> T_AND
| "ARRAY" -> T_ARRAY
| "BEGIN" -> T_BEGIN
| "BY" -> T_BY
| "CASE" -> T_CASE
| "CODE" -> T_CODE
| "CONTROLS" -> T_CONTROLS
| "DATA" -> T_DATA
| "DATATYPES" -> T_DATATYPES
| "DIV" -> T_DIV
| "DO" -> T_DO
| "DOWNTO" -> T_DOWNTO
| "ELSE" -> T_ELSE
| "END" -> T_END
| "EXIT" -> T_EXIT
| "EVENT" -> T_EVENT
| "FALSE" -> T_FALSE
| "FIELDS" -> T_FIELDS
| "FOR" -> T_FOR
| "IF" -> T_IF
| "IN" -> T_IN
| "KEYS" -> T_KEYS
| "LOCAL" -> T_LOCAL
| "MENUITEMS" -> T_MENUITEMS
| "MOD" -> T_MOD
| "NOT" -> T_NOT
| "OF" -> T_OF
| "OR" -> T_OR
| "PROPERTIES" -> T_PROPERTIES
| "PROCEDURE" -> T_PROCEDURE
| "PROPERTY" -> T_PROPERTY
| "REPEAT" -> T_REPEAT
| "REQUESTFORM" -> T_REQUESTFORM
| "SECTIONS" -> T_SECTIONS
| "TEMPORARY" -> T_TEMPORARY

let B.3 Lexer Definition

// Lexer for NAV objects (C/AL) //

let getLoc lexbuf = Lexing . lexeme_start_p lexbuf
let sysKeyWords = SysCallKeywords . GetWordsAsHashSet ()
let isSysCall s = sysKeyWords . Contains s

let keyword = function

| "AND" -> T_AND
| "ARRAY" -> T_ARRAY
| "BEGIN" -> T_BEGIN
| "BY" -> T_BY
| "CASE" -> T_CASE
| "CODE" -> T_CODE
| "CONTROLS" -> T_CONTROLS
| "DATA" -> T_DATA
| "DATATYPES" -> T_DATATYPES
| "DIV" -> T_DIV
| "DO" -> T_DO
| "DOWNTO" -> T_DOWNTO
| "ELSE" -> T_ELSE
| "END" -> T_END
| "EXIT" -> T_EXIT
| "EVENT" -> T_EVENT
| "FALSE" -> T_FALSE
| "FIELDS" -> T_FIELDS
| "FOR" -> T_FOR
| "IF" -> T_IF
| "IN" -> T_IN
| "KEYS" -> T_KEYS
| "LOCAL" -> T_LOCAL
| "MENUITEMS" -> T_MENUITEMS
| "MOD" -> T_MOD
| "NOT" -> T_NOT
| "OF" -> T_OF
| "OR" -> T_OR
| "PROPERTIES" -> T_PROPERTIES
| "PROCEDURE" -> T_PROCEDURE
| "PROPERTY" -> T_PROPERTY
| "REPEAT" -> T_REPEAT
| "REQUESTFORM" -> T_REQUESTFORM
| "SECTIONS" -> T_SECTIONS
| "TEMPORARY" -> T_TEMPORARY
<table>
<thead>
<tr>
<th>THEN</th>
<th>TO</th>
<th>TRUE</th>
<th>UNTIL</th>
<th>VAR</th>
<th>WHILE</th>
<th>WITH</th>
<th>EVENTS</th>
<th>EDN</th>
</tr>
</thead>
</table>

B.4 Parser Definition

////////////////////////////////////////////////////

// Parser for NAV objects (C/AL)

////////////////////////////////////////////////////

%! #light "off"
% open CALAST
% open PropertiesParser
%


%start NAVObjectList

// Headers
%token <string> T_CODEUNITHEADER
%token <string> T_TABLEHEADER
%token <string> T_FORMHEADER
%token <string> T_FIELDHEADER
%token <string> T_KEYHEADER
%token <string> T_REPORTHEADER

// Constants
%token <string> T_ID
%token <string> T_REAL
%token <int> T_INT
%token <string> T_STRING
%token <string> T_DATE
%token <string> T_TIME
%token <string> T_DATETIME
%token <string> T_SYSCALL

