Analysing Persistent Language Applications

Dag I.K. Sjøberg
Department of Informatics, University of Oslo
Oslo, Norway. dagsj@ifi.uio.no

Quintin Cutts
Department of Mathematical and Computational Sciences, University of St Andrews
St Andrews, Scotland. quintin@dcs.st-andrews.ac.uk

Ray Welland, Malcolm P. Atkinson
Department of Computing Science, University of Glasgow
Glasgow, Scotland. {ray, mpa}@dcs.glasgow.ac.uk

Abstract

Most research into persistent programming has been directed towards the design and implementation of languages and object stores. There are few reports on the characteristics of systems exploiting such technology. This paper reports on a study of the source code of 20 applications consisting of more than 108,000 lines of persistent language code. The authors of the applications range from students to experienced programmers. The programs have been categorised and examined with respect to a persistent application model and the extent of inconsistencies relative to this model is presented. The results confirm the need for and give input to the design of programming methodologies and tools for persistent software engineering. Measurements also include the use of names, types, (polymorphic) procedures and persistent bindings. It is hoped that analysis of the measurements will be used as input to the next generation of languages and programming environments. As part of this new generation, a measurements system is outlined operating entirely within the persistent environment, thus simplifying access to and measurement of both static and dynamic information.

1 Introduction

This paper reports on an analysis of applications written within a persistent programming environment. Whilst much persistence research has been directed towards the design and implementation of persistent languages and object stores [1-3], there are few reports on the characteristics of application systems exploiting such technology. As the programming effort using persistent languages expands from the current core system building domain to the wider application building domain, results of this nature are required to ensure that the expected wide-ranging benefits of persistent systems are realised. The analysis of persistent language applications presented in this paper aims to inform research in the areas of persistent programming methodology design, the design and implementation of persistent programming environments and associated tools and the design and implementation of persistent programming languages.

The analysed software was written in Napier88 [4], a strongly typed, higher-order persistent programming language. The applications derive from a number of sources, from the Napier88 programming environment software, to student programs to the first major non-system applications written using the Napier88 system. The measured software comes from three separate sites: University of St Andrews, University of Glasgow and Napier University in Edinburgh.

These programs have been written over a number of years during which time different application construction styles have evolved. Each style is termed a programming methodology, and the adherence to and design of these methodologies are of particular interest here. The designer of a programming methodology takes the features of a programming language and its associated environment and generates guidelines and constraints to aid construction of applications according to a clear, widely used and coherent framework. Such a standardisation is intended to improve the quality of software engineering and is particularly important in a fledgling programming style such as persistent programming since there are few, if any, well-founded and proven methodologies available.

The measurements collected in this study are assessed against one of the first well-defined persistent programming methodologies, first to see to what degree the applications adhere to the methodology and second to determine how the methodology should best evolve to meet the requirements of application builders. Statistics on the dependencies among the various parts of an application system indicate the consequences of change, including the extent of necessary change propagation [5].

As a supplement to anecdotal description of user experiences, attempts should be made to quantify the potential benefits of new and enhanced methodologies and tools [6, 7]. This may be achieved by measuring software before and after the methodologies have been adhered to and the supporting tools applied. A strictly controlled experiment was not conducted, but groups of different kinds of programmers were compared.

The measurements are also used to indicate areas of concern in current persistent programming languages and environments. In Napier88, for example, some application construction styles are not efficiently supported by the current programming environment because it is outside the persistent environment. An integrated persistent programming environment is required to make the most of the benefits offered by persistence.

The measurements presented here are taken from a static analysis of source code only. Studies based on static analysis have been reported for other languages, e.g. FORTRAN [8, 9], PL/1 [10] and APL [11], but these studies focused on other issues than those reported in this paper. Also programs written in persistent programming languages have been analysed by others, but only some dynamic aspects relating to performance have been measured [12-14]. The analysis reported in this paper is restricted to static analysis and to a particular language environment but represents a first attempt at analysing the characteristics of persistent language applications. The emergence of integrated persistent programming environments, e.g. [15], will allow measurement of both static and dynamic application characteristics within a fully persistent system. Such measurements should give a more complete analysis of persistent applications. Although measurements from such systems are not yet available, this paper outlines measurement techniques that exploit the power of the new technology and will be used in the next stage of persistent language analysis. As the size and complexity of persistent applications increase, these features will be

required to ensure that effective and enhanced measurement and analysis can be performed.

The paper is organised as follows. Section 2 describes the apparatus used to gather the measurements. Sections 3 reports the main results, which are analysed in Section 4. Section 5 outlines new measurement technology. Section 6 concludes.

2 Measurement Apparatus – Methodology, Tools and Applications

The methodologies, tools and measurements presented here are drawn from experience using the first release of the Napier88 system. In this version, the programming environment in which source programs are constructed and compiled into executable programs is the Unix™ system [16] (Figure 1). Compiled programs are executed against a persistent store. The internal structure of the store is not defined by the language or the store and may consist of an arbitrary graph of Napier88 values, both data and first-class procedures. The graph is reachable from a single point known as the root of persistence. By convention, the graph is structured using a Napier88 type constructor known as an environment [17], which is a collection of name-type-value-constancy bindings [18]. A binding in an environment is to a location containing a value of the given type that may be overwritten with another according to the binding's constancy. Nesting of environments allows a typed hierarchical structure analogous to the directory structure of a file system. A value persists beyond the invocation of the program that created it if it is within the transitive closure of the persistent root.

Libraries of values may be constructed within the store by executing compiled programs that create the new values and bind them to the environment structure where they may be accessed by other programs. Bindings between values and environments are just one example of the bindings between values that may be created during program execution. Applications are typically made up of a number of independently constructed values bound together in such a way that the required task may be performed. In addition, some or all of these values may be bound to the environment structure to allow, for example, an entry point for execution of the application or application components to be examined and possibly updated.

