Thesaurus-Based Software Environments

Dag I.K. Sjøberg,
Department of Informatics, University of Oslo,
N-0316 Oslo, Norway. dagsj@ifi.uio.no

Malcolm P. Atkinson and Ray Welland,
Computing Science Department, University of Glasgow,
Glasgow G12 8QQ, Scotland. {mpa,ray}@dcs.glasgow.ac.uk

1 Introduction

Software environments support the process of constructing and maintaining application systems. This paper describes the idea of a *thesaurus*[1] as a viable foundation for software environments. A thesaurus contains information about the names and identifiers in *all* the software written in *all* the languages of an application. Information about extensional data in a database or persistent store is also included. The comprehensiveness of the thesaurus is in contrast to most commercially available tools which focus either on the source code only (source code analysers) or on database-specific information (data dictionaries). A few data dictionary tools also include source code information, but relationships between names and identifiers in the software written in the various languages are not recorded automatically. All the contents of the thesaurus are automatically maintained. The whole application system is analysed, and the thesaurus updated, regularly at times specified by the user, for example daily at 02:00. A full analysis and update can also be initiated at any time.

Two thesaurus tools have been built. The HMS thesaurus tool was developed for a health management system (HMS) in an industrial (C, C^{++}, X Window System and relational database) environment [13]. Another thesaurus tool was thereafter built in the context of the strongly typed, persistent programming language Napier88 [12]. The software environments that have been built around the thesauri focus on change management and include tools that display structures and dependencies and provide impact analysis. In the persistent case, automatic build management is supported, including installation, smart recompilation [15] and re-execution according to a persistent programming methodology. To prevent deteriorating structure and improve maintainability, a set of application independent constraints have been defined [14]. The programming environment automatically verifies these constraints.

The present tools focus on the implementation phase (initial construction and maintenance). However, automatically maintained thesauri with extended information may form a basis for tools supporting other phases of the life cycle as well.

2 The HMS Thesaurus Tool

The HMS thesaurus tool was developed in an industrial environment in order to identify and help solve real-world problems of maintenance. The analysed software includes

[1] The term *thesaurus* generally denotes “a ‘treasury’ or ‘storehouse’ of knowledge, as a dictionary, encyclopædia, or the like” [1]. In this context the “knowledge” is information about names and identifiers such as where they are defined and used, what kinds they are, in which contexts they occur, etc.

programs written in a screen definition language, a procedural language for defining actions, a query dictionary language and a schema definition language. The tool stores information about name occurrences, like type and container, and records dependencies between occurrences in all the software (including between software written in different languages). An interface provides, among other things, some consistency checking and impact analysis which localises the effects of change within the system. The impact analysis of schema changes has proved particularly useful since software written in all the languages is affected by such changes [13].

3 Thesauri in Persistent Programming Environments

The idea of thesaurus-based software environments is applicable whether the environment is in the context of an industrial relational database with conventional programming languages or an experimental object-oriented database with more modern programming languages, etc. The experiences with the HMS thesaurus tool in an industrial context were valuable, but the provision for integration and longevity makes persistent programming technology a more suitable platform for research into software environments. The concept of persistence tackles the mismatch between database systems and programming languages [2]; a uniform model for representations and operations on persistent and transient data is provided. Tools, programs and data may reside in the same store. Many of the benefits of persistent language technology have been described in the literature [4, 5, 3]. In particular, it has been argued that a transactional, structured and typed persistent store may be an appropriate technology for implementing software environments [6, 11]. Our research supports this view.

The persistent thesaurus is a fine-grained meta-database containing information about all user-introduced names occurring in the source programs of an application and the names of the bindings to programs and other data in the associated persistent store. A thesaurus entry holds information such as: *name* of an identifier, *date* and *time* of when the entry was inserted and *container*, *kind*, *constancy, usage* and *context* of the entry.

In a persistent programming environment, the database schema (set of type definitions), application programs and extensional data are integrated in the same store. This is reflected in the thesaurus in its provision of information about database operations (insert, use update and delete), use of libraries, instances of type definitions, etc. In particular, the thesaurus' description of dependencies between database schemata, application programs and extensional data support maintenance; it is simpler to track down the consequences of change, e.g. schema evolution [13].

Figure 1 illustrates how the notion of persistent programming language (PPL) integrates the notion of database systems and conventional programming languages. Analogous to, and enabled by, this integration is the notion of thesaurus which integrates the notions of data dictionary in the database area and cross-referencer in the programming language area. Traditionally, the integration has been poor.

