Building an Integrated Persistent Application

Dag Sjøberg
Malcolm Atkinson
João Lopes
Phil Trinder
Computing Science Department, University of Glasgow
Glasgow, Scotland

{sjoberg,mpa,jlopes,trinder}@dcs.glasgow.ac.uk

Abstract

The major motivation for database programming language (DBPL) research is to facilitate the construction and maintenance of large data-intensive applications. To fully benefit from DBPLs, supporting methodologies and tools are needed. This paper reports requirements for such methodologies and tools that were experienced when constructing a multi-authored, multi-levelled Thesaurus Application (TA) in a higher order DBPL. Although built in a specific language (Napier88), the major principles discovered apply to other DBPLs.

The TA comprises several loosely-integrated components constructed by different programmers. The components were themselves implemented using general purpose sub-components, including libraries. We experienced that a realistic application could be constructed quickly in a DBPL. Rapid construction was facilitated by the use of libraries, code reuse and an incremental construction methodology supported by the persistent store. Language features such as a polymorphic type system and structural type equivalence were important. Nevertheless, some problems were encountered with code reuse, with integrating independently constructed components, with the lack of concurrency and with build management such as installation and recompilation. Our experiences lead us to suggest several improvements, including models, methodologies and supporting tools for persistent application construction and maintenance.

1 Introduction

The major motivation for database programming language (DBPL) research is the belief that such languages will facilitate the construction of large, long-lived data-intensive applications. Most database programming language research has been directed towards constructing new languages and object stores. There are relatively few reports of experience using these facilities to construct and maintain persistent applications. Non-trivial DBPL applications are also rare because many languages and stores are experimental prototypes. However, Napier88 [MBCD89] is a new persistent programming language (PPL) that is

sufficiently well-engineered to support the construction of non-trivial applications.

The following sections describe experience gained in constructing the Thesaurus Application (TA) in Napier88. Although built in a specific language, many of the principles discovered apply to any DBPL where code is accumulated in a single repository, for example object-oriented DBPLs. One impact of a single repository of code is that programs are no longer constructed as large discrete units. Instead, a typical program is a small unit that retrieves and reuses code already in the repository.

Although small, around 14000 lines of code, TA has an interesting external and internal architecture. The first external feature is that, like many much larger applications, it was *multi-authored*. That is, each of the three components was constructed by a different programmer. Secondly, the application is not a close-knit product, such as might be delivered to a "user". Instead the architecture is more open, or extensible. Indeed it appears that, in a persistent world, programs from several sources are accreted to manipulate and interrogate the thesauri.

The TA internal architecture is *multi-level*, i.e., the three components are implemented using several general-purpose sub-components, namely the WIN and Maps libraries [CDK90, ALPR91] and the Ringad comprehension translator [ATW93]. Some sub-components are used in more than one component; Section 3 describes the architecture in detail. Moreover, some subsidiary components were constructed by non-TA authors, and some originate from a remote site. A second implementation feature is the degree of reuse in the internal architecture. Both the Thesaurus-based Software Information Tool (TSIT) and the Ringad comprehension translator reuse parts of the St Andrews Napier88 compiler.

The TA construction started using the tools and methodologies available in 1990. Because DBPL research is advancing, better tools and methodologies are now available. We are, however, constrained to report our experiences with the tools and methodologies used during the development.

We experienced that a realistic, i.e., multi-author and multi-level, application could be constructed quickly in a DBPL. All three programmers, although experienced in other languages, were novice Napier88 programmers. Despite our inexperience, TA was constructed in less than eight person-months work. The rapid construction was facilitated by the use of libraries, code reuse and an incremental construction methodology.

As detailed in Section 4, the TA project benefited from the powerful type system of Napier88. Useful libraries existed and proved easy to use. Reuse of code, even foreign-site code, was possible, although some difficulties were encountered. In particular, locating all of the source code of a sub-component is hard. Separately developed sub-components can be integrated by inserting them into the same store, but discovering an *installation-order*, i.e., a legal order for installing sub-components, might be very difficult. Lack of concurrency[1] complicated both multiple-author cooperation and software maintenance.

As described in Section 5, our experiences lead us to propose several improvements. Many of these issues are even now being addressed by the Napier88 community. A well-developed methodology would facilitate construction and

[1] The latest versions of Napier88 have multi-threading.

maintenance of large applications. Several tools would also have been useful, particularly tools that provide cross-referencing and consistency checking, determine installation-order and perform partial recompilation when a small part of an application changes.