The programming environment embedded within the Unix system allows source program files to be constructed and compiled into executable program files. Where complex type descriptions are shared by many programs, a single version of the type descriptions may be created in a file and compiled into a type library against which source programs may subsequently be compiled.

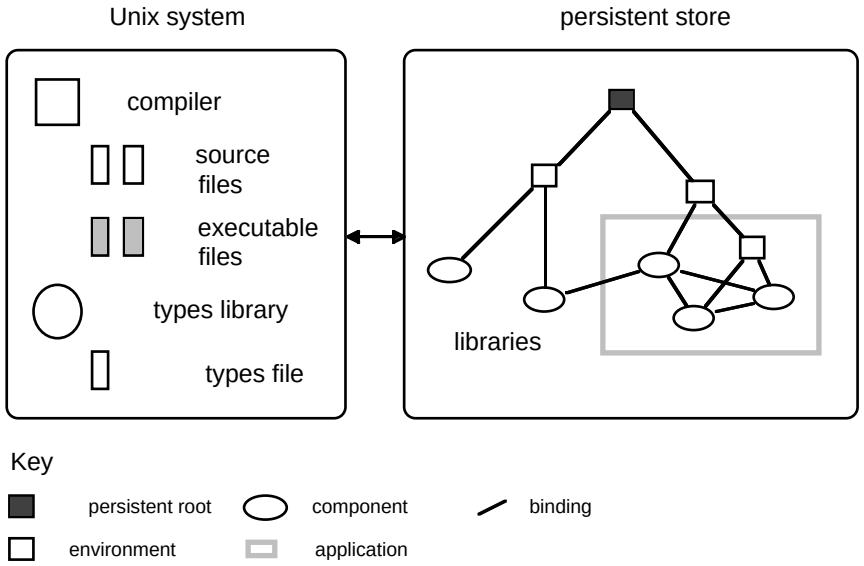

Figure 1: The programming environment

Using a persistent system of this kind, the starting point for the work described here is defined by the following components:

- the persistent programming methodology known as the Structured Persistent Application System Model (SPASM)
- the Thesaurus-based Software Information Tool (TSIT), used to gather information from the source programs
- EnvMake, used to assess this information against SPASM, and
- a collection of persistent applications

2.1 The SPASM Methodology

SPASM defines a programming methodology as a set of constraints to which each suite of application software should adhere in order to help ensure correctness and maintainability [19, 20]. Well-structured software is a requirement for easy maintenance in the future [21, 22]. The SPASM constraints apply to the static software characteristics determined during source code analysis. Some of the constraints are explicit formulations of rules and conventions in a programming culture already adhered to by experienced Napier88 programmers. Other constraints have been defined as a result of the inconsistencies detected in this study.

As a means to improve the way applications are organised around the persistent store, SPASM restricts each program to perform only one kind of operation on the store. Any program should belong to exactly one of the following categories:

- *Insert-program* – inserts at least one binding into an environment in the persistent store but neither updates a persistent location nor deletes any binding.

- *Update-program* – updates at least one persistent location but neither inserts nor deletes any binding.
- *Drop-program* – deletes at least one binding but neither updates a persistent location nor inserts any binding.[1]
- *Startup-program* – uses at least one binding but neither changes the binding to a persistent location, nor inserts or deletes any binding. A startup-program's distinguishing feature is that it does not change any of the bindings in any persistent environment; it typically invokes an interactive menu or any persistent procedure.
- *Type-program* – its contents are exclusively type definitions.

Several constraints help ensure adherence to an incremental construction methodology based on updatable persistent locations [23]. Using the methodology, insert-programs create stub locations in environments, one for each component of the application. For each component, an update-program finds bindings to locations of components required by the component under construction. The update-program creates the new component with bindings to these locations, and updates the component's location with the newly constructed version.

The following are examples of another category of SPASM constraints:

- all type definitions should be used within the application
- a binding inserted into the store, not intended for export, should be used somewhere within the application
- programs and data in the persistent store should be used in at least one application program

2.2 The TSIT and EnvMake Tools

The measurements were provided by TSIT which integrates the notions of data dictionary in the database area and cross-referencer in the programming language area [20, 24]. The TSIT analyser is based on the Napier88-in-Napier88 compiler [25] and extracts a variety of information during source code analysis and inserts it into the thesaurus, which is a fine-grained, cross-reference database containing information about all user-introduced names occurring in the source programs of an application and the names of the bindings to code and other data in the associated persistent stores. A thesaurus entry holds information relevant to our study such as: *name, type, constancy* of an identifier and *usage* and *context* of identifier occurrences. Usage indicates how the identifier is being used, e.g. declaration or use of a type identifier, or declaration, left context or right context of a value identifier. Context indicates whether the identifier occurs in an environment operation or as a declaration of a type parameter, procedure parameter, structure field, variant tag, etc. or as a dereferenced structure field, projected variant, etc.

Another thesaurus-based tool, EnvMake, verifies programs against the constraints of SPASM, using the thesaurus built up by TSIT. EnvMake gives warnings when violations are detected and can be compared with modern grammar

[1] "Delete" is called "drop" in Napier88 terminology. These terms are used synonymously in this paper.

checkers; they check the text against some internal rules and give a warning if the text is not compliant with those rules. EnvMake does not invalidate the program if violations are detected; it informs the programmer about the kind and source of violation and then checks the next constraint. EnvMake features optional selection of the constraints; programmers may "switch off" the check of individual constraints. For example, a programmer may know that certain constraints will not be adhered to during periods of development (typically during initial construction) and may wish to avoid the noise of unnecessary inconsistency messages.