// General object properties
%token <string> T_CAPTIONML
%token <string> T_DATACAPTIONFIELDS
%token <string> T_DECIMALPLACES
%token <string> T_DRILLDOWNFORMID
%token <string> T_FONTNAME
%token <string> T_LOOKUPFORMID
%token <string> T_MAXVALUE
%token <string> T_MINVALUE
%token <string> T_OPTIONCAPTIONML
%token <string> T_OPTIONSTRING
%token <string> T_RUNFORMLINK
%token <string> T_RUNFORMVIEW
%token <string> T_RUNOBJECT
%token <string> T_TABLERELATION
%token <string> T_VALUESALLOWED
%token <string> T_VERSIONLIST

// Field properties
%token <string> T_ALTSEARCHFIELD
%token <string> T_CALCFORMULA
%token <string> T_DESCRIPTION
%token <string> T_INITVALUE
%token <string> T_SUBTYPE

// Key properties
%token <string> T_KEYGROUPS
%token <string> T_SIFTLEVELSTOMAINTAIN
%token <string> T_SQLINDEX
%token <string> T_SUMINDEXFIELDS

// Form properties
%token <string> T_CALCFIELDS
%token <string> T_SOURCETABLEVIEW
%token <string> T_SOURCETABLEVIEW

// Control properties
%token <string> T_BITMAPLIST
%token <string> T_BORDERWIDTH
%token <string> T_NAME
%token <string> T_OPTIONVALUE
%token <string> T_PAGENAMESML
%token <string> T_SUBFORMLINK
%token <string> T_TOOLTIPML

// Menuitem properties
%token <string> T_SHORTCUTKEY

// Report properties
%token <string> T_PAPERSIZE

// DataItem properties
%token <string> T_DATAITEMLINK
%token <string> T_DATAITEMLINKREFERENCE
%token <string> T_DATAITEMTABLEVIEW
%token <string> T_DATAITEMVARNAME
%token <string> T_GROUPTOTALFIELDS
%token <string> T_REQFILTERFIELDS
%token <string> T_REQFILTERHEADML
%token <string> T_REQFILTERHEADML

// Section control properties
%token <string> T_FORMAT
%token <string> T_PADCHAR
// Special tokens
%token T_EOF

// Keyword tokens
%token T_PERMISSIONS T_BEGIN T_END T_VAR
%token T_PROCEDURE T_LOCAL T_EVENT T_IF T_THEN T_ELSE
%token T_CASE T_OF T_WHILE T_DO T_REPEAT T_UNTIL
%token T_FOR T_WITH T_EXIT T_TO T_DOWNTO
%token T_TRUE T_FALSE T_NOT
%token T_MOD T_IN T_TEMPORARY T_WITHEVENTS T_ARRAY
%token T_BY T_DATA T_DATAITEMS T_SECTIONS T_REQUESTFORM

// Operator tokens
%token T_ASSIGN T_NOTEQUAL T_LE T_GE T_DOT T_DOTDOT
%token T_PLUS T_MINUS T_MULT T_DIV T_AND T_COMMA
%token T_SEMICOLON T_LPAR T_RPAR T_LBRACK T_RBRACK
%token T_LCURLY T_RCURLY T_XOR T_EQUAL T_LT T_GT
%token T_COLON T_SCOPE T_OR
%token T_AT T_PLUSEQ T_MINUSEQ T_MULTEQ T_DIVEQ

// Object tokens
%token T_OBJECTPROPERTIES T_PROPERTIES T_CODE T_KEYS
%token T_FIELDS T_CONTROLS T_MENUITEMS

// Precedence and associativity
%left T_DOTDOT
%right T_SEMICOLON
%right T_AT
%left T_PLUS T_MINUS
%left T_MULT T_DIV
%left T_IN T_SCOPE T_DOT
%left T_DO
%left T_AND T_OR T_XOR
%left T_EQUAL T_NOTEQUAL T_LE T_GE

%left T_IF T_THEN T_ELSE
%left T_LBRACK T_RBRACK

%%

// Input should consist of a list of NAV objects
NAVObjectList: { [] } | NAVObject NAVObjectList {$1 :: $2}

// The different kinds of NAV objects (supported for now)
NAVObject: CodeUnit { Codeunit $1 } | Table { Table $1 } | Form { Form $1 } | Report { Report $1 }

// -- NAV Codeunit --
CodeUnit T_CODEUNITHEADER T_LCURLY CodeUnitBody T_RCURLY { let header = ParseRegEx "OBJECT Codeunit ([0-9]+) (.*)" $1 in (List.nth header 0, List.nth header 1, $3) }