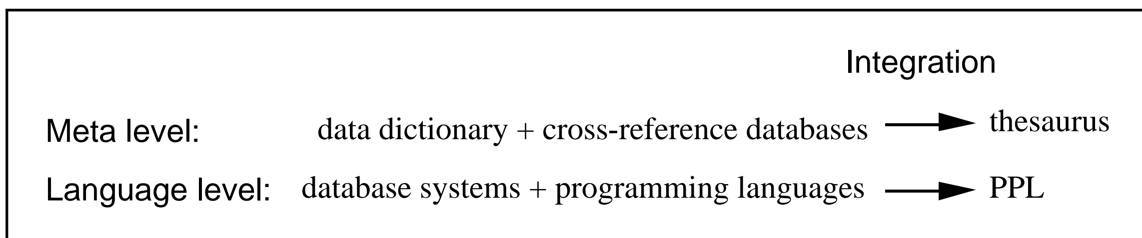

Figure 1: Integration at the language level and meta-data level

4 A Persistent Thesaurus-Based Software Environment

A prototype software environment that utilises database or persistent programming technology has been built around the thesaurus. The provision of persistence has made it easy to build tools working on top of the thesaurus. Some examples follow.

4.1 Interface to the Thesaurus

In addition to a simple textual query interface to the thesaurus implemented by the first author, Lopes has developed a sophisticated window-based interface with enhanced query possibilities [9]. It provides a graphical interface to one or more thesauri and includes a simple query language, a subset of a generalised relational algebra. Complex queries (involving recursion), however, cannot be expressed. To meet this deficiency, the Ringad comprehension query language was constructed by Trinder [16].

EnvMake [14] is another thesaurus-based tool that provides programmers and tool components with dependency tables which, among other things, are particularly useful for determining the consequences of change. For example, one kind of table shows dependencies between programs that insert persistent code (and other data) and those that update them – a so-called "insert/update dependency table". There are similar dependency tables for insert/use, update/use, type definition/type use, etc. Another form of presentation is matrices showing which programs perform which operations on which parts of the database.

4.2 Build Management

At present, many persistent programmers use Make [7] to install software and to help rebuild applications after change. When using Make, the programmers have to manually work out the order of installing components into the persistent store. This may be a difficult task for non-trivial applications. A component must be inserted into the store before it can be used by another component. EnvMake determines the correct installation order by topological sorting [8]. EnvMake automatically infers the necessary dependencies from the thesaurus to initiate (re)compilation and (re-)execution.[2] Hence, there is no notion of an *(Env)Makefile* which has to be created and maintained manually.

In large application systems, recompilations represent a significant part of the maintenance costs and may thus be a hindrance for required system evolution. Make is not particularly helpful in avoiding unnecessary recompilations; it is unlikely that any language independent tool can be smart in that respect. Using the dependency tables (Section 4.1), EnvMake features smart recompilation. The L-value binding model embodied in the methodology [14] significantly reduces the need for cascades of recompilations.

4.3 Constraint Verification

The compiler of a programming language already performs many forms of consistency checks such as type checking, ensuring declaration and unique naming of identifiers, etc. EnvMake is concerned with complementary checks such as those between programs and those between programs and data in a persistent store.[3] Being specific, EnvMake verifies[4] a Structured Persistent Application System Model (SPASM) [14] which is a collection of 24 constraints like the following: "all type definitions should be used within the

[2] Re-execution is used to replace one version of some data or code with a new version.

[3] Similar work has been reported in the context of conventional programming languages [10], but in those cases the constraints involve source code only. The persistent language technology and the thesaurus information enable formulation and verification of constraints concerning the whole processing environment.

[4] At the time of writing, some of the checks still have to be implemented.

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", etc. A violation of a constraint could be a logical error, or it may just indicate a situation that might eventually cause problems. Inconsistent states might be the normal case, particularly during the initial development. Programmers may find it helpful to be able to request that certain subsets of these inconsistencies be enumerated.

Programmers who share a common view of how to develop applications in their environment form a particular programming culture. Such cultures may differ considerably from group to group even though the programming language is the same. The rules and conventions of a programming culture *implicitly* express application models and programming methodologies adhered to within that culture. SPASM is an *explicit* formulation of such a model and methodology in a database programming environment. Several of the constraints are based on a categorisation of programs according to their semantics. On the criteria of how they operate on the persistent store and where types are defined the programs are divided into five categories. The categorisation, which is performed automatically by EnvMake, is also the basis for the build management features described in Section 4.2.

EnvMake also assists in other aspects of construction and maintenance such as organising the structure of environments in the persistent store and directories in the file system. The tool ensures isomorphism and adherence to naming conventions by actively taking part in the creation and maintenance of files and environments.