2.3 The Applications

An application is a collection of related programs expected to support a task. Source code information collected from 20 Napier88 applications developed by students and experienced persistent programmers forms the basis for this analysis. The application collection consists in total of 1544 programs, 108,000 lines of code and 180,000 name occurrences (which may be a better measure for the size than the traditional lines of code). A *program* in this context is a unit of compilation, typically contained in a single file.[2] The analysis focuses on the use of names and identifiers. The same name can denote different identifiers if they appear in different scopes. In those cases there are more identifiers than names.[3] In the following program example there are one name, two identifiers and three name (or identifier) occurrences:

```
let counter := 0
begin
   let counter := 1
end
counter := 1
```

In order to investigate the potential benefits of recent innovations in programming methodologies, the applications were divided into four groups: OLD applications developed before the latest methodologies were developed, applications of the STUDENTS who were taught the latest methodologies, new applications of experienced programmers who were AWARE of those methodologies, without fully committing to them, and finally, applications with authors who were explicitly COMMITTED.

3 Results

Table 1 lists a sample of the measurements and summarises immediate findings. Following the table, the measurements are described in more detail. The reader is reminded that these are static measurements of source code.

---

[2] In principle, a program may be represented by several files (e.g. assembled first by a pre-processor, held in a source code control system like RCS, etc.) or may be extracted from one file. The term *module* is often used in the literature synonymously with our definition of program.

[3] In the analysed applications 13% of the names denote more than one identifier.

Table 1: Summary of measurements

Measurement	Immediate findings
Program categories	• old programs do not adhere to the categories, the newest ones do
Inconsistencies	• a relatively large proportion of the SPASM constraints are violated
Use of names	• a name is used between 1 and 7124 times, 13 on average, 90% less than 25 • significant correlation (Spearman [26] 0.88) between number of different names and size of application; no correlation (0.14) between application size and the number of times a name is used • average name length 8 characters, ranging from 4 to 10 in the respective applications; maximum 29
Use of types	• identifiers: structure 21%; monomorphic procedures 18%; ADT only 0.2% • 52 type definitions are used on average 34 times in the applications; average use varies significantly: from 2 to 313 • no significant correlation between the three pairs of number of type definitions, type uses and application size
Use of procedures	• procedures are the most common kind of persistent binding • procedures are used 5 times on average; range of application average: 3 – 9 • one quarter of all procedures are polymorphic
Constancy	• 76% variables, 24% constants
Variable usage	• 15 times more read than update
Store operations	• 73% use, 12% insert, 8% contains check and 7% delete

3.1 Adherence to SPASM

The measurements reported in this section are intended to stimulate persistent methodology designers and tool builders. The large number of inconsistencies and violations of constraints drawn from the measurements justify the implementation of the SPASM verification tool, EnvMake.

3.1.1 Program Categories

Table 2 shows the percentage of the programs belonging to the categories defined in Section 2.3 and illustrates SPASM's effect on the program organisation of the applications under study. The "insert/drop" and "ins/drop/update" columns show the percentage of programs (discouraged by SPASM) containing both insert and drop (and update) statements. It appears that 34% of the programs in the old applications violate the constraint that a program should belong to exactly one of the program categories, as opposed to 6%, 10% and 0% in the other groups. The reason for the large proportion of combined "insert/drop" and "insert/drop/update" programs in the old applications is that many of them adhered to a pre-SPASM methodology in which operations on the same binding were collected in one program. Note also the extremely low proportion (4%) of pure update-programs in the old applications.

Table 2: Distribution of program categories, expressed in percentages

Application group	insert	update	drop	start-up	type	insert/ drop	ins/drop /update	Total
OLD	19	4	3	25	15	22	12	100
STUDENT	34	22	1	29	8	6	0	100
AWARE	11	63	2	8	6	9	1	100
COMMITTED	25	28	25	21	1	0	0	100
MEAN	22	29	8	21	7	10	3	100

One of the two committed applications has one type-program and for each of the persistent components exactly one insert, update, drop and startup program. (The startup programs are test programs in this case.) The average program length is 35 lines in the committed group as opposed to 137 in the other groups. Keeping programs small is in compliance with good software engineering, as maintenance costs have been measured to be significantly affected by program size [27]. On the other hand, there are indications that too small programs should also be avoided [27]. However, if programs have simple semantics and are as well-structured as in this case, they may be the subject of (semi) automatic creation and maintenance [19].

3.1.2 Inconsistencies

SPASM and EnvMake were developed to support software builders in creating more consistent and maintainable application systems. The inconsistencies enumerated in Tables 3 and 4 are examples of inconsistencies the SPASM constraints aim to prevent. Some constraints operate within programs only (Table 3); other operate between programs, i.e. at the application level (Table 4). The applications were operational at the time of the analysis. One would expect an even larger number of inconsistencies during periods of development. A violation of a constraint could be a logical error or could just indicate a situation that might eventually cause problems. For example, redundant type and value declarations do not affect the functionality of a program, but should be avoided since they may cause confusion when someone tries to understand the program, and the programs become unnecessarily large and complex, which in turn may impair performance and maintainability. In a study of FORTRAN programs a correlation was found between the proportion of unused variables and fault rate [9].

Table 3: Measurements of inconsistencies within a program

Inconsistency within a program	Percentage	Percentage of
Variables not updated	35	all variables
Unused value identifiers	8	all declared identifiers
Variables updated but not read	4	all updated variables

Table 4: Measurements of inconsistencies within an application

Inconsistency within an application	Percentage	Percentage of
Repeated type declarations	29	all type identifiers
Unused type identifiers	24	all type identifiers
Repeated drop statements	10	all drop statements
Inserted procedure variables not updated	9	all inserted procedures
Inserted bindings not used	8	all inserted bindings
Repeatedly inserted bindings	7	all inserted bindings
Inserted procedure variables updated more than once	5	all updated procedures

More than one third of all variables are never updated and could therefore have been declared constants (Table 3, row 1). Store managers might exploit this information, which also indicates possible improvement in programming precision. There are large individual variations among the applications (from 0% to 83%, the student applications in the upper range). In addition to being updated, the value of a local (i.e., transient) variable should also be read within the program (row 3 shows 4% violations). A *persistent* variable should be assigned but not necessarily read within the program since its value may be read in other programs.