CodeUnitBody: ObjectProperties CodeUnitProperties Code ($1,$2,$3)
T_OBJECTPROPERTIES T_LCURLY ObjPropList T_RCURLY { $3 }

ObjPropList:
182 { [] }
| ObjProp T_SEMICOLON ObjPropList { $1 :: $3 }

ObjProp:
187 T_ID T_EQUAL Exp { (ExtractObjectProperty($1,$3), Parsing.symbol_end_pos()) }

ObjPropList:
192 { [] }
| CodeunitProp T_SEMICOLON CodeunitPropList { $1 :: $3 }

CodeunitProp:
197 T_ID T_EQUAL Exp { (ExtractCodeunitProperty($1,$3), Parsing.symbol_end_pos()) }
| Trigger { (CUPTrig(first $1, snd $1), Parsing.symbol_end_pos()) }
| Permissions { (CUPPerm($1), Parsing.symbol_end_pos()) }

Trigger:
202 T_ID T_EQUAL ProcedureBody { ($1,$3) }

Permissions:
207 T_PERMISSIONS T_EQUAL PermListOpt { $3 }
| PermList { [] }
| Perm { [ $1 ] }
| Perm T_COMMA PermList { $1 :: $3 }

Perm:
212 T_ID T_INT T_EQUAL T_ID { ($1,$2,$4) }

Code:
217 T_CODE T_LCURLY VarDeclListOpt ProcedureDeclList EventDeclList T_BEGIN Documentation T_END T_DOT T_RCURLY { ($3,$4,$5,$7) }

VarDeclListOpt:
222 { [] }
| T_VAR VarDeclList { $2 }

VarDeclList:
227 VarDecl { [] }
| VarDecl VarDeclList { $1 :: $2 }

VarDecl:
232 Identifier T_COLON Type T_SEMICOLON { ($1,$3) }

ProcedureDeclList:
237 { [] }
| ProcedureDecl ProcedureDeclList { $1 :: $2 }

ProcedureDecl:
242 ProcedureHeading T_SEMICOLON ProcedureBody T_SEMICOLON { ($1,$3,$5,$7) }

ProcedureHeading:
LocalOpt T_PROCEDURE Identifier T_LPAR FpSectionListOpt T_RPAR ReturnTypeOpt { ($1,$3,$5,$7) }

LocalOpt:
247 { [] }
| T_LOCAL { "LOCAL" }

FpSectionListOpt:
252 { [] }
| FpSectionList { [] }

VarOpt:
257 { [] }
| T_VAR { "VAR" }

65
ReturnTypeOpt:
   { NoReturnType } |
   Identifier Type { NamedReturn($1,$3) } |

ProcedureBody:
   VarDeclListOpt Block { ($1,$2) }

EventDeclList:
   { [] } |
   EventDeclaration EventDeclList { $1 :: $2 }

EventDeclaration:
   EventHeader ProcedureBody T_SEMICOLON { ($1,$2) }

EventHeader:
   T_EVENT Identifier T_SCOPE Identifier T_LPAR FpSectionListOpt T_RPAR T_SEMICOLON { ($2,$4,$6) }

Documentation:
   { "" }

// -- Statement Parser --
Block:
   T_BEGIN Statements T_END { $2 }

// Sequence of statements (only allowed within BEGIN-END, REPEAT-UNTIL and ELSE-branch of CASE statements)
Statements:
   Statement { [$1] } |
   Statement T_SEMICOLON Statements { $1 :: $3 }

Statement:
   { (StmtEmpty, Parsing.symbol_end_pos()) } |
   Block { (StmtBlock($1), Parsing.symbol_end_pos()) } |
   Exp AssignOp Exp { (StmtAssign($1,$2,$3), Parsing.symbol_end_pos()) } |
   Exp { (StmtExp($1), Parsing.symbol_end_pos()) } |
   T_IF Exp T_THEN Statement T_ELSE Statement { (StmtIfThenElse($2,$4,$6), Parsing.symbol_end_pos()) } |
   T_IF Exp T_THEN Statement { (StmtIfThen($2,$4), Parsing.symbol_end_pos()) } |
   T_CASE Exp T_OF CaseList CaseElse T_END { (StmtCase($2,$4,$5), Parsing.symbol_end_pos()) } |
   T_WHILE Exp T_DO Statement { (StmtWhile($2,$4), Parsing.symbol_end_pos()) } |
   T_REPEAT Statements T_UNTIL Exp { (StmtRepeat($2,$4), Parsing.symbol_end_pos()) } |
   T_FOR Exp T_ASSIGN Exp ForOp Exp T_DO Statement { (StmtFor($2,$4,$5,$8), Parsing.symbol_end_pos()) } |
   T_WITH Exp T_DO Statement { (StmtWith($2,$4), Parsing.symbol_end_pos()) } |
   T_EXIT { (StmtReturnNone, Parsing.symbol_end_pos()) } |
   T_EXIT T_LPAR Exp T_RPAR { (StmtReturn($3), Parsing.symbol_end_pos()) }