The remainder of the paper is structured as follows. Section 2 describes the external TA structure. Section 3 describes the internal implementation structure. Section 4 is a discussion of the experience we gained from writing the application. Section 5 describes the improvements we recommend. Section 6 concludes.

1.1 Enabling Technology

Napier88 is a persistent programming language, that is, it conforms to the principles of orthogonal persistence [ABC+83]. Napier88 is a strongly typed language with some type inference. It provides labelled Cartesian product (structures), labelled disjoint sums (variants) and explicit parametric polymorphism. Existential polymorphism is used to implement ADTs. Napier88 is a store-based language that combines persistence, higher-order procedures [AM85] and L-value and R-value binding [MBDA90].

During the TA development we adhered to a programming methodology based on L-values bound to named persistent locations [Dea87, Mem90, Con91, DCC92]. A stub for a procedure or another kind of value is initially inserted into a new location to which other programs can then bind. For each stub a template program is created that inserts a meaningful value into the location. Incremental development is supported in that the template program can be edited and the location correspondingly updated with a new value without the need for editing, recompilation or re-execution of the other programs using the value.

Currently most Napier88 applications are components of what may be called a Napier88 programming environment which includes a callable compiler [Cut93a], a window manager (WIN), browsers [Kir93, FDK+92], a hyper-programming environment [KCC+92], a Maps library and both model and schema editors [Zhe92]. TA enhances the Napier88 programming environment even further, and this paper demonstrates how TA was built by *using* the Maps and WIN libraries, *reusing* components of a Napier88 compiler and browser and *integrating* existing components (TSIT, ShTh, Utility Queries).

2 External Thesaurus Application Structure

The Thesaurus Application (TA) assists persistent programmers in keeping track of the structure of the programs and other data in the persistent store. It is a meta-application in that its universe of discourse is applications themselves. TA provides answers to questions like the following: Which environments, types, procedures, etc. exist? In which programs or environments are they defined or used? Which places may be affected by changes to them? etc. The need for such a tool has often been experienced by persistent programmers.

The fundamental component of TA is the collection of thesauri (Figure 1). For each application registered with TSIT there is an associated thesaurus holding information about names in the source programs and names denoting

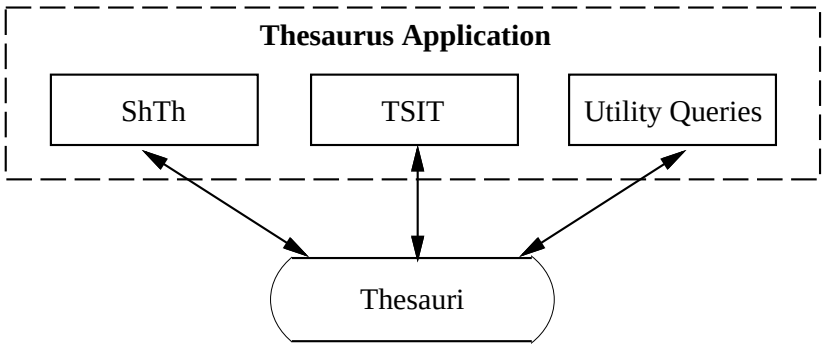

Figure 1: TA structure

name-type-value-constancy bindings in a persistent store [ABC+83, MBDA88]. The thesaurus is automatically updated by TSIT at times specified by the user, for example daily at 02:00. A thesaurus update can also be initiated at any time.

As the interface to the thesaurus and the query facilities provided by TSIT are rather primitive, a need was felt for a more convenient window-based interface with enhanced query possibilities. The ShTh component was then developed by using WIN and provides a graphical interface to one or more thesauri. ShTh also includes a simple query language for operations on the thesauri. Complex queries (involving recursion), however, cannot be expressed. To meet this deficiency another software component, the comprehension query language, was constructed. Queries and their results can be saved in the persistent store.

2.1 A Thesaurus-based Software Information Tool

The heart of the Thesaurus-based Software Information Tool (TSIT) is the thesaurus which keeps track of identifiers of all kinds (denoting types, structures, (polymorphic) procedures, ADTs, etc.).

Names are the focus of attention of the thesaurus since they are central to programmers and system builders' thinking and thus influence the way software is organised. The meanings attached to names are relatively stable when dealing with concepts at an abstract level — even though the detailed semantics and interpretation may vary between people and between contexts. This contrasts with all changes in physical software implementations [Sjø93a]. The most important information held by a thesaurus entry is as follows:

- name (a textual form of an identifier in a source program or of a binding in a persistent store)
- container (file or environment)
- line number (if container is a file)
- kind — base type (integer, real, string, etc.) or constructed type (structure, procedure, etc.)