5 Status and Future Work

The focus on names, one of the characteristics of this work, is justified by the observation that within a context (e.g. an application) people tend to use the names to have a consistent intended meaning. Thus names are interesting markers when trying to administer and manage change. Another novel feature of this work is the architecture of the thesaurus tools. Unlike many CASE tools the thesaurus approach does not utilise a strongly-coupled architecture. In those systems, the compilers and other software production tools have to be modified to update a data repository. This has two serious costs: an impact on tool performance and a need to modify or complicate the tools. In contrast the thesaurus system adopts a different strategy. It scans and analyses the data (database, files, persistent store) associated with the evolving application and derives the relevant data. This has the advantage of accuracy, of allowing independent development of construction tools and of improved performance when programmers are busy.

The idea of automatically generated and updated thesauri, containing information about all user-introduced names in an application, has proved computationally feasible and extremely useful both in an open, industrial environment and in a closed, research environment. Persistent language technology, because it enables applications and tools to be contained in the same coherent, transactional, structured and typed persistent store, creates new possibilities for enhanced and more integrated software engineering support environments. A prototype of such an environment has been designed, and partly implemented, around the thesaurus as illustrated in Figure 2.[5] For example, a tool called EnvMake supports build management such as installation, recompilation, relinking and re-execution. EnvMake automatically tracks down dependencies and initiates the appropriate actions. The tool also supports a persistent programming methodology, including adherence to a collection of application independent constraints. It is a particular concern of this methodology to ensure that the application is and remains amenable to change.

[5] At present, tools operate directly at the data structures of the thesaurus. Defining a general, more abstract interface is a major issue for future work.

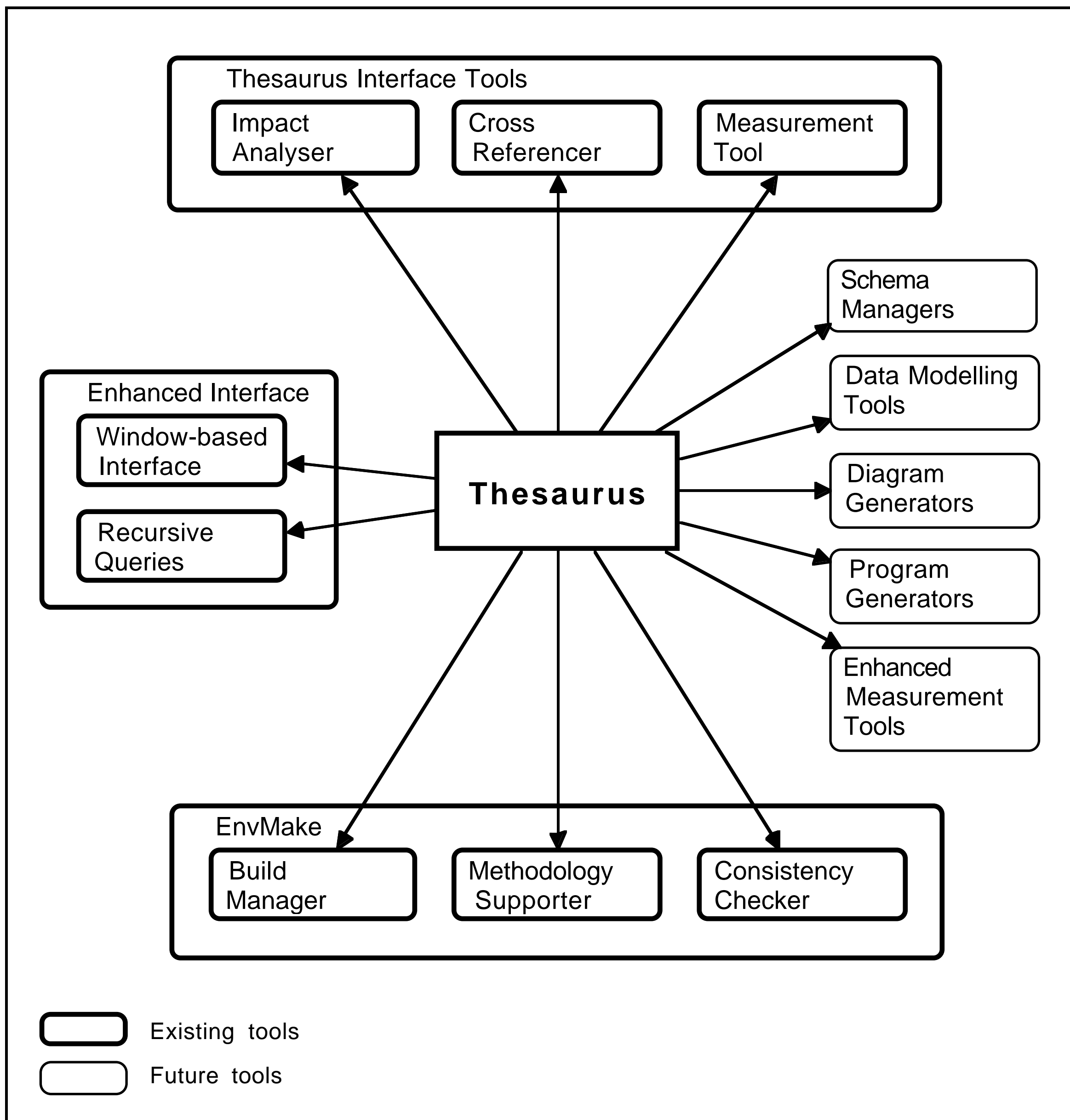

Figure 2: A thesaurus-based software environment

A whole class of tools that could utilise the thesaurus information to support change and build management, incremental schema design, visualisation, schema evolution, etc. can be envisaged in a persistent software engineering environment. Future research will emphasise change management. Supporting tools can operate at two levels [6]. First, informative systems like the thesaurus interfaces and parts of EnvMake provide application developers and maintainers with data about the existing system, its present representation and some of its dependencies. Second, more challenging to build are automatic systems that directly implement some of the steps necessary to deal with the consequences of change. Further work on automation requires more knowledge about which changes should be propagated and which absorbed. Notations to describe propagation requirements are being developed.

Our position statement is that thesauri that collect and correlate information about all names used in the *whole* processing environment of an application system form a useful platform for software development environments. For reliability and efficiency reasons, we require that all the information is automatically maintained. (Information that relies on

manual update is usually out of date.) A consequence of this requirement, at least at present, is that most of the thesaurus information relates to the implementation and operational phases since information related to earlier phases is harder to collect and analyse automatically. Ultimately, however, information related to all phases of the life cycle should be collected, e.g. the analysis tool should scan design structures.

Most of our research has been performed in the context of a persistent programming language with a sophisticated, polymorphic type system, enabling arbitrary complex and generic structures to be contained in the thesaurus. Even though our tools have been built in a closed (language dependent) environment, the ideas are generally applicable. For example: the loosely-coupled background analysis architecture, the automation of all data acquisition, the various kinds of dependency information, impact analysis, categorisation of programs according to their semantics, most of the SPASM constraints, etc. could be applied to other environments (e.g. object-oriented database applications) with only minor adaptations.

References

[1] *The Oxford English Dictionary*. Oxford University Press, London, 1961.

[2] M.P. Atkinson. "Programming Languages and Databases". In *Proceedings of the Fourth International Conference on Very Large Data Bases (Berlin, West Germany, 13th–15th September 1978),* S.B. Yao (editors), pp. 408–419, IEEE and ACM, 1978.

[3] M.P. Atkinson and O.P. Buneman. "Types and Persistence in Database Programming Languages". *ACM Computing Surveys*, Vol. 19, No. 2, pp. 105–190, 1987.

[4] M.P. Atkinson, K.J. Chisholm and W.P. Cockshott. "PS-algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices*, Vol. 17, No. 7, pp. 24–31, July 1982.

[5] M.P. Atkinson and R. Morrison. " Procedures as Persistent Data Objects". *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 539–559, 1985.

[6] M.P. Atkinson, D.I.K. Sjøberg and R. Morrison. "Managing Change in Persistent Object Systems". In *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software,* Kanazawa, Japan, November 1993.

[7] S.I. Feldman. "Make – A Program for Maintaining Computer Programs". *Software – Practice and Experience*, Vol. 9, No. 4, pp. 255–265, April 1979.

[8] D.E. Knuth. *Fundamental Algorithms*. In Series *The Art of Computer Programming*, Addison-Wesley, Vol. 1, January 1973.

[9] J.C. Lopes. ShTh – Show Thesaurus User Interface. Technical report FIDE/93/76, ESPRIT Basic Research Action, Project Number 6309 – $FIDE_2$, Computing Science Department, University of Glasgow, 1993.

[10] S. Meyers, C.K. Duby and S.P. Reiss. "Constraining the Structure and Style of Object-Oriented Programs". In *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP93)* , April 1993.

[11] R. Morrison, C. Baker, R.C.H. Connor, Q.I. Cutts and G.N.C. Kirby. "Approaching Integration in Software Environments". Submitted for publication. (Available as University of St Andrews Technical Report CS/93/10, 1993.)

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

[13] D.I.K. Sjøberg. "Quantifying Schema Evolution". *Information and Software Technology*, Vol. 35, No. 1, pp. 35–44, January 1993.

[14] D.I.K. Sjøberg. Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems. PhD Thesis, University of Glasgow, July 1993.

[15] W. Tichy. "Smart Recompilation". *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 3, pp. 273–291, July 1986.

[16] P.W. Trinder. "Comprehensions, a Query Notation for DBPLs". In *Proceedings of the Third International Workshop on Database Programming Language (Nafplion, Greece, 27th–30th August 1991),* P. Kanellakis and J.W. Schmidt (editors), pp. 55–70, Morgan Kaufmann Publishers, San Mateo, CA, 1991.