The majority (72%) of unused value identifiers (8%, row 2) are declared in the constructs used to access bindings in the persistent environment. There are several reasons for why this kind of redundancy occurs: large specifications are copied indiscriminately from other programs; too many identifiers are declared in the belief that they would be needed later; and code using identifiers is removed without the programmer remembering to remove the corresponding declarations. One application has a very low value (0.6%) due to the use of EnvMake, which detects and invites the programmers to eliminate such anomalies.

In a language allowing definition of types in different scopes, two or more types may be defined with the same name and type (expression). In that case, according to the model of SPASM, they should be replaced by exactly one definition in the innermost scope covering the scope of the replaced type definitions. Also, type definitions may have the same name, but denote different types. To avoid confusion they should then be renamed to acquire unique names. Multiple declarations of type names are confusing, require unnecessary compilation and are a potential problem concerning change. Maintaining consistency requires that all declarations describing the same concept (e.g. *Person*) must be changed if the intention is to modify the implementation of the concept (e.g. add a new attribute). It is difficult to arrange that when several programmers (responsible for several components) who require use of a common type, each writes out equivalent type definitions (particularly if they are complex). It is even harder to ensure that when the type is amended, the same amendments are applied in every usage context. One concept should therefore be represented by only one type definition. In our sample 29% of the type declarations are re-declarations (Table 4, row 1). In the most extreme application all types are declared within the program in which they are used. In that application there are 5.6 declarations per name. A requirement for type management is identified here.

The second row of Table 4 shows that 24% of the type identifiers are unused. Some applications use all the type identifiers declared within the application; other

applications use only one third. In the latter extreme cases the reason is that when libraries are used, all the types associated with the library are copied even though only a small part of the library is actually used in the application. This indiscriminate copying of types is indicative of a requirement for a tool to collect required items (types or values).

Bindings inserted into a persistent store by one program should be used in some other program (row 5). Even library components intended for export should be used in at least one program testing the component. More than one declaration of insert for the same binding may cause confusion and are unnecessary. Row 6 shows that 7% of all insert declarations are re-declarations. Several drop statements for the same binding should also be avoided (row 3). Attempts to re-insert a binding already present in a persistent store or drop a binding not present will cause run-time errors.

The fourth and seventh rows of Table 4 relate to the methodology where code resides in updatable persistent locations in the form of procedures. Each such procedure should have exactly one corresponding program updating it. The measurements show that 9% of the procedures are not updated at all; 5% are updated twice or more. Not all the applications have explicitly committed to the methodology at the time of development. This is reflected in great individual variations. Generally, the extent of inconsistencies is clearly smallest in the COMMITTED group, but still not ignorable – automatic detection tools would be useful for all the applications.

3.2 Use of Names

Names are central to system builders' thinking and thus influence the way software is organised. Meaningful names are important for problem solving, understanding of semantic structure and memorisation [28-30]. Within an application people should use names with a consistent intended meaning. The choice of names for identifiers is crucial for the readability of programs and is particularly important when trying to administer and manage change.

A property of a name is its length. There may be different guidelines for the optimal length: the names should generally be long since long names can convey more information than short ones; the less frequently an identifier is used the longer it should be; the greater the distance between the declaration and use of an identifier the longer it should be; etc.[4] Another view is that the important thing is that the name is carefully chosen – which is independent of the name length (e.g. abbreviations can be very meaningful). The appropriateness of these guidelines, which are not mutually exclusive, is not an issue of this paper. The point is, however, that the thesaurus provides a means for testing the software against such guidelines.

3.3 Use of Types

The language under study (Napier88) supports a rich type system, including labelled Cartesian products (structures), labelled disjoint sums (variants), polymorphic procedures [31] and abstract data types [32]. Knowledge of the distribution of base types and type constructors (*kinds*) may be useful for language designers. The most

[4] However, longer names are harder to type correctly. There is therefore a case for completers and information retrieval tools that operate using the thesaurus.

frequently used kinds in all the applications are structures (records) and monomorphic procedures. Some kinds vary significantly, such as polymorphic procedures (from 0.1% to 10%). (The use of polymorphic procedures is discussed further in Section 3.4.) The abstract data type construct is hardly used (see Section 4.3).

Studies have confirmed that type definitions undergo considerable change in large application systems [33, 34]. Measurements on the use of type definitions are interesting when studying the consequences of type evolution. A change to a type definition in the applications under study would require 34 individual edits on average. In the best (but useless) case only the definition itself needs to be changed (no uses), while 3211 places in the worst case.

The argument above should be modified slightly. A renaming of a type definition would require all the places where the type identifier is used to be edited. If the expression of a type definition is changed, the places of use must be changed depending on the context and whether the type is parameterised. If a type identifier is used to create instances of the type denoted (16% of all uses), a change must be propagated to all places where the identifier is used to create new instances (five places on average in the applications). If a type identifier is used in the declaration of another type, in the signature of a procedure parameter declaration or in the construct used to access persistent bindings from a program (these three cases constitute 84% of all uses), a change does not affect the code if the type is not parameterised; only recompilation is necessary. (For the 18% of type definitions that are parameterised, the place of use must be edited if the number of parameters is changed.) However, in addition to the required edits on the places where a type is used, cascades of consequential change might be necessary (e.g. the places where instances of the types are used).

The number of programs affected by a type change may be a measure for how modular the code is. On average a type definition is used in respectively 38%, 13%, 12% and 5% of all programs in the OLD, STUDENTS, AWARE and COMMITTED groups, indicating that the "committed" programmers produce the most modular code. In any case, the measurements presented above confirm that software builders and maintainers need sophisticated change management tools and that SPASM will make this kind of change easier to manage.