- constancy (constant or variable)
- usage (informs whether the name occurs as a declaration or use of a type identifier, or as a declaration, left context or right context of a value identifier)
- context (declaring use of binding in the store, declaring type parameter, procedure parameter, structure field, variant tag, etc., inserting into or dropping binding from persistent store, dereferencing structure field, projecting variant tag, etc.)

There are two categories of thesauri:

1. The *master-thesauri* contain automatically generated data reflecting the state of the source code and persistent store of an application at the time of the analysis.
2. The *derived-thesauri* are generated from the master-thesauri via queries.

TSIT is the tool responsible for creation and update of the master-thesauri. Initially one thesaurus is created for each application registered with TSIT. A thesaurus is populated by an analyser component of TSIT that scans all the source files and the persistent store of the application being analysed. Each time a name occurrence is encountered associated information is stored in the thesaurus. To ensure correctness and consistency of the master-thesauri, entries cannot be inserted, modified or removed interactively or by any program that is not part of TSIT. A more detailed description can be found in [Sjø92].

TSIT provides primitive search facilities and simple consistency checks via a textual interface. A thesaurus created by ShTh or Utility Queries is typically a result of a query on a master-thesauri. Such derived-thesauri may, in turn, be the subject of new queries.

2.2 Show Thesaurus User Interface

The ShTh user interface was implemented in order to provide an easy way of filtering the (potentially) huge amount of information contained in a thesaurus. ShTh provides menu-driven facilities that help the user to query a thesaurus and visualise the result of query application. It also provides facilities to store and retrieve queries and thesauri; thesauri may also be imported or exported via text files.

Figure 2 presents the ShTh interface; the top menu line gives a broad idea of its functionality. In the "Actions" menu the user has options to load, close, save, save as, delete, revert, and run queries; it also has undo and quit. The "Select", "Project" and "Sort" menus are used to build a query. Queries and thesauri may be stored, retrieved and visualised using this interface. In the same figure, part of the display of a thesaurus can be seen. The user can also perform a union of two thesauri or renumber a thesaurus to compact the sequential key.

By using pull-down menus from the menu line, the user can incrementally build a query to be run against the current thesaurus. Figure 2 shows an example of a query and the corresponding results. The notation used to visualise the

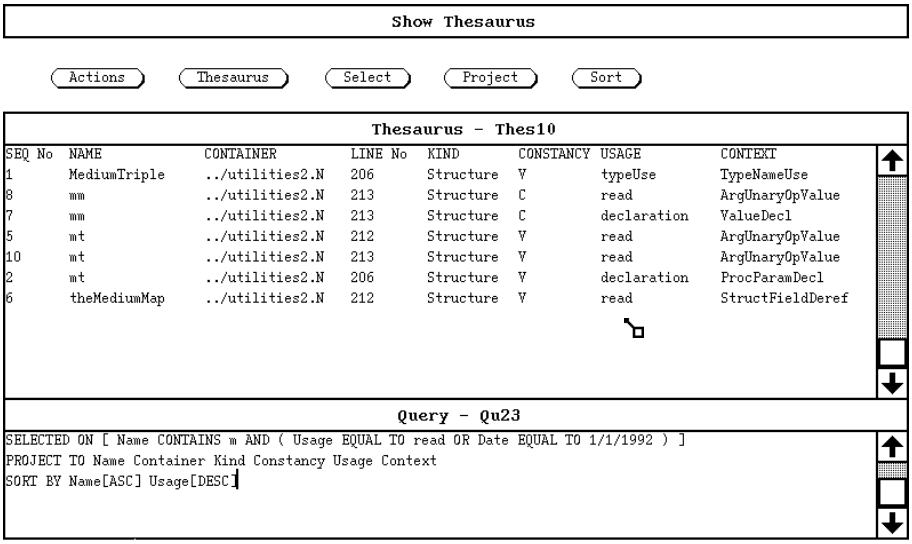

Figure 2: ShTh Interface

query is a subset[2] of ASTRID[3]. The query consists of a selection, a projection over domains and a display order specified by a nested sort clause.

2.3 Utility Queries

Typical data-intensive applications often use a powerful, and usually embedded, query language for three reasons. First, although interactive query languages, like those provided by TSIT and ShTh, may be used by naïve users, they lack the computational power to express some useful queries. A TA query that requires a powerful query language is to generate a call-graph or procedure explosion. An explosion discovers all of the procedures that are called by a given procedure, and all of the procedures they in turn call. An explosion can be used to split off a subsystem within a larger system. Another TA query requiring power implodes a procedure to find all procedures that call it, and any procedure that calls the caller.