ForOp:
   T_TO { "TO" } |
   T_DOWNTO { "DOWNTO" }

Case:
   Exp T_COLON Statement { ($1,$3) }

CaseList:
   Case { [$1] } |
   Case T_SEMICOLON CaseList { $1 :: $3 }

CaseElse:
   { (CaseElseEmpty, Parsing.symbol_end_pos()) } |
   T_ELSE Statements { (CaseElse($2), Parsing.symbol_end_pos()) }

AssignOp:
   T_ASSIGN { ":=" } |
   T_PLUSSEQ { "++" } |
   T_MINUSSEQ { "--" } |
   T_MULTEQ { "*=" } |
   T_DIVEQ { "/=" }

// -- Expression Parser --
Exp:
   AtomExp { $1 }
| UnExp { $1 } |
| BinExp { $1 } |
| NaryExp { $1 } |

347 AtomExp:
| Identifier { (Ident($1), Parsing.symbol_end_pos()) } |
| T_STRING { (String($1), Parsing.symbol_end_pos()) } |
| T_REAL { (Real($1), Parsing.symbol_end_pos()) } |
| T_INT { (Int($1), Parsing.symbol_end_pos()) } |
| T_DATE { (Date($1), Parsing.symbol_end_pos()) } |
| T_TIME { (Time($1), Parsing.symbol_end_pos()) } |
| T_DATETIME { (DateTime($1), Parsing.symbol_end_pos()) } |
| T_TRUE { (True, Parsing.symbol_end_pos()) } |
| T_FALSE { (False, Parsing.symbol_end_pos()) } |

357 UnExp:
| T_LPAR Exp T_RPAR { $2 } |
| T_NOT Exp { (Opr(" NOT ", [2]), Parsing.symbol_end_pos()) } |
| T_PLUS Exp { (Opr(" + ", [2]), Parsing.symbol_end_pos()) } |
| T_MINUS Exp { (Opr(" - ", [2]), Parsing.symbol_end_pos()) } |

362 BinExp:
| Exp T_PLUS Exp { (Opr(" + ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_MINUS Exp { (Opr(" - ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_MULT Exp { (Opr(" * ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_DIV Exp { (Opr(" / ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_MOD Exp { (Opr(" MOD ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_AND Exp { (Opr(" AND ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_OR Exp { (Opr(" OR ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_XOR Exp { (Opr(" XOR ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_LT Exp { (Opr(" < ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_GT Exp { (Opr(" > ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_EQUAL Exp { (Opr(" = ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_LE Exp { (Opr(" <= ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_GE Exp { (Opr(" >= ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_NOTEQUAL Exp { (Opr(" < > ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_SCOPE Exp { (Opr(":: ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_DOT Exp { (Opr(" . ", [1; 3]), Parsing.symbol_end_pos()) } |
| Exp T_DOTDOT Exp { (Opr(" .. ", [1; 3]), Parsing.symbol_end_pos()) } |

372 NaryExp:
| Identifier T_LPAR ExpListOpt T_RPAR { (Call($1, $3), Parsing.symbol_end_pos()) } |
| Exp T_LBRACK ExpList T_RBRACK { (Opr(" arrref ", $1 :: $3), Parsing.symbol_end_pos()) } |
| Exp T_IN T_LBRACK ExpList T_RBRACK { (Opr(" IN ", $1 :: $4), Parsing.symbol_end_pos()) } |

382 NaryExp:
| Exp T_LPAR ExpListOpt T_RPAR { (Call($1, $3), Parsing.symbol_end_pos()) } |
| Exp T_LBRACK ExpList T_RBRACK { (Opr(" arrref ", $1 :: $3), Parsing.symbol_end_pos()) } |
| Exp T_IN T_LBRACK ExpList T_RBRACK { (Opr(" IN ", $1 :: $4), Parsing.symbol_end_pos()) } |