The type equivalence model of the language must be taken into account when attempting to manage types. In a language with structural type equivalence, such as Napier88 [35], determining the consequences of type change can be difficult. The thesaurus information about the use of types, on which our measurements are based, may be incomplete. Instead of the name of a type definition, anonymous types may be used in value instantiations and other declarations. A problem occurs when an anonymous type is semantically the same as another explicitly defined type. This illustrates that programmers should be encouraged to use named types in order to facilitate efficient change propagation. However, sometimes using names only may impair readability. Change management tools could in those cases pass back information to the programmer on an interactive basis when an anonymous type is found that is equivalent to a changed named type. The programmer could then specify the desired course of action.

From the point of view of language implementors, information about the types involved in type checking would be useful to collect in a future study (Section 5). Choosing an efficient strategy for managing representations of types depends on the proportions of time spent on, for example, constructing the representations,

examining components of representations and testing for type equivalence over representations. In conjunction with dynamically gathered information of this kind, static information about the structure and use of types will help indicate the appropriate implementation strategy.

3.4 Use of Procedures

In an orthogonally persistent language providing first-class procedures, executable code can be contained in persistent stores in the form of procedures. Our study shows that this feature is heavily exploited: 58% of all bindings inserted into a persistent store are monomorphic or polymorphic procedures (as opposed to only 16% of the transient identifiers). The large proportion of procedures is primarily caused by the kinds of the analysed applications. The infancy of our programming environment has resulted in more tools and libraries than application systems with huge amounts of data.

The name, type or value of a procedure may change. A change to the value (body) will normally not require any propagation to other parts of the application. Renaming or changing the type implies in general that all places of use must be changed accordingly. In the analysed applications an average of five places was measured.

Language designers should note that polymorphic procedures, a relatively new construct, are becoming more widely used. Table 5 shows the use of polymorphic and monomorphic procedures in proportion of all identifier occurrences.

Table 5: Use of polymorphic and monomorphic procedures

Application group	Polymorphic procedures %	Monomorphic procedures %
OLD	1.4	18.1
STUDENT	4.7	21.8
AWARE	5.0	17.9
COMMITTED	7.6	20.4
MEAN	4.7	19.6

Useful information for language implementors is that 86% of the polymorphic procedures have only one quantifier, 13% have two and 1% have three. Moreover, the most efficient implementation strategy for polymorphic procedures will depend on the number of specialisations [25]. Procedures are specialised without call in only four of the 20 applications, in contrast to the belief that specialisation without call is the expected method of using polymorphic procedures [36]. This may affect the chosen implementation of polymorphism. In the four applications, 43% of the polymorphic procedures are involved in specialisations without calls, each procedure being specialised, but not called, on average 4.4 times, with 2.7 different type combinations. However, dynamic measurements of specialisations and calls are also important to get a complete picture of what is going on (Section 5).

3.5 Constancy and Name Usage

A value identifier is declared to be either constant or variable. The proportion of constants is between 2% and 47% in the applications and is significantly lower in the

applications of the undergraduate students than in the other applications.[5] Section 3.1.2 reported measurements showing that in many cases programmers use variables where they should have used constants.

Name usage has been divided into type declarations, type uses, value declarations, left contexts and right contexts. Among the value identifiers respectively 33%, 5% and 62% occur in declarations, left and right contexts, indicating that identifiers are rarely updated compared with how often their values are read.

3.6 Persistent Store Operations

In the applications under study, it appears that in total about 20% of all name occurrences pertain to the access and manipulation of bindings within the persistent store. Figure 2 shows the distribution of the operations in terms of used, inserted and dropped bindings and a check to determine if an environment contains a certain binding. Introducing persistent bindings in the scope of a program is the dominant operation (73%). This is a tedious task that may impair programming efficiency – particularly for large applications with complex type expressions and many bindings. Furthermore, 10% of the identifiers declared in such binding specifications are unused (see also Section 3.1.2). This may result in confusing, verbose and inefficient programs. A binding construct represents a view of an environment (a partial specification of the environment's contents), but the precision in the view identification is lost if the view contains unused bindings as well. Hence, the measurements confirm the need for tools that (partly) automate the process of specifying bindings. Such tools are under development.

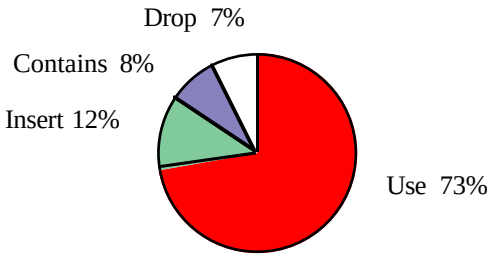

Figure 2: Operations over the persistent environment

The 20% proportion concerning operations over the persistent environment may be compared with corresponding measurements in other programming environments. One example is the classical figure in the persistent literature that typically 30% of all code in conventional languages is concerned with transferring data to and from secondary storage [37]. Comparing those figures requires a closer analysis, however. First, the current language system under test is not a complete self-contained persistent programming environment. Source and executable exist outside the environment, and on execution many bindings to data within the persistent

[5] Probably because the languages the students used previously did not offer initialising declarations.

environment must be made. As Figure 2 reveals, most of the 20% will be concerned with these bindings. Once the complete programming exercise may be carried out within the persistent environment, new technologies such as hyper-programming [38] may be used to remove almost all of this code. Second, even in the integrated persistent environment, such code may still be required when constructing programs in isolation from the data over which they will operate, for example constructing code in one store to be executed against another store [39]. However, note that finding bindings in the store requires relatively simple specifications, which might be (semi) automated. The literature's 30% contains complex data translation algorithms in addition to any binding constructs.

4 Analysis

Some of the results were discussed in the previous section. This section shows how a deeper analysis of the results gives input to further research in persistent methodology design, language design and programming environments.