The second reason for using a powerful, non-interactive query language is for "canned" queries. These are queries or reports that are run regularly (end-of-day, end-of-month, etc.). Primarily to avoid errors such queries are stored, i.e. canned. Storing a query may also aid efficiency and ensure that the information is always provided in the same format. A typical TA canned query is to find type or value identifiers defined but not used. Third, many TA users will have a high degree of computing skill and be able to use a powerful query notation themselves to extract information of interest about their programs. Incidentally, for any naïve users, the utility queries could easily be packaged into a menu.

[2]Note that there is no join as there is only one thesaurus at a time.
[3]ASTRID is a generalised relational algebra [Gra84].

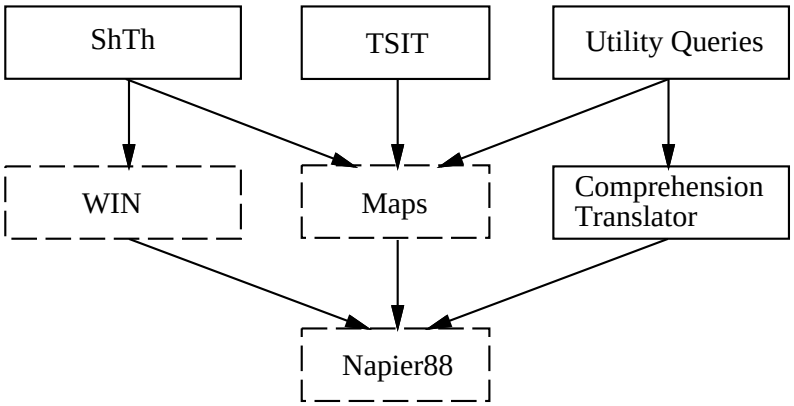

Figure 3: Implementation levels

3 Internal Thesaurus Application Structure

Figure 3 presents the implementation levels of the TA application. Each component usage is represented by an arrow; the solid boxes indicate work done by the TA authors. The authors were using libraries and other software components that they had not written themselves: the WIN Library and the Napier88 compiler were developed in St Andrews; the Maps library in Glasgow.

The first level comprises the three components already described: TSIT, ShTh and Utility Queries. The second level has add-on libraries of Napier88 procedures implementing the WIN windowing system, the Maps bulk data types and the translator from comprehension programs to Napier88 programs. All these components were entirely written in Napier88.

Table 1 illustrates the size of the (sub-)components in terms of lines of code and number of name occurrences. The table also shows the distribution of value identifiers occurring in declarations, right contexts and left contexts, respectively. The measurements were collected by TSIT itself [Sjø92].

(Sub-)component	*Lines of code*	*Name occ.*	*% Declaration*	*% R-Context*	*% L-Context*
ShTh	4986	9934	26	67	7
TSIT	8356	14626	31	64	5
Utility Queries	360	943	11	88	1
Total Components	13702	25503	23	73	4
WIN	24053	37546	28	68	4
Maps	4844	9479	32	63	5
Compr. Translator	1018	1555	22	69	9
Total Sub-comp.	29915	48580	27	67	6
Total TA	43617	74083	25	70	5

Table 1: Measurements of the (sub-)components

3.1 Components

Utility queries are typically small programs written to extract specific information from the thesaurus. Queries are written in Napier88 with comprehensions. Although "Utility Queries" is currently the smallest component, additional queries can be written.

The Shth interface was built in Napier88 and by using the Maps and WIN libraries. Napier88 provides linguistic support for two graphical types: *pic* and *image*. Using pic, pictures are constructed as line drawings in two dimensions of an infinite real space. Pictures may be combined with the operations of join, concatenate, shift, scale, rotate, colour and text [MBDA86]. Images are simply arrays of pixels. Maps were used to provide the storage and retrieval facilities, sets, indexes and stored finite partial functions. WIN provided the user interface building blocks.

The component of TSIT that processes Napier88 source programs is based on the Napier88-in-Napier88 (NinN) compiler developed at St Andrews [Cut93a]. The lexical and syntax analyser of the compiler have been adjusted to conform to special information needs of TSIT. Instead of generating code, the TSIT analyser extracts various information during the analysis and inserts it into the thesaurus.