392 List:
| ( ] ) |
| ExpList { $1 } |

397 | Exp T_COMMA ExpList ($1 :: $3) |

// -- Identifier Parser --
Identifier:
402 T_ID { (Id($1), Parsing.symbol_end_pos()) } |
| T_ID IdIndexNumber { (IdOnlyIndex($2), Parsing.symbol_end_pos()) } |
| Identifier T_AT IdIndexNumber { (IdentOnlyIndex($2), Parsing.symbol_end_pos()) } |
| T_SYSCALL { (SysCall($1), Parsing.symbol_end_pos()) } |

407 IdentifierList:
| ( ] ) |
| Identifier T_COMMA IdentifierList { $1 :: $3 } |

412 IdIndexNumber:
| T_INT { (IndexPos ($1), Parsing.symbol_end_pos()) } |
| T_MINUS T_INT { (IndexNeg ($2), Parsing.symbol_end_pos()) } |

417 // -- Type Parser --
Type:
SimpleType { $1 } |
| SizedType { $1 } |
| NumberedType { $1 } |
| ParamType { $1 } |
| ConstructedType { $1 } |
| TextConstType { $1 } |

SimpleType:
427  T_ID { ((match $1 with
    | "Action" -> TypeAction
    | "Date" -> TypeDateTime
    | "Time" -> TypeTime
    | "Integer" -> TypeInteger
    | "Boolean" -> TypeBoolean
    | "DateTime" -> TypeDateTime
    | "Decimal" -> TypeDecimal
    | "RecordRef" -> TypeRecordRef
    | "GUID" -> TypeGUID
    | "Dialog" -> TypeDialog
    | "Char" -> TypeChar
    | "DateFormula" -> TypeDateFormula
    | "Option" -> TypeOption
    | "Variant" -> TypeVariant
    | "InStream" -> TypeInStream
    | "OutStream" -> TypeOutStream
    | "FieldRef" -> TypeFieldRef
    | "KeyRef" -> TypeKeyRef
    | "File" -> TypeFile
    | "RecordID" -> TypeRecordID
    | "BigInteger" -> TypeBigInteger
    | "Duration" -> TypeDuration
    | _ -> failwith "Parser : SimpleType - This should not happen!")
}), Parsing . symbol_end_pos()) }

452  SizedType:
  T_ID T_LBRACK T_INT T_RBRACK { ((match $1 with
    | "Code" -> TypeCode $3
    | "Text" -> TypeText $3
    | _ -> failwith "Parser : SizedType - This should not happen"
  ), Parsing . symbol_end_pos()) }

467  NumberedType:
  T_ID T_INT { ((match $1 with
    | "Record" -> TypeRecord $2
    | "Codeunit" -> TypeCodeunit $2
    | "Form" -> TypeForm $2
    | "Report" -> TypeReport $2
    | "XMLport" -> TypeXMLport $2
    | _ -> failwith "Parser: NumberedType - This should not happen!")
    Parsing . symbol_end_pos()) }

487  ParamType:
  T_ID Identifier T_WITHEVENTS { ((match $1 with
    | "Automation" -> TypeAutomationEvents $2
    | _ -> failwith "Parser: ParamType - This should not happen"
  ), Parsing . symbol_end_pos()) }

487  ConstructedType:
  T_ARRAY T_LBRACK ArraySize T_RBRACK T_OF Type { (TypeArray ($3 ,$6),
    Parsing . symbol_end_pos()) }

492  TextConstType:
  T_ID T_STRING { ((match $1 with
    | "TextConst" -> TypeTextConst $2
    | _ -> failwith "Parser: TextConstType - This should not happen"
  ), Parsing . symbol_end_pos()) }

497  ArraySize:
  T_INT { [$1 ]
  | T_INT T_COMMA ArraySize { $1 :: $3 }

502  // -- NAV Table --
  Table:
  T_TABLEHEADER T_LCURLY TableBody T_RCURLY { let header = ParseRegex "OBJECT Table ([0-9]+)
  (\.* )" $1 in
    (List.mth header 0, List.mth header 1, $3) }
let header = ParseRegex "OBJECT Form ([0-9]+)\n(\.+)" $1 In
(List.nth header 0, List.nth header 1, $3) }