4.1 SPASM

In the existing programming environment, where source programs are represented as Unix files, adherence to SPASM gives more well-structured (each program performs only one kind of operation on the persistent store, for example) and more maintainable programs (unused identifiers and bindings are an obvious source of confusion, for example). The results also show that there are fewer inconsistencies in the applications that have explicitly committed to the methodology. However, the study reveals many suggestions for improvement of SPASM.

In our study 8% of all value identifiers were unused, which is in contrast to, for example, 28% reported in a study of production PL/1 programs [10]. As opposed to Napier88, PL/1 does not allow declaration with initialisation, which is surely a cause for the large proportion of unused declarations in PL/1. Three quarters of the unused identifiers in our study were declared in binding specifications. This finding of the paper may suggest one of the following:

i) Binding specifications should be the subject of automation, possibly with some interaction with the programmer [19].
ii) In conjunction with new programming environments (Section 4.3), SPASM could evolve to actively support improved ways of binding values and type representations to programs.

Many inconsistencies relative to a methodology like SPASM might be due to poor software development or insufficient detection tools. On the other hand, programmers might deliberately violate the constraints due to new ways of constructing applications. For example, in the current version of SPASM a component should have only one update-program, but one could envisage programming styles where one would wish several update-programs for each component, e.g. a compiler could be implemented with different versions for different machine architectures. Each update-program may configure the application in a slightly different way, by inserting a component of the right type but with differing internals. The point is that the programming methodology and supporting

tools should be easy to modify in compliance with changed working practices, which can be detected by measurements such as those reported in this study.

4.2 Suggestions for the Language Designers

Abstract data types are hardly used in the applications under study. (It should be noted though that many of the applications measured are system programs which would not be expected to depend heavily on the use of abstract data types of this kind.) Little use of a construct could indicate that it is useless, but in this case discussion with the language designers and with programmers indicates that the low usage is through lack of understanding of how to use them and of the extra power they give over first-order information hiding. Better tuition is required – a programmer's tutorial manual in particular.

The large proportion of variables not updated (35%) might suggest that identifiers in binding specifications, procedure headers, etc. should be declared constant by default, instead of variable which is the default at present. This is a borderline case, however, since given a reduction of 35% in the number of variables there would still be as many variables as constants.

4.3 Suggestions for Programming Environments

The results show that there are two areas requiring examination in the measured programming environment. The first involves the binding mechanism between components. The underlying binding architecture, where a component links to the typed *locations* of other components that it uses, and performs a dereference on each access, gives the advantage of type-safe incremental linking. However, the programming environment does not support a simple mechanism for setting up an application in this style; indeed, the dependence on dynamic binding constructs during application construction has been detected here as a major cause of programming inconsistency.

The second area for examination is the management of type information. In a persistent system, use of the type system plays a major rôle in any application for both data modelling and protection. The measured programming environment has very limited support for controlling type information causing essential processes such as type evolution to be hard to support. Even with a tool such as TSIT it is difficult to detect the manner in which types are used and related to one another.

The next generation programming environments [40] aim to overcome the problems detected here. The principal change is that the entire application construction process is supported within a single persistent environment. The separate processes of program construction, compilation, linking and execution may all be performed within the environment. The advantages of such a construction environment are described in [25].

Of interest here is the hyper-programming concept [38], where links to values and locations already existing in the persistent store may be included directly in source programs under construction. By analogy with hyper-text, a hyper-program is a structured version of the traditional flat source code representation that contains both flat code and links to language values. Using hyper-programming, linking between components and locations may take place during code construction, avoiding the requirement for the error-prone dynamic binding clauses required in the generator/update programs of the current programming environment.

Hyper-links may be used to represent the values of free variables in a source-code representation of a procedure closure. This style of source code, known as hyper-code, may be used to construct a source representation for any language value [41]. Using hyper-code, the two separate entities of source and executable exist as alternative views of a single value. A value may be analysed by directly examining its state via the bindings contained in its hyper-code source representation.

In addition, hyper-links to type representations may be included in programs. These may be used to form the basis of a type management system. Programs are linked only to the types they use directly. So the spread of type information, viewed as a problem here, is minimised.[6] Engineering reverse links from types to the programs that use them might optimise some aspects of the difficult task of type evolution [34, 42]. In general, the features supported by hyper-programming [43] will play a major part in the formulation of new programming methodologies in addition to the direct knowledge gained from use of current methodologies and the measurement results as described here.

5 The Next Generation Measurement Tools

TSIT gathers information based on a purely static analysis of the source code of the programs making up an application. A drawback of this approach is the inability to collect measurements on the dynamic behaviour of programs. Examples of dynamic measurements that have been requested by the language implementors and that cannot currently be determined using TSIT include:

- The proportions of type equivalence checks that fail and succeed. The efficiency of type equivalence checking is significant in persistent systems [44], and a measurement such as this would inform decisions on the appropriate implementation for the checking algorithm.
- Specialisations of polymorphic procedures. Information about the range and frequency of types used to specialise particular polymorphic procedures can be used to provide optimised implementations for those procedures. *Ad hoc* measurement techniques to gather this information have been used in an optimisation of polymorphic procedures described in [25].

In addition, TSIT is not well integrated with the operation of current persistent programming environments. Programs making up an application are passed to TSIT in isolation from the processes of compilation and linking. Unless use of such a tool can be simply incorporated into the programming process, it is unlikely to be used regularly. Solutions to these two problems will be addressed separately.

5.1 Integrating TSIT into a Persistent Programming Environment

TSIT may be integrated into a persistent programming environment by making it an optional part of the compilation process. It has many of the features of a parser anyway in its ability to examine code. Such features may be shared between both

[6] However, this approach does not help when we consider types as descriptive meta-data to aid program understanding. In that context identifiers are vital.

TSIT and the parser. For each application, a database of information may be built up as the separate components are compiled.[7] The TSIT part of the compiler may be parameterised by this database before compilation begins, enabling access to meta-data during compilation.

5.2 Measuring Dynamic Behaviour

In an integrated persistent programming environment, the compilation process may augment programs with extra code to gather information during subsequent execution of the compiled code. This information is cheaply retained in the persistent environment where it can be accessed by analysis programs at a later time. Whilst such a process is possible to achieve in a non-persistent programming environment, it will generally be more expensive as the compilation, execution and analysis phases are less well integrated. A full discussion of this style of measurement, used as part of a general optimisation scheme, is described in [45].[8]

In this context, the new version of TSIT has access to application source code and may add measurement code during the compilation process. The data gathered during execution may be made available to an analysis program making up part of the measurements suite. Linking the static and dynamic analyses is important and not yet done.

6 Conclusions

As a means to acquire more knowledge about persistent software engineering, relevant measurements should be obtained. Claimed problems and proposed solutions should be quantified. The explorative study of 108,000 lines of persistent language code in 20 applications reported in this paper is one step in that direction. Some of the results are applicable to any programming environment, for example:

- The extent of unused types, repeatedly declared types, unused variables, variables not updated, etc. confirm the need for automatic prevention or detection tools.
- Statistics on the use of type definitions and procedures illustrate the consequences of change and the need for change management tools.

Other results are specific to persistent programming:

- The effect of persistent design principles applied in some applications was measurable in terms of improved program structure and consistency.
- The study detected inconsistencies relative to a certain methodology such as bindings inserted into a persistent store but not used, repeatedly inserted bindings, bindings dropped more than once, etc. Automatic prevention or detection tools are needed.

---

[7] The disadvantage of this approach is that it slows compilation and collects “noise”. It is a pertinent move to the design of the compiler but less to the design of the language. The registration model with user activated and periodic scans also has advantages and can be easier to use.

[8] Also Jackson’s monitor at Glasgow allows relevant instrumentation [14].

- In the current programming environment, in which source code is contained in Unix files, measurements show a relatively large proportion of code containing specifications of persistent bindings to be used in a program and a high inconsistency rate in this code. This indicates the usefulness of tools for (semi) automatic generation of such specifications when building applications in the current environment. In a fully integrated persistent programming environment [40], however, the proportion is expected to be significantly reduced.

The measurements were collected by first extracting information about all the identifier occurrences in all the software of the applications. The information was stored in a meta-database that was then the subject of statistical analysis. This measurement technique can be applied to any programming language. One example is an earlier study we conducted in an industrial (C, C++, X Window System and relational database) environment [34]. However, studying the pattern of how programs operate on persistent data, the consequences of change, etc. are simpler in a persistent programming environment in which only one language is used and the application programs, database schema (set of type definitions) and extensional data are integrated in the same store.

The results of this experiment enable us to design controlled experiments that could, for example, test the real effect of persistent programming methodologies and tools such as SPASM and EnvMake [19, 20]. Knowledge of the use of language constructs (little use of abstract data types, increased use of polymorphic procedures, etc.) is useful for language designers. The results also form a basis for further experiments on optimisation, consequences of change, etc. by combining static and dynamic analysis.

Acknowledgements

The St Andrews persistent programming team provided the underlying language technology and made several useful comments on the analysis and on this paper. We also thank the anonymous referees for their helpful suggestions for improvement. We are grateful to Paul Philbrow and others who helped providing the software that was analysed. Dag Sjøberg was supported by the Research Council of Norway. Some of the reported work was also supported by a European Community ESPRIT Basic Research Action, $FIDE_2$, number 6309.

References

1. Dearle A, Shaw GM, Zdonik SB. Proceedings of the Fourth International Workshop on Persistent Object Systems, Their Design, Implementation and Use. Martha's Vineyard, USA, 23rd–27th September: Morgan Kaufmann, 1990

2. Albano A, Morrison R. Proceedings of the Fifth International Workshop on Persistent Object Systems: Design, Implementation and Use. San Miniato, Italy, 1st–4th September: Springer-Verlag and British Computer Society, 1992

3. Beeri C, Ohori A, Shasha DE. Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages. Manhattan, New

York City, USA, 30th August – 1st September: Springer-Verlag and British Computer Society, 1993

4. Morrison R, Brown F, Connor R, Dearle A. The Napier88 Reference Manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989

5. Atkinson MP, Sjøberg DIK, Morrison R. Managing Change in Persistent Object Systems. In: Nishio S, Yonezawa A, ed. First JSSST International Symposium on Object Technologies for Advanced Software. Kanazawa, Japan, 4th—6th November: Lecture Notes in Computer Science 742, Springer-Verlag, 1993, pp 315–338

6. Basili VR, Reiter RW. A Controlled Experiment Quantitatively Comparing Software Development Approaches. IEEE Transactions on Software Engineering 1981; SE-7(3):299–320

7. Law D, Naeem T. DESMET – Determining an Evaluation Methodology for Software Methods and Tools. In: Spurr K, Layzell P, ed. CASE, Current Practice, Future Prospects. J. Wiley & Sons, Chichester, England, 1992, pp 167–181

8. Knuth DE. An Empirical Study of FORTRAN Programs. Software – Practice and Experience 1971; 1(2):105–133

9. Card DN, Church VE, Agresti WW. An Empirical Study of Software Design Practices. IEEE Transactions on Software Engineering 1986; SE-12(2):264–270

10. Elshoff JL. An Analysis of some Commercial PL/1 Programs. IEEE Transactions on Software Engineering 1976; SE-2(2):113–120

11. Saal HJ, Weiss Z. An Empirical Study of APL Programs. Computer Languages 1977; 2(3):47–59

12. Bailey PJ. Performance Evaluation in a Persistent Object System. In: Rosenberg J, Koch D, ed. Third International Workshop on Persistent Object Stores. 10th–13th January 1989, Newcastle, New South Wales, Australia: Springer-Verlag and British Computer Society, 1989, pp 289–299

13. Loboz Z. Monitoring Execution of PS-algol Programs. In: Rosenberg J, Koch D, ed. Third International Workshop on Persistent Object Stores. 10th–13th January 1989, Newcastle, New South Wales, Australia: Springer-Verlag and British Computer Society, 1989, pp 279–288

14. Atkinson MP, Birnie A, Jackson N, Philbrow PC. Measuring Persistent Object Systems. In: [2], pp 63–85

15. Morrison R, Brown AL, Connor RCH, Dearle A, Kirby GNC, Cutts QI. The Napier88 Reference Manual (release 2.0). Technical Report CS/93/15, Department of Mathematical and Computational Sciences, University of St Andrews, 1993

16. Ritchie DM, Thompson K. The UNIX Time-Sharing System. The Bell System Technical Journal 1978; 63(6):1905–1930

17. Dearle A. Environments: A Flexible Binding Mechanism to Support System Evolution. In: 22nd International Conference on Systems Sciences. Hawaii, January 1989, pp 46–55

18. Morrison R, Brown AL, Dearle A, Atkinson MP. On the Classification of Binding Mechanisms. Information Processing Letters 1990; 34(1):51–55

19. Sjøberg DIK. Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems. Ph.D. thesis, Department of Computing Science, University of Glasgow, 1993

20. Sjøberg DIK, Atkinson MP, Welland R. Thesaurus-Based Software Environments. Workshop on Software Engineering and Databases in conjunction with the 16th International Conference on Software Engineering. Sorrento, Italy, 16th–17th May 1994

21. Lehman MM, Belady L. Program Evolution, Processes of Software Change. A.P.I.C. Studies in Data Processing No. 27. London: Academic Press, 1985

22. Gibson VR, Senn JA. System Structure and Software Maintenance Performance. Communications of the ACM 1989; 32(3):347–358

23. Dearle A, Cutts Q, Connor R. Using Persistence to Support Incremental System Construction. Microprocessors and Microsystems 1993; 17(3):161–171

24. Sjøberg DIK, Atkinson MP, Lopes J, Trinder P. Building an Integrated Persistent Application. In: [3], pp 359–375

25. Cutts QI. Delivering the Benefits of Persistence to System Construction and Execution. Ph.D. thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1993

26. Kendall MG. Rank Correlation Methods. (Second ed.) London: Charles Griffin, 1955

27. Banker RD, Datar SM, Kemerer CF, Zweig D. Software Complexity and Maintenance Costs. Communications of the ACM 1993; 36(11):81–94

28. Barnard P, Hammond NV, MacLean A, Morton J. Learning and Remembering Interactive Commands in a Text-Editing Task. Behaviour and Information Technology 1982; 1:347–358

29. Weiser M, Shneiderman B. Human Factors of Computer Programming. In: Salvendy G, ed. Handbook of Human Factors. John Wiley & Sons, 1987, pp 1398–1415

30. Anand N. Clarify Function! ACM SIGPLAN Notices 1988; 23(6):69–79

31. Cardelli L, Wegner P. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys 1985; 17(4):471–522

32. Mitchell JC, Plotkin GD. Abstract Types Have Existential Types. Twelfth ACM Symposium on Principles of Programming Languages. New Orleans, 1985, pp 37–51

33. Marche S. Measuring the Stability of Data Models. European Journal on Information Systems 1993; 2(1):37–47

34. Sjøberg DIK. Quantifying Schema Evolution. Information and Software Technology 1993; 35(1):35–44

35. Connor RCH. Types and Polymorphism in Persistent Programming Systems. Ph.D. thesis, Department of Mathematical and Computational Sciences, University of St Andrews, 1991

36. Morrison R, Dearle A, Connor RCH, Brown AL. An Ad Hoc Approach to the Implementation of Polymorphism. ACM Transactions on Programming Languages and Systems 1991; 13(3):342–371

37. Internal Report on the Contents of a Sample of Programs Surveyed. IBM Research Centre San Jose, California, 1978

38. Kirby G, Connor R, Cutts Q, Dearle A, Farkas A, Morrison R. Persistent Hyper-Programs. In: [2], pp 86–106

39. Atkinson MP, Buneman OP, Morrison R. Binding and Type Checking in Database Programming Languages. The Computer Journal 1988; 31(2):99–109

40. Morrison R, Connor RCH, Cutts QI, Kirby GNC. Persistent Possibilities for Software Environments. Workshop on Software Engineering and Databases in conjunction with the 16th International Conference on Software Engineering. Sorrento, Italy, 16th–17th May 1994

41. Connor RCH, Cutts QI, Kirby GNC, Moore VS, Morrison R. Unifying Interaction with Persistent Data and Program. Second International Workshop on User Interfaces to Databases, 1994

42. Connor RCH, Cutts QI, Kirby GNC, Morrison R. Using Persistence Technology to Control Schema Evolution. In: Deaton E, Oppenheim D, Urban J, Berghel H, ed. Ninth ACM Symposium on Applied Computing. Phoenix, Arizona: ACM Press, 1994, pp 441–446

43. Morrison R, Baker C, Connor RCH, Cutts QI, Kirby GNC. Approaching Integration in Software Environments. Accepted subject to revision, Computer Journal 10/93 1993

44. Connor RCH, Brown AB, Cutts QI, Dearle A, Morrison R, Rosenberg J. Type Equivalence Checking in Persistent Object Systems. In: Dearle A, Shaw GM, Zdonik SB, ed. Implementing Persistent Object Bases. Morgan Kaufmann, 1990, pp 151–164

45. Cutts QI, Connor RCH, Kirby GNC, Morrison R. An Execution Driven Approach to Code Optimisation. 17th Australasian Computer Science Conference. Christchurch, New Zealand, 1994, pp 83–92