The thesaurus also contains information about the contents of the persistent store. The code for scanning the store was implemented by reusing low-level procedures used in the implementation of the Napier88 browser [KD90]. These procedures are not type-safe and are thus not available in standard Napier88. Instead of using the low-level procedures reflection could have been used, but that alternative might have impaired the performance [Kir93]. The Maps library is heavily used in the implementation of TSIT (Figure 3). In particular, each thesaurus is stored as a map.

3.2 Sub-Components

WIN

WIN is a library of procedures providing a set of generators of the most common user interface objects: windows, editors, dialogue boxes, window managers, icon managers, menus, light buttons and check boxes.

In WIN actions take place after the occurrence of some *event*. Events are generated by the mouse or the keyboard. An interactive program is built upon a number of sub-programs that perform an action in response to some event. These programs are *applications* which are packed together in a *notifier*'s list. The application part of notifiers (*distributeEvent*) can also be an application inside other notifiers. This allows hierarchies to be built as multi-level notifiers. WIN also allows dynamic reconfiguration of notifiers by adding or removing applications from a notifier list. An *event monitor* will send events to the top most notifier which will route that event through the notifier's list of applications to the first application that accepts it. Each window has encapsulated in it an application that processes input events received by the window and an image on which raster operations are performed.

Maps Bulk Data Types

Maps constitute an add-on bulk data type language implemented as a library of polymorphic Napier88 procedures [ALPR91]. Formally, maps are extensional functions from a domain of type A to a range of type Z, with A and Z being any Napier88 data type. Values of this type constructor denote a stored finite mutable function and may be considered as a set of tuples. The operations provided include iterations (e.g. *for each*); insert, find and remove entries; copy, union, intersection, difference, filter and size, among others. A map may be arbitrarily large, and in TA the thesauri are maps.

Ringad Comprehension Translator

Both WIN and Maps are libraries of procedures. In contrast, the comprehension translator is a language processor. It is a preprocessor that translates Napier88 programs with embedded Ringad comprehensions into standard Napier88. A comprehension is translated into several recursive polymorphic functions. The translator reuses the lexical analyser, error handling and utility components of the St Andrews Napier88 compiler [Cut93a].

Ringad comprehensions are a general purpose query language. In particular they can be defined over several different bulk types, e.g. maps, lists, ordered sets and vectors in Napier88. Comprehension queries are both powerful and easily optimised [Tri91]. Because the utility queries access thesauri, which are maps, they use procedures out of the Maps library.

4 Construction and Cooperation Experience

4.1 Napier88 Language

Almost all DBPLs are experimental languages and hence far from robust. In contrast, Napier88 is well engineered. Moreover, the polymorphic type system provided by Napier88 was crucial to the implementation of the comprehension translator. A generic translator that permits queries over lists, maps, vectors and sets could not have been constructed in a language without polymorphism, e.g. PS-algol [ACC82], an early persistent language. However, if Napier88 had supported bounded parametric polymorphism, the translator would have been further simplified. In summary, the most important observation we make is that very few other DBPLs are sufficiently robust and simultaneously possess a sufficiently rich type system to permit the construction of a complex application like TA. Built-in support of graphical types and first class procedures packaged together with the related objects and parameterised types proved useful in the ShTh implementation.

4.2 Napier88 Libraries

Because Napier88 is a robust and stable language platform, useful libraries of software have been constructed, and more are under construction [ABJP93]. Again this contrasts favourable with many other DBPLs. The Maps and standard libraries are used by all of the TA sub-components; the WIN library is

used by ShTh. These libraries proved very useful, and due to persistence accessing them was straightforward. More complete documentation of some of these libraries would have made them easier to use.

4.3 Reuse

In a large-scale software development project one should aim at software reuse [Kru92]. The success of reuse depends heavily on the availability and quality of the information about the existing software.

As described in Section 3.1, the TSIT analyser is a based on a modified version of the NinN compiler developed at St Andrews [Cut93a]. Similarly, the comprehension translator reuses the lexical analysis, printing and some utility components of NinN.

The code for extracting information from the persistent store into the thesaurus was implemented by directly reusing low-level procedures used in the implementation of the Napier88 browser. Implementing the code proved easy due to good documentation [KD90].

Reusing the syntax analyser was not straightforward. The compiler is one-pass, i.e., the parsing and code generation are inter-twined which means that detecting all program parts concerned with code generation is difficult. The documentation and some structuring principles alleviated the problem but were not sufficient for easy modification of the software to the needs of TSIT. In spite of this problem we heavily benefited from reusing the compiler components — developing TSIT would have been very much harder without reusing NinN. In order to simplify reuse, work is in progress to identify substitutable generation interfaces [Cut93b].

When installing bindings, including environments and procedures, into a persistent store, the installation-order is significant. That is, a binding must be inserted before it can be used. For example, an environment must be created before it can be populated, a location must be created for a procedure before its L-value can be updated, etc. It was a hard task to install the modified compiler components from St Andrews into the store of the TSIT author. The original Unix Makefiles [Fel79] could not be used since they had not been updated in accordance with the changes to the Napier88 code. Since the author did not fully understand the structure of the compiler components and their interaction with the persistent store, the installation-order was determined by trial-and-error.

Whereas TSIT uses most of the NinN compiler, the comprehension translator uses primarily the lexical analyser. Discovering which environments and procedures are required by the lexical analyser is difficult. The difficulty arises because the lexical analyser code extracts environments containing procedures and other values. Discovering the name of the program that inserted those values may also be hard, but is necessary in order to locate the source code. The problem is exacerbated in an application with several levels of directory structure.

4.4 Integration

The code necessary for integrating ShTh and Utility Queries with TSIT (including the master-thesauri) is quite simple. There is one environment for each of

the three components. All procedures and data structures local to these applications are contained in their respective environments. That is, the persistence paradigm enhances extensibility and integration.

The collection of type definitions in a Napier88 application approximately corresponds to the schema in a conventional database. The source of such type definitions may be contained in *type-programs* which consist exclusively of type definitions; the compiled form is stored in persistent environments. A few cases of naming conflicts were experienced when the type definitions of the sub-components were integrated. In addition to resolution of naming conflicts, integrating persistent software components also requires removal of duplicates, determination of dependencies among type definitions, etc. This is equivalent to the problem of schema integration in conventional databases [BLN86].

4.5 Multiple Authors — Cooperation

A severe problem of Napier88 (in use at the time of this experiment) is that it does not provide concurrent and distributed access to a persistent store implying that the software components had to be developed separately in the private store of each author. Eventually the components were integrated by re-installing ShTh and Utility Queries in the store of the thesaurus and TSIT tool. However, maintenance is complicated as long as no concurrency is provided. Another consequence of the lack of concurrency is that libraries have to be installed into each of the private stores of the users, raising well-known problems with keeping multiple copies of software consistent. A partial solution to this problem can be envisaged by using Munro's store-to-store copying facilities [Mun93].

5 Proposed Improvements

To simplify and improve large-scale application development, which typically involves integrating separately developed components, comprehensive construction and maintenance methodologies are needed. Appropriate supporting tools are also essential. From, *inter alia*, our experience with the TA project they should provide the following functionality:

Cross-referencing Information about where and how things are defined and used should be provided. This includes for example finding the corresponding source file of a procedure in the persistent store.

Recompilation and re-execution In addition to recompiling every changed program there are some special cases to be considered. If a type definition is changed, the dependent programs need to be recompiled. Moreover, if there are dependencies between several type-programs, a compilation order must be determined. After program modifications programs that change the contents of the persistent store (e.g. updating a procedure value) should be re-executed.

Installation Determining the installation-order for programs that insert bindings into the persistent store may be a non-trivial task for medium or large applications and should therefore be automated.

Consistency checking Constraints like the following should be checked: "all declared bindings in a program should be used within that program," "a binding inserted into the store, not intended for export, should be used somewhere within the application," "there should be exactly one program updating a procedure (or another kind of value) bound to a persistent location initialised with a stub (cf. methodology of Section 1.1)," etc. Such constraints are part of a proposed model for persistent application systems [Sjø93b].

Indeed our experiences of constructing the TA application have confirmed the need for TA itself — one of the purposes of building TA was to provide a foundation for tools that tackle the deficiencies described in the previous section. TA keeps track of all the names denoting identifiers in source code or bindings in the persistent store of an application. Extensive cross-referencing is also supported.

At present, many Napier88 programmers use Make [Fel79] to help rebuild the application after change. The programmers, however, have to manually specify compilation and execution dependencies. Similarly, Make and sometimes Unix shell-scripts are used to install software into the persistent store. A correct installation-order, however, has to be determined and typed in manually into a Makefile or a script.These problems are addressed by EnvMake [Sjø93b] — another thesaurus-based tool that automatically infers the necessary dependencies from the thesaurus and initiates (re)compilation and (re-)execution. If installation is requested, EnvMake installs components in correct order (if such an order exists[4]) into the persistent store. EnvMake thus relieves the programmers from the burden of maintaining Makefiles and scripts.

In addition to replacing the use of Make, EnvMake provides additional functionality tailored for persistent application development such as checking a whole range of constraints that the application should adhere to. Among other things, inconsistencies between the programs and data in the persistent store of an application will be detected.

6 Conclusions

At first sight application development appears more complicated in a database programming language context than in a traditional context. The reason is that issues that earlier were dealt with by the operating system or DBMS, and not made explicit, are now dealt with within the programming language itself.

The main purpose of DBPLs is to facilitate the construction of large long-lived data-intensive applications. A conclusion from the TA project is that realistic applications can be constructed quickly in a DBPL. (The three TA programmers, all inexperienced in Napier88, used only eight person-months in total to develop TA.) Our experiences fall into two categories: those specific to Napier88 and those relevant to any DBPL with a code repository.

Several of the issues specific to Napier88 are known and are being addressed. As there were several authors, we felt the lack of concurrency [Mun93]. Bounded parametric polymorphism would have simplified the implementation of the Ringad comprehension translator.

[4] If components cyclically refer to each other, installation is impossible.

Several issues arise in any DBPL where code resides in a repository. In a typical object-oriented DBPL, for example, a class definition with its methods may also be preserved in the database. In contrast, in a conventional programming environment code resides in a file system. Keeping code in a repository gives several advantages. Programs are no longer large discrete units; instead they are smaller and typically extract and use sub-programs from the repository. Under this model libraries are easy to use, and large applications can be constructed incrementally [Mem90, Con91, DCC92]. Code reuse would be easy with suitable tools, but without the tools proposed in Section 5 we had some difficulties.

Many of the potential benefits of storing the code in a repository rather than in a file system are not realised because of transitional problems. There are many tools available to support application development using file-based code, e.g. *make*, *rcs* and *grep*. Analogous tools are required to operate on code in a repository. Potentially these tools can be superior to those operating on byte-stream files because a repository is coherent, transactional, structured and typed. Such tools were not available to us during the TA construction, but some are now being constructed [Sjø93b].

Similarly, guidelines or design principles were developed for traditional programming languages where programs communicate via data files, e.g. structured programming [DDH72, Jac75] or modularisation where a high degree of cohesion and a low degree of coupling among software components should be pursued [CY79]. Such guidelines would also apply to a DBPL, but due to more sophistication and communication via a strongly typed repository more comprehensive programming methodologies are needed to fully benefit from the new technology. Currently such methodologies are rudimentary, although they are being developed [Atk93, Sjø93b]. Comparing file-based program construction methodologies with those based on persistent stores, we observe that in the persistent store case all possible data structures (e.g. Maps) and their types are accommodated and preserved when data is stored for later use or passed between programs. Typically file-based program composition has little support from the type system and perforce loses structural information as data is mapped to a sequence of bytes. Although persistence leads to more sophisticated interfaces between program parts using arbitrary modules, we believe it will ultimately yield benefits because of the significant structural information that is conveyed between programs.

Acknowledgements

Paul Philbrow co-authored the Maps libraries. Ray Welland made several useful comments on the paper. The St Andrews development team, in particular Quintin Cutts and Graham Kirby, provided robust software for us to reuse.

Dag Sjøberg's work was supported by the Research Council of Norway, Division NAVF. Malcolm Atkinson's work was supported by FIDE (ESPRIT BRA 3070). João Lopes's work was supported by the Portuguese National Council for Science and Technology Research (JNICT, "Programa Ciência", scholarship BD/1310/91–IA). Phil Trinder's work was supported by the SERC Bulk Data Type Constructors (project GR/F28953).

References

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[ABJP93] M.P. Atkinson, P.J. Bailey, N. Jackson, and P.C. Philbrow. Napier88 libraries. Technical report, in preparation, Department of Computing Science, University of Glasgow, 1993.

[ACC82] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-algol: An algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[ALPR91] M.P. Atkinson, C. Lécluse, P.C. Philbrow, and P. Richard. Maps as bulk types for data base programming languages. In *Proceedings of the Annual Esprit Conference (1991)*, 1991.

[AM85] M.P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 4(7):539–559, October 1985.

[Atk93] M.P. Atkinson. Lecture notes in Napier88 programming. Department of Computing Science, University of Glasgow, 1993.

[ATW93] M.P. Atkinson, P.W. Trinder, and D.A. Watt. Bulk type constructors. Technical Report FIDE/93/61, ESPRIT Basic Research Action, Project Number 3070—FIDE, 1993.

[BLN86] C. Batini, M Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, April 1986.

[CDK90] Q.I. Cutts, A. Dearle, and G.N. Kirby. WIN programmers' manual. Technical Report CS/90/17, Department of Computational Science, University of St Andrews, 1990.

[Con91] R.C.H. Connor. *Types and Polymorphism in Persistent Programming Systems.* PhD thesis, Department of Computational Science, University of St Andrews, 1991.

[Cut93a] Q.I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution.* PhD thesis, Department of Computational Science, University of St Andrews, 1993.

[Cut93b] Q.I. Cutts. Private communication, 1993.

[CY79] L.L. Constantine and E. Yourdon. *Structured Design.* Englewood Cliffs, NJ, 1979.

[DCC92] A. Dearle, Q.I. Cutts, and R.C.H. Connor. An application architecture using type-safe incremental linking. Technical Report FIDE/92/56, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1992.

[DDH72] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Number 8 in A.P.I.C. Studies in Data Processing. Academic Press, 1972.

[Dea87] A. Dearle. Constructing compilers in a persistent environment. In R. Carrick and R.L. Cooper, editors, *Proceedings of the Second International Workshop on Persistent Object Systems: Their Design, Implementation and Use (Appin, Scotland, 25th–28th August 1987)*, pages 443–455, 1987. Technical Report PPRR-44-87, Universities of Glasgow and St Andrews.

[FDK+92] A. Farkas, A. Dearle, G. Kirby, Q. Cutts, R. Morrison, and R. Connor. Persistent program construction trough browsing and user gesture with some typing. In A. Albano and R. Morrison, editors, *Fifth International Workshop on Persistent Object Systems. Design, Implementation and Use (San Miniato, Italy, 1st-4th September 1992)*, Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1992.

[Fel79] S.I. Feldman. Make — a program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, April 1979.

[Gra84] Peter Gray. *Logic, Algebra and Databases.* Ellis Horwood Limited, Chichester, 1984.

[Jac75] M.A. Jackson. *Principles of Program Design.* Number 12 in A.P.I.C. Studies in Data Processing. Academic Press, 1975.

[KCC+92] G. Kirby, R. Connor, Q. Cutts, A. Dearle, A. Farkas, and R. Morrison. Persistent hyper-programs. In A. Albano and R. Morrison, editors, *Fifth International Workshop on Persistent Object Systems. Design, Implementation and Use (San Miniato, Italy, 1st-4th September 1992)*, Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1992.

[KD90] G.N.C. Kirby and A. Dearle. An adaptive graphical browser for Napier88. Technical Report CS/90/16, Department of Computational Science, University of St Andrews, 1990.

[Kir93] G.N.C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* PhD thesis, Department of Computational Science, University of St Andrews, 1993.

[Kru92] C.W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[MBCD89] R. Morrison, F. Brown, R. Connor, and A. Dearle. The Napier88 reference manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, June 1989.

[MBDA86] R. Morrison, A.L. Brown, A. Dearle, and M.P. Atkinson. An integrated graphics programming environment. *Computer Graphics Forum*, 5(2):147–157, June 1986. Also available as PPRR-14-86.

[MBDA88] R. Morrison, A.L. Brown, A. Dearle, and M.P. Atkinson. Flexible incremental binding in a persistent object store. *ACM SIGPLAN Notices*, 23(4):27–34, April 1988.

[MBDA90] R. Morrison, A.L. Brown, A. Dearle, and M.P. Atkinson. On the classification of binding mechanisms. *Information Processing Letters*, 34:51–55, February 1990.

[Mem90] Members of the FIDE types club with M.P. Atkinson and P. Richard as editors. Types for large scale systems. Club report of meeting in Pisa, 5th–6th July 1990. Technical Report FIDE/90/1, ESPRIT Basic Research Action, Project Number 3070—FIDE, October 1990.

[Mun93] D. Munro. *On the Integration of Persistence, Concurrency and Distribution.* PhD thesis, submited, Department of Computational Science, University of St Andrews, 1993.

[Sjø92] D. Sjøberg. Measuring name and identifier usage in Napier88 applications. Technical Report FIDE/92/37, ESPRIT Basic Research Action, Project Number 3070—FIDE, 1992.

[Sjø93a] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[Sjø93b] D. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems.* PhD thesis, Department of Computing Science, University of Glasgow, 1993.

[Tri91] P.W. Trinder. Comprehensions, a query notation for DBPLs. In P. Kanellakis and J.W. Schmidt, editors, *Proceedings of the Third International Workshop on Database Programming Languages (Naflion, Greece, 27th-30th August 1991).* San Mateo, CA: Morgan Kaufmann Publishers, 1991.

[Zhe92] Qin Zhenzhou. Second year report. Department of Computing Science, University of Glasgow, 1992.