FormBody:
ObjectProperties FormProperties Controls Code ($1,$2,$3,$4)

FormProperties:
T_PROPERTIES T_LCURLY FormPropList T_RCURLY ($3)

FormPropList:
{ [] }

| FormProp T_SEMICOLON FormPropList ($1 :: $3)

FormProp:
FormProperty ($1, Parsing.symbol_end_pos())
| Permissions (FDPMem($1), Parsing.symbol_end_pos())

FormProperty:
T_ID T_EQUAL Exp (ExtractFormProperty($1,$3))
| T_CALECFIELDS (ParseFDPCalcFields $1)
| T_PUTCALCFIELDS (ParseFDPCalcFields $1)
| T_SOUNCABLEVIEW (ParseFDPSourceTableView $1)

Controls:
T_CONTROLS T_LCURLY ControlsList T_RCURLY ($3)

ControlsList:
{ [] }

| Control ControlsList ($1 :: $2)

Control:
ControlHeader T_RCURLY ($1,[[]])
| ControlHeader T_SEMICOLON ControlPropList T_RCURLY ($1,$3)

ControlHeader:
T_LCURLY T_INT T_SEMICOLON T_ID T_SEMICOLON XPos T_SEMICOLON T_INT T_SEMICOLON T_INT T_SEMICOLON T_INT T_SEMICOLON T_INT { ($2 ,$4 ,$6 ,$8 ,$10 ,$12) }

XPos:
T_INT ($1.ToString())

| T_ID ($1) // "infinite"

ControlPropList:
{ [] }

| ControlProp ($1)
| ControlProp T_SEMICOLON T_ID T_SEMICOLON ControlPropList ($1,$3)

ControlProp:
ControlProperty ($1, Parsing.symbol_end_pos())
| ControlProperty ($1, Parsing.symbol_end_pos())
| MenuItems ($1, Parsing.symbol_end_pos())

MenuItems:
T_ID T_EQUAL T_MENUITEMS T_LCURLY MenuItemsList T_RCURLY (COPMenuItems($1,$5))

MenuItemsList:

MenuProp { [1] } |

MenuProp:
  MenuProp { [1] } |

MenuProp:
  MenuProperty { ($1, Parsing.symbol_end_pos()) } |
  Trigger { (MIPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }

MenuProperty:
  T_ID T_EQUAL Exp { ExtractMenuProperty($1,$3) } |
  T_CAPTIONML { ParseMIPCaptionML $1 }
  T_RUNFORMVIEW { ParseMIPRunFormView $1 }
  T_RUNOBJECT { ParseMIPRunObject $1 }
  T_SHORTCUTKEY { ParseMIPShortCutKey $1 }

Report:
  T_REPORTHEADER T_LCURLY ReportBody T_RCURLY { |
    ReportBody { ObjectProperties ReportProperties DataItems RequestForm Code { ($1, $2, $3, $4, $5) }
    ReportProperties:
      T_PROPERTIES T_LCURLY ReportPropList T_RCURLY { } |
    ReportProp:
      ReportProperty { ($1, Parsing.symbol_end_pos()) } |
      Trigger { (REPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
      Permissions { (REPPerm($1), Parsing.symbol_end_pos()) }
    ReportProperty:
      T_ID T_EQUAL Exp { ExtractReportProperty($1,$3) } |
      T_PAPERSIZE { ParseREPPaperSize $1 }
    DataItems:
      T_DATAITEMS T_LCURLY DataItemList T_RCURLY { } |
    DataItem:
      T_LCURLY DataItemProperties Sections T_RCURLY { } |
      DataItemProperty:
        DataItemProperty { ($1, Parsing.symbol_end_pos()) } |
        Trigger { (DIPTrig(fst $1, snd $1), Parsing.symbol_end_pos()) }
        T_CALCFIELDS { ParseDIPCalcFields $1 }
        T_DATAITEMLINKREFERENCE { ParseDIPDataItemLinkReference $1 }
        T_DATAITEMTABLEVIEW { ParseDIPDataItemTableView $1 }
        T_DATAITEMVARNAME { ParseDIPDataItemVarName $1 }
        T_REQFILTERFIELDS { ParseDIPReqFilterFields $1 }
        T_REQFILTERHEADINGML { ParseDIPReqFilterHeadingML $1 }
        T_TOTALFIELDS { ParseDIPTotalFields $1 }
    Sections: