CoSET2000

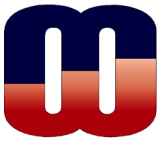
PROCEEDINGS

THE SECOND INTERNATIONAL SYMPOSIUM ON CONSTRUCTING SOFTWARE ENGINEERING TOOLS

5 June, 2000
University of Limerick
Limerick, Ireland

PRINCETON SOFTECH

University of Wollongong



MetaCase Consulting

Language Concepts

Advertisements

Language Concepts

Language Concepts is a Manchester-based startup dedicated to improving thequality of software by providing tools which manage and reduce the complexity of the software engineering process. Building on experience in CASE and MetaCASE, the company will focus on solutions which address ther eality of development in the field, providing a range of tools, components and IDE extensions which tackle problems which organisations face at different stages of the Capability Maturity Model. Our aim is to create software written "by developers for developers", working to solve the pressing practical problems developers face today.

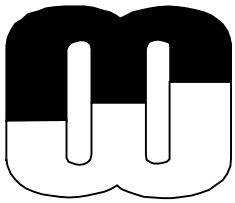
Most software engineering today occurs within the lower end of the CMM; the phrase "hack and test" is all too familiar to those who have worked in the industry. Typically, the response of CASE tool vendors to this position is to extol the benefits of a more structured and disciplined approach. Only a few companies, however, are in a position to listen, so that CASE tools, and the good practices they carry with them, have had limited acceptance.

Language Concepts will address this problem by providing tools which integrate seamlessly with existing popular IDE's such as Symantec Visual Café and Microsoft Developer Studio. These tools will provide pattern-based generation capabilities which will improve developer productivity by eliminating the need to write formulaic code, and software analysis and visualisation tools which will allow developers to take a more abstract view of their code without introducing any barriers to rapid development.

Where possible Language Concepts' tools will not introduce additional learning requirements on developers. Using the very latest user interface technologies, they will instead offer all the information needed to make best use of them as they are used. In fact, many of the planned tools exist solely to reduce the learning burden developers face by capturing knowledge of sophisticated design patterns and specialist system APIs, so that developers can be up and running quickly with new approaches and technologies.

Language Concepts will also provide integrated project management tools which give software managers – particularly those facing the special challenges of working with small, multi-project teams – the very best information they need to deliver a quality product on time and within budget. We are committed to the exploitation of the Internet, and will use Internet technologies such as XML, XSLT, HTML and HTTP wherever possible, creating products which are inherently connected and open.

Language Concepts is currently seeking investors and technology partners. For more information contact Paul Dundon at concepts@pdundon.dircon.co.uk.



MetaEdit+

Build your own CASE tool!

MetaEdit+® allows you to build your own CASE tool - without having to write a single line of code. The object-oriented method modelling along with an extensive library of reusable method components makes CASE tool development fast, real-time and cost-effective.

> *"MetaEdit+ provides a quick yet powerful way to implement CASE tool support for your own methods. Custom modelling tools can be developed in a few hours," Aarno Kansikas, ICL.*

MetaEdit+ provides simple yet powerful tools for method development. These tools allow you to define the concepts, their properties, symbols, dialogs, links to other method concepts, associated rules and generators. Method development is fast with easy-to-use form based method specification tools, drawing tools and interface painters. As soon as you define a method, or even a partial prototype, you can start to use it in MetaEdit+.

The created CASE tool supports *your* visual modelling languages, code generators, document generation and links to your application development environment (component library, simulators etc.). And this is no lightweight drawing tool: it's a full-blooded multi-user, multi-project CASE environment, running on all major platforms (Windows95/98/NT, Linux 5.2/6.X, Solaris 2, HP-UX). It has diagram, matrix and table editors, several browsers, component selection and reuse tools. It offers instant documentation of your designs to desktop publishing and the web.

> *"By implementing our own methods into MetaEdit+ we have obtained a flexible development environment which fits our needs", David Narraway, Nokia Mobile Phones.*

Even as you use your method, you can make changes to it. Existing models are automatically updated to reflect the changes you make. You can further extend your method by defining how code is generated from it, adding model analysis and checking reports, and automating linkages to external programs — compilers, simulators, documentation publishing tools etc.

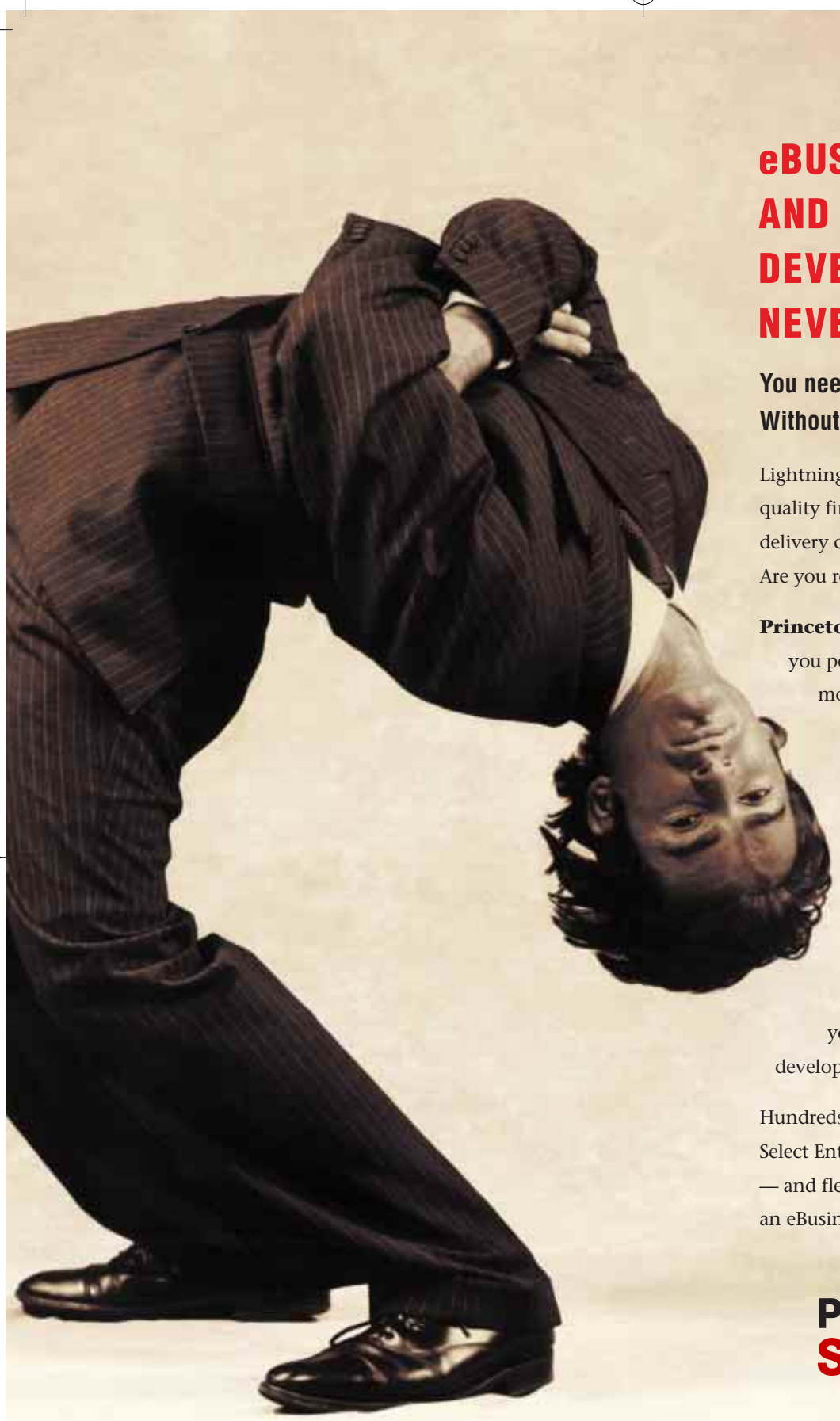
MetaEdit+ is tried and proven technology. It has been applied to build hundreds of visual modelling editors with their model analysis tools, code generators, and document generators. Many of these are supplied with MetaEdit+, making its method support the largest in the market. All these method components can be modified or reused in your own methods!

Let's build better CASE tools! Download an evaluation version of MetaEdit+ from www.metacase.com

MetaCase Consulting

Ylistönmäentie 31
FIN-40500 Jyväskylä, Finland
http://www.metacase.com

Tel +358-14-4451 400
Fax +358-14-4451 405
E-mail: info@metacase.com



eBUSINESS IS HERE. AND APPLICATION DEVELOPMENT WILL NEVER BE THE SAME.

You need speed. You need quality. Without having to bend over backwards.

Lightning-fast time to market. Mission-critical quality first time, every time. eBusiness has cut delivery cycles from 18 months down to six. Are you ready?

Princeton Softech's Select Enterprise lets you perform these minor miracles — and more. Keeping applications focused on business needs by integrating Business Process Modeling with UML. Generating code for Java, Visual Basic, C++ and Forté. Maximizing component development and reuse throughout the enterprise. "Wrappering" legacy business functions into new component based applications. And helping you leverage the power of iterative development and incremental delivery.

Hundreds of companies are already using Select Enterprise to gain the speed, quality — and flexibility — needed to succeed in an eBusiness world. How about yours?

PRINCETON SOFTECH

TIRED OF BENDING OVER BACKWARDS? VISIT princetonsoftech.com TODAY.

We've got success stories. White papers. FAQs. Everything you need to get your development team up to eSpeed. Visit our website or call +1.609.688.5000 (+44(0)1242.229.700 UK) today.

Proceedings of

The Second International Symposium on Constructing Software Engineering Tools (CoSET2000)

5 June, 2000

University of Limerick
Limerick, Ireland

Sponsored by
University of Wollongong, Australia
Princeton Softech, UK
MetaCase Consulting, Finland
Language Concepts, UK

Edited by
Ian Ferguson, Jonathan Gray, and Louise Scott

ISBN: 0 86418 725 4

Published by the

School of Information Technology and Computer Science
University of Wollongong, Australia

Copies of this publication are available at a cost of $20 each from the
School of Information Technology and Computer Science
University of Wollongong
Northfields Avenue, Wollongong, NSW 2522
Australia
Tel. +61 2 4221 3606 Fax. +61 2 4221 4170
http://www.itacs.uow.edu.au/

Table of Contents

Forward

Welcome to the Second International Symposium on Constructing Software Engineering Tools (CoSET2000). I hope you enjoy attending the symposium and I trust that you will find the presentations and discussions stimulating. This symposium has been co-located with the 22nd International Conference on Software Engineering (ICSE2000) in Limerick, Ireland, 5-11 June 2000. The inaugural CoSET symposium was held at ICSE'99 in Los Angeles, May 1999 [1]. This event received 25 submissions from authors representing 12 different countries, and 16 of these papers were selected for publication in the symposium proceedings [2]. A selection of papers from CoSET'99 were subsequently republished in a special issue of the Journal of Information and Software Technology that appeared in January 2000 [3].

CoSET2000 continues the investigation of themes and issues explored in CoSET'99, including:

- specification and generation techniques
- interchange formats and tool API's
- forward and re-/reverse- engineering tools
- tool evaluation, usability issues, and cognitive support
- tools for tool builders
- and languages, frameworks, and component-based development.
-

The symposium is based around the participants' experience reports of constructing their software engineering tools. The purpose of the symposium is to bring together an international audience of researchers and practitioners with similar interests and experience, to exchange ideas, and to learn about different technologies and techniques for software engineering tool development. The symposium focuses principally on practical software engineering issues encountered by tool developers.

For CoSET2000 we requested two categories of symposium submission:

- short papers, of typically 2000-4000 words;
- full papers, of 5000-6000 words plus figures/tables.

The Call for Papers generated 34 submissions of which 14 were accepted from authors representing 9 different countries. Papers in both categories of submission were fully refereed by the international programme committee. The accepted short and full papers are published in the Symposium Proceedings. The symposium organisers will select the most promising full papers for submission to IEEE Software for possible publication after a further process of peer review.

A successful symposium is the result of the efforts of many people, and I would like to thank all those who made this symposium possible. In particular, thanks to our referees and to our sponsors. Also thanks to the symposium participants for their attendance and interest in software engineering tool construction.

Jonathan Gray
CoSET2000, Symposium Chair
jpgray@computer.org

[1] Gray J.P., Scott L., Liu A., Harvey J. CoSET'99 Workshop Summary in *Proc. ICSE'99*, Los Angeles, USA, 16-22 May 1999, ACM (1999), 707-708

[2] Gray J.P., Scott L., Liu A., Harvey J. (eds) *Proceedings of the First International Symposium on Constructing Software Engineering Tools (CoSET'99)*, Los Angeles, USA, 16-22 May 1999, University of South Australia (1999) ISBN 0 86803 629 3.

[3] Special Issue on Constructing Software Engineering Tools. *Information and Software Technology*, Vol 42, Number 2, 25 January 2000, 71-158

Committees

Symposium Chair

Jonathan Gray, *University of Wollongong, Australia*

Organising Committee

Ian Ferguson, *University of Sunderland, UK*
Jonathan Gray, *University of Wollongong, Australia*
Louise Scott, *Fraunhofer IESE, Kaiserslautern, Germany*

Programme Committee

Albert Alderson, *Staffordshire University, UK*
Ira Baxter, *Semantic Designs, USA*
Sonia Berman, *University of Cape Town, South Africa*
David Budgen, *Keele University, UK*
Nacer Boudjlida, *UHP Nancy 1, France*
Peter Croll, *University of Wollongong, Australia*
Jean Claude Derniame, *LORIA, France*
Ian Ferguson, *University of Sunderland, UK*
Michael Godfrey, *University of Waterloo, Canada*
Jonathan Gray, *University of Wollongong, Australia*
John Grundy, *University of Auckland, New Zealand*
Jennifer Harvey, *Motorola, Australia*
Chris Harrison, *UMIST, UK*
Dirk Jäger, *RWTH Aachen, Germany*
Paul Layzell, *UMIST, UK*
Timothy Lethbridge, *University of Ottawa, Canada*
Anna Liu, *CSIRO, Australia*
Fred Long, *University of Wales, Aberystwyth, UK*
Robert Manderson, *University of Westminster, UK*
Marc Monecke, *University of Siegen, Germany*
Rick Mugridge, *University of Auckland, New Zealand*
Paddy Nixon, *Trinity College Dublin, Ireland*
David Redmiles, *University of California, Irvine, USA*
Steven Reiss, *Brown University, USA*
Peter Rösch, *TECHMATH AG, Germany*
Matti Rossi, *Helsinki School of Economics, Finland*
Reinhard Schauer, *University of Montreal, Canada*
Louise Scott, *Fraunhofer IESE, Germany*
Ewan Tempero, *Victoria University of Wellington, NZ*
Tony Wasserman, *Software Methods & Tools, USA*
Richard Webby, *Telcordia, USA*
Bernhard Westfechtel, *RWTH Aachen, Germany*
Jörg Zettel, *Fraunhofer IESE, Germany*

SESSION 1

RE-ENGINEERING, MAINTENANCE, AND CODE MANAGEMENT

Secrets from the Monster: Extracting Mozilla's Software Architecture

Michael W. Godfrey and Eric H. S. Lee
Software Architecture Group (SWAG)
Department of Computer Science, University of Waterloo
Waterloo, Ontario, N2L 3G1, CANADA
email: {migod,ehslee}@plg.uwaterloo.ca

ABSTRACT
As large systems evolve, their architectural integrity tends to decay. Reverse engineering tools, such as PBS [7, 19], Rigi [15], and Acacia [5], can be used to acquire an understanding of a system's "as-built" architecture and in so doing regain control over the system. A problem that has impeded the widespread adoption of reverse engineering tools is the tight coupling of their subtools, including source code "fact" extractors, visualization engines, and querying mechanisms; this coupling has made it difficult, for example, for users to employ alternative extractors that might have different strengths or understand different source languages.

The TAXFORM project has sought to investigate how different reverse engineering tools can be integrated into a single framework by providing mappings to and from common data schemas for program "facts" [2]. In this paper, we describe how we successfully integrated the Acacia C and C++ fact extractors into the PBS system, and how we were then able to create software architecture models for two large software systems: the Mozilla web browser (over two million lines of C++ and C) and the VIM text editor (over 160,000 lines of C).

Keywords
Interchange formats, reverse engineering, software architecture.

1 INTRODUCTION

Large software systems must evolve or they risk losing market share to competitors [11]. However, the architectural integrity of such systems often decays over time as new features are added, defects are fixed, performance is tuned, and support for new platforms is added [18, 22]. Reverse engineering tools such as PBS [7, 19], Rigi [15], and Acacia [5], can be used by developers to regain an understanding of the "as-built" software architecture of a system, and to reconcile it with the "conceptual" or intended software architecture [9]. However, most such tools are comprised of tightly coupled subcomponents, such as source code "fact" extractors and visualization engines. This tight coupling has impeded the widespread adoption of such tools, as it is difficult for users to substitute alternative subtools that might have different strengths or model different source code languages.

The TAXFORM (Tuple Attribute eXhange FORMat) project has sought to investigate how different subtools can integrated into a single framework by providing mappings to and from common data schemas for program "facts". Previous work has included the design of generic schemas for procedural and object-oriented programming languages, an exploration of problematic issues in representing and translating facts about programs, and some preliminary experiments in using the Acacia and Rigi extractors as "front-ends" to the PBS system [2].

Our primary motivation for the work described in this paper was the desire to create software architecture models of the Mozilla web browser [14]. Mozilla is written using a combination of C++ and C; however, the extractor for the PBS system, `cfx`, does not support the C++ language, and furthermore we found that it was unable to process much of the portion of Mozilla that is written in C. In this paper, we describe how we created a systematic translation mechanism to allow the integration of the Acacia fact extractors for C and C++ into the PBS system, and how we subsequently used the translators to create software architecture models for two large software systems: Mozilla (over two million lines of C++ and C code) and the VIM text editor [23] (over 160,000 lines of C code).

2 THE PBS AND ACACIA SYSTEMS

The work we describe here involves the PBS [7, 19] and Acacia [5] reverse engineering systems. The Acacia system provides facilities for extracting and visualizing low-level facts about systems written in C and C++, but it provides little automated support for creating high-level views of a system's software architecture. Acacia includes two fact extractors: `CCia` which can be configured to process C++ or C code, and the older extractor `cia` which extracts less information and works only with the C language but which we found to be more robust when applied to some C systems. The re-

sults of the extractions are stored in textual databases which can be queried at the command line or by using the CIAO visualization tool.

We have chosen to base our work around the PBS system as we have extensive experience with it, and because it provides rich support for the creation and querying of high-level views of software systems. PBS includes a special "relational calculator" language called `grok` that allows users to create customized views quickly and easily [19]. Extracted "facts" about a system are stored using a generic schema language called TA (Tuple Attribute); a user may define desired abstract relations on these facts, which the `grok` interpreter then processes, by performing the appropriate relational calculations, to create high level architectural views of the system. In this way, a user can create structured and multi-layered views of the system's software architecture which can be navigated and queried by the PBS visualization tool.

We decided to adapt the C and C++ extractors from the Acacia system for use within PBS for several reasons. Our primary motivation was the desire to create software architecture models of systems written in C++ without having to create a customized C++ fact extractor.[1] The Acacia C++ extractor, `CCia`, performs a detailed extraction of entities and relationships of C++ code[2], and it uses a production-quality front end.[3] Second, the fact extractor for PBS, `cfx`, supports only the C language and has been found to be fairly fragile; we hoped to gain an alternative extractor for the C language, and also evaluate the relative quality of each extractor.[4] And finally, we wished to explore the practical problems in translating "facts" extracted by one system for use with a different system.

3 TRANSLATING ACACIA OUTPUT INTO TA

We decomposed the task of creating a translation mechanism from Acacia into TA (PBS's format) into two stages. First, we adapted Acacia's C language extractors for use as drop-in replacements for PBS's C extractor, and then we built on this experience to create a mechanism for translating `CCia` output of C++ code into PBS. This second step also involved the creation of new `grok` scripts for modelling and visualizing object-oriented systems in PBS.

The PBS extractor `cfx` generates an intermediate format that is used by another tool, `fbgen`, to generate textual tuples (in TA format) that describe attributes of the program entities (*e.g.,* files, functions, variables, macros) and their interrelationships (*e.g.,* containment, function calls, variable references, macro uses). For example, the following TA facts are taken from an extraction of the source code for version 3.0 of the `ctags` system:

```
funcdcl read.h fileClose
funcdef read.c fileClose
funcdcl main.h getFileSize
funcdef main.c getFileSize
linkcall fileClose getFileSize
```

These TA facts assert that `fileClose` and `getFileSize` are C functions declared in `read.h` / `main.h` respectively, defined in `read.c` / `main.c` respectively, and that there is a call from `fileClose` to `getFileSize` that must be resolved by the linker. The resolution of which function calls which other function and what these relationships mean at the file and subsystem level is performed subsequent to the extraction by a set of `grok` scripts.

Acacia extraction output is stored in two semi-colon delimited plain-text databases, one for entities and one for relationships. Each entity is assigned a unique identifier (UID) by the extractor.[5] A typical entry in the entity database includes the entity's name, its UID, the UID of the containing file, its visibility, its signature/datatype (if appropriate), and whether the entity is a declaration or a definition (if the entity is a function or variable). Resolution of relationship information (*e.g.,* "which function `f` is being called by function `g`?") is performed by the extractor; a typical relationship database entry lists the details of each entity involved in the relationship (including the UIDs) together with attributes of the relationship (*e.g.,* two functions may be "friends", or one may call the other, or one may be a template instantiation of the other).

While the Acacia and PBS fact extractors perform similar tasks and are used in similar ways, there were a number of semantic discontinuities that had to be addressed. In particular, the idea of what an entity is (*e.g.,* is function declaration a distinct entity from a like-named function definition?) and how entities involved in relationships are resolved (*e.g.,* if `f` calls `g`, does `f` call the declaration or the definition of `g`, and is there also a relationship between their respective containing files?) were incompatible. For example, unlike PBS, Acacia considers declarations and definitions to be distinct entities, and they are given distinct UIDs. Also, the function call relationship described above in TA would be modelled by Acacia as a relationship between the function *definitions* in the "dot-c" files. This is subtly different from the PBS assumption and required "unfolding" some of the relationships extracted by Acacia in the conversion scripts.

[1] Creating a correct and robust parser for C++ is known to be a difficult problem due to the language's inherent complexity. By comparison, a high-quality fact extractor for the Java language was created by a member of our group in only a few days [4].

[2] We also briefly considered using two other C++ extractors: Gen++ [6] and Datrix [10]. Anecdotal evidence suggested that the Gen++ tool was relatively fragile and hard to configure, and we found that while Datrix extracts finely grained entity-level information, it does not resolve relationship references beyond the "name-level" [2].

[3] `CCia` is built around the Edison Design Group (EDG) C++ front end, a commercial product.

[4] Murphy [16] and Armstrong [1] have performed comparative analyses of several extractors.

[5] `cia` uses a simple counter to implement UIDs while `CCia` generates an eight digit hexadecimal UID using an an attribute-based hashing function.

There were two major steps in the conversion process. First, simple textual queries were made of the entity and relationship databases, and processed through `awk` and `perl` scripts to generate TA. Then, a `grok` script was used to change the semantic model of the facts to what the PBS tool was expecting.

We now discuss our experience in using these translation mechanisms on the VIM and Mozilla systems.

4 EXTRACTING VIM'S SOFTWARE ARCHITECTURE

Our first two example systems written in C that we tried out were the VIM text editor (150,000 lines of code) and its companion tool `ctags` (12,000 lines of code). The source code for VIM made `CCia` crash; we discovered that `CCia` was less robust than `cia` when applied to some C systems that used non-ANSI conventions. Consequently, we also added support for the older `cia` extractor, although it extracts less information and with a different output format than `CCia`.

The fact extraction and conversion of `ctags` was straightforward, although it revealed some internal problems with the `CCia` extractor. We found that the `CCia` extractor sometimes created multiple UIDs for the same entity. While this might seem benign, it proved to be troublesome; when a function declaration had multiple UIDs, some relations were resolved incorrectly. Once we discovered this problem, we were able to work around it by discarding the `CCia` UIDs and using our own "name mangling" convention within a grok script to work out entity resolutions correctly. In so doing, we found our results still differed from the `cfx` extraction, we discovered several subtle bugs in how PBS performs "linking" (entity resolution) that have since been fixed.

Results for VIM

We performed a full extraction on version 5.6 of the VIM editor using both `cfx` and `cia`, and then we translated the `cia` facts into TA format using our scripts. The `cia` extraction was faster, but when combined with the translation time, the total was slightly more than that for the `cfx` extraction. The total time for both approaches was slightly faster than a full compile of the system.[6]

The full distribution of version 5.6 of VIM, which includes the companion utility `ctags`, comprises over 163,000 lines of C code (including comments and blank lines). The breakdown of the distribution into header files (`.h` and `.pro` files) and implementation files (`.c` files) is shown below:

File type	Total # of files	Total KLOC
`.h`	35	8,051
`.pro`	47	1,316
`.c`	67	154,360
TOTAL	149	163,727

Unlike Mozilla, almost all of the source code files are included in a typical compile. We found that the breakdown of the system into source files was primarily based on functionality and features; while VIM can be compiled to run on a variety of platforms, most of the platform-specific code is distributed throughout the various source files.

We found that a `cfx` extraction of VIM (ignoring `ctags`) produced over 43,000 "facts".[7] Performing a analogous extraction using `cia` plus our translation scripts produced over 51,000 facts. Comparing the two extractions in detail, we found several notable differences:

- `cia` (and `CCia`) perform macro expansion to extract more detailed relationship information. For example, if a function `f` calls a macro `m` that in turn expands to a call to a function `g`, then both Acacia extractors will record that `f` uses macro `m` *and* that `f` calls function `g`. `cfx` does not perform this level of analysis. This was the primary source of "extra" facts extracted by `cia`.
- We added some extra detail that `cia` extracts but `cfx` does not model, including references to library variables, such as `__ctype` and `errno`.
- `cia` does a more accurate extraction of function call information than `cfx`. We found that `cfx` missed a number of straightforward function calls that `cia` found.
- A fairly common programming convention in C is to define a macro named `EXTERN` that precedes function and variable declarations in "`.h`" files. This macro expands to the keyword `extern` in all implementation files that use (but do not define) the entity, and expands to the empty string in the implementation file that defines the entity. We found that `cfx` was able to model this convention correctly, but that `CCia` did not.

In summary, we found that we were able to successfully adapt `cia` and `CCia` into high quality C extractors for the PBS system with performance similar to that of the native PBS C extractor. With the exception of the `EXTERN` problem, we were able to adjust for all of the semantic inconsistencies and other problems using `grok` scripts.

Observations about VIM

Figure 1 shows a top level view of the software architecture model for VIM. This model was created using a variety of

[6] On a Sparc running Solaris 2.6 with four 300MHz processors and 1 gigabyte of memory, the `cfx` extraction took 4:27 minutes, the `cia` extraction took 1:52 minutes, the translation of the `cia` output to TA took 3:20 minutes, and a full compile of VIM took 6:29 minutes.

[7] A total of 30 kinds of facts were extracted for the C language model, including `funcdef`, `usemacro`, and `include`. Precise details of the schemas for entities and relationships extracted by `cfx` can be found elsewhere [2, 7, 19].

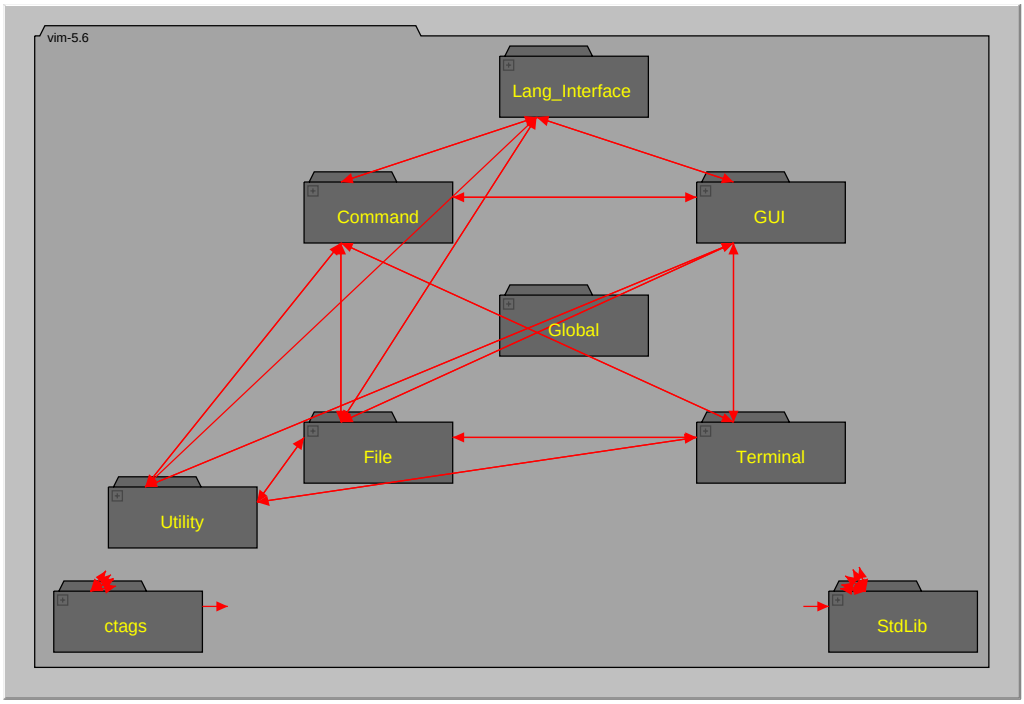


Figure 1: Top level view of the extracted architecture of VIM as shown by the PBS viewer. Folder icons denote subsystems, and arrows denote function calls between subsystem members (some calls are elided and are shown as arrow stubs). The subsystems are described briefly in Fig. 2.

Subsystem name	**# of contained source files**	**Total KLOC**	**Description**
Command	29	55	User command processing
File	16	20	File I/O and buffer manipulation
Lang_Interface	9	6	Interface to prog. langs. (*e.g.*, Perl, Python, tcl)
Global	15	5	Contains global variables, data structures defs, *etc.*
GUI	21	33	User interface code
Terminal	4	5	Mappings for kbd/mouse
Utility	10	14	Implements regexps, message routines, *etc.*
Ctags	36	18	VIM's companion tool
Stdlib	303	72	System `include` files (*i.e.*, not part of distribution)
TOTAL *(all)*	443	228	
(ignoring Stdlib)	140	156	

Figure 2: The major subsystems in our architectural model of VIM version 5.6, as shown in Fig. 1. This model includes only the code that was used during a typical compile of VIM for the Linux operating system running on an Intel 686 processor.

knowledge sources including the system documentation, domain knowledge about text editors, a detailed examination of source code, and the authors' extensive experience in using VIM.

It is not our intention to discuss VIM's software architecture in detail in this paper, as we do so elsewhere [22]; however, we do note some general observations. First, we discovered that VIM has been implemented using a repository-style software architecture [20]. The data structures that implement the buffer being edited are globally accessible variables defined within the *Globals* subsystem; this explains why there are no function call arrows going into or out of the *Globals* subsystem in Fig. 1.

Another result that we found to be surprising was that the *Utility* subsystem had functional dependencies on other subsystems. Upon closer examination, we found that most of these unexpected dependencies were contributed by two large files `misc1.c` and `misc2.c` comprising over 5700 LOC and 2400 LOC respectively. As their names suggest, they contain a variety of unrelated functions; we found comments within the code such as *"Various functions"* and *"functions that didn't seem to fit elsewhere"* that confirmed our hypothesis. Our subsequent "repair" of VIM's architecture resulted in moving many of these functions to other files in other subsystems [22].

5 EXTRACTING MOZILLA'S SOFTWARE ARCHITECTURE

We next considered how to create a software architecture model of the Mozilla web browser using the Acacia extractor and the PBS system. Mozilla is the "open source" subset of the Netscape browser [14, 17]. It is a huge, multi-function, multiplatform system comprising over two million lines of C++ and C code in the release version we examined (Milestone-9 or "M9").

We rewrote our translation scripts to use an object-oriented language schema; the schemas we created comprised 71 kinds of facts (compared to 24 for the procedural C model) [12]. We created additional infrastructure for the PBS system to be able to create and navigate through software architecture models of object-oriented systems, which consisted mostly of `grok` scripts and data files used by the PBS viewer.

The biggest challenge in creating these scripts was in distinguishing between entities that might have the same name. In C, "name collisions" between globally visible entities are fairly rare, but in C++ they are much more common due to overloading, polymorphism, use of templates *etc*. We used a more complex "name mangling" scheme than we had used with the C scripts; we did not use Acacia's UIDs since, as mentioned above, `CCia` sometimes generated spurious extra UIDs for some entities.

Initial attempts at fact extraction led us to rewrite our translation scripts yet again, as we found the performance to be unacceptable; our approach with VIM has been to use simple minded `awk` scripts to transliterate the Acacia facts into TA using a series of queries, and then perform "intelligent" translation using `grok`. We found we had to read the entire Acacia databases into a large associative array and then generate the "naive" TA facts in one go.[8]

Results for Mozilla

As mentioned above, Mozilla release M9 consists of over two million lines of C++ and C code. The source distribution of C and C++ header and implementation files breaks down as shown below:

File type	Total # of files	Total KLOC
`.h`	4,531	610
`.c`	811	434
`.cpp`	2,079	1,043
TOTAL	7,421	2,087

Total KLOC denotes thousands of lines of source code including comments and blank lines. This count includes all source files for all supported platforms in the source distribution, but does not include header files that are generated automatically during a system build. Using the utility `ctags`, we calculated that there are over 2,500 classes, 33,000 class methods, 18,000 class/struct/union data members, 11,000 global ("`extern`") functions, and 3,500 global ("`extern`") variables in the source code contained in the `tar` file distribution.

Because Mozilla is multiplatform, a large part of its distributed code base consists of parallel sets of platform-specific implementation files [8]. In order to perform an analysis of the relationships within a typical instantiation of Mozilla, it made sense to construct an architectural view of one build of the system. We therefore compiled Mozilla for Linux, and found that it processed 192 of the 811 "`.c`" files and 1319 of the 2079 "`.cpp`" files. We then used a trace of the build process to decide which files to extract facts from.

We used the C++ option of the `CCia` extractor for both the C++ and the C portions of Mozilla. We had considered using the `CCia`'s C extraction option for the C code, but we decided that it would be too awkward to generate two sets of databases with different schemas and different translation mechanisms that then had to be reconciled into a coherent whole. The use of the C++ option required some manual adjusting of the C code to account for the stronger type checking rules of C++; in particular, many C implementation files were edited to add explicit type casting (this approach did not work so easily for macros that take parameters). Additionally, we discovered that the commercial front end used by `CCia` did not recognize the `static const` construct of C++.

[8] Godfrey wrote the original C translation scripts in `awk`; Lee reimplemented them for C++ using `perl`.

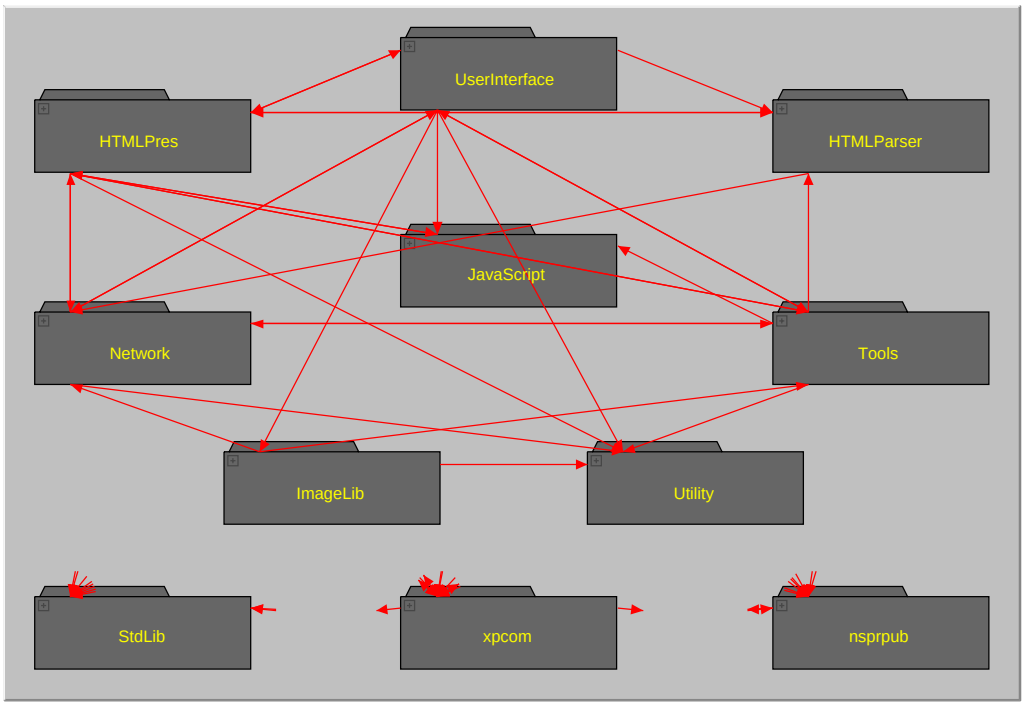


Figure 3: Top level view of the extracted architecture of Mozilla as shown by the PBS viewer. Folder icons denote subsystems, and arrows denote function calls between subsystem members (some calls are elided and are shown as arrow stubs). The subsystems are described briefly in Fig. 4.

Subsystem name	**# of contained subsystems**	**# of contained source files**	**Total KLOC**	**Description**
HTMLPres	47	1,401	484	HTML layout engine
HTMLParser	8	93	42	HTML parser
ImageLib	5	48	15	Image processing library
JavaScript	4	134	47	JavaScript engine
Network	13	142	31	Networking code
StdLib	12	250	45	System `include` files (*i.e.*, "`.h`" files)
Tools	47	791	269	Major subtools (*e.g.*, mail and news readers)
UserInterface	32	378	147	User interface code (widgets, *etc.*)
Utility	4	60	35	Programming utilities (*e.g.*, string libraries)
nsprpub	5	123	51	Platform independent layer
xpcom	23	224	63	Cross platform COM-like interface
TOTAL	200	3,650	1,229	

Figure 4: The major subsystems in our architectural model of Mozilla release M9, as shown in Fig. 3. This model includes only the code that was used during a typical compile of Mozilla for the Linux operating system running on an Intel 686 processor.

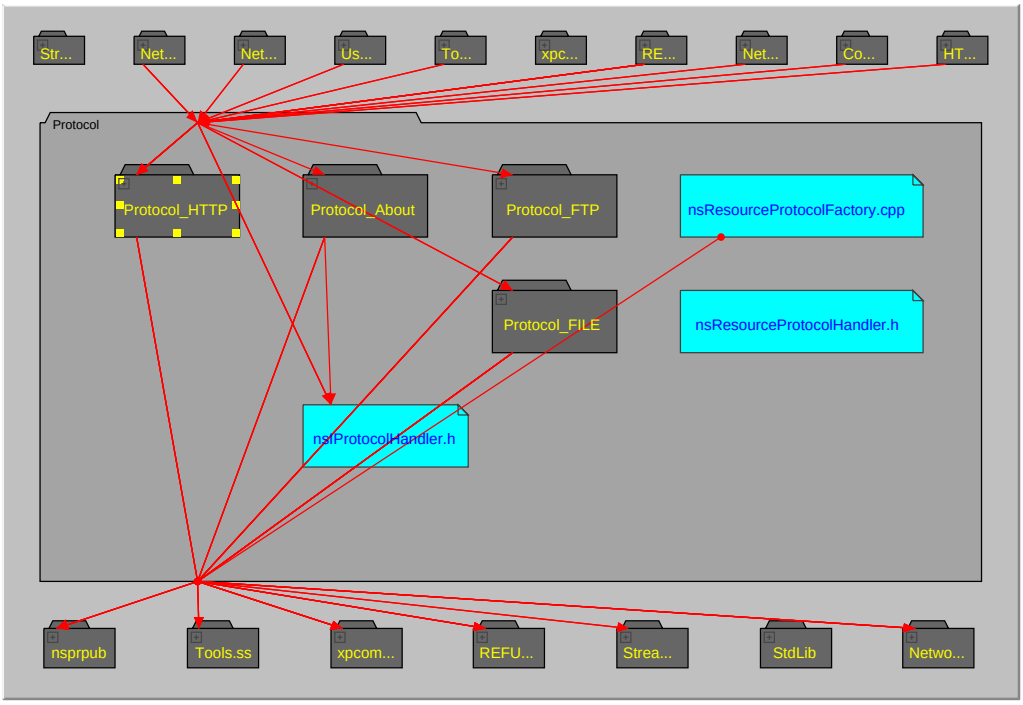


Figure 5: View of the Protocol subsystem of Mozilla (a member of the top-level Network subsystem) as shown by the PBS viewer. Folder icons denote subsystems, document icons denote source files, and arrows denote function calls.

The manual adjustment of code was laborious and time consuming. Eventually, we decided that 23 of the 1511 files were too difficult to fix without an enormous effort in restructuring and program understanding. However, we note that we still managed to process more than 98% of the files in the system.

A full source build of Mozilla M9 on a dual processor Pentium-III 450 MHz system with 512 megabytes of RAM running Redhat Linux 6.1 took 35 minutes. The `CCia` extraction took three and a half hours, and the translation into TA using our scripts took another three hours on the same system. The extraction generated over 990,000 facts, taking up over 133 megabytes of disk space (uncompressed). We note that the total extraction time is still much less than the amount of time we spent editing the source code so that the extractor would be able to process it.

Observations About Mozilla

We created the subsystem hierarchy of our software architecture model based on several sources of information, including the source directory structure, examining the extracted facts, the use of an automated subsystem clustering tool [19], reading through the source code and documentation, and browsing the Mozilla website [14]. Our architecture model contains 11 top-level subsystems, as shown in Fig. 3 and Fig. 4; of these, the largest were concerned with HTML layout, the implementation of subtools such as the mail and news readers, and user interface code. Figure 5 shows a typical intra-subsystem view as shown by the PBS viewer/navigator.[9]

As with VIM, we do not discuss Mozilla's software architecture in detail in this paper, as we do so elsewhere [13]; however, we also note some general observations. First, our in-depth examination of Mozilla leads us to conclude that either its architecture has decayed significantly in its relatively short lifetime, or it was not carefully architected in the first place. For example, the top-level view of Mozilla's architecture resembles a near-complete graph in terms of the dependencies between the different subsystems (Fig. 3 shows the function call dependencies); while we might reasonably expect function calls from the user interface subsystem to most other subsystems, we were surprised to see functional dependencies from the image processing library to the network and tools subsystems. Overall, we found the architectural coherence of Mozilla to be significantly worse than that of other large open source systems whose software architectures we have examined in detail (Linux and VIM) [3, 21, 22].

However, we do not consider these results to be surprising, as Netscape was among the first generation of web browsers; it is well known that competition during the "browser wars"

[9] These figures show only function call relations at the file and subsystem level; other information can also be shown by the viewer, such as variable references and class inheritance. Additionally, the architecture views can be navigated hierarchically as well as queried.

has been intense. Netscape and its main competitor, Microsoft's Internet Explorer, have evolved extremely rapidly over the last few years, leading not only to an abundance of new features, but also to a very large number of "bugfix" releases and a notorious reputation for unreliability. Mozilla seems to be a telling example of Lehman's laws of software evolution, which state that a useful software system must undergo continual and timely change or it risks losing market share [11].

6 SUMMARY

In this paper, we have described our experiences in extending the work of the TAXFORM project [2]. We have created automated mechanisms for converting the output of Acacia's C and C++ extractors into generalized textual schemas for procedural and object-oriented languages using the TA notation. We also described our experiences in using these mechanisms in the creation of software architecture models for two large software systems: the Mozilla web browser (over two million lines of C++ and C code) and the VIM text editor (over 160,000 lines of C code).

We have undertaken this work for several reasons: to investigate the practical issues involved in transforming extracted data between abstract schemas; to allow the creation of navigable high-level software architecture models for systems written in C++; and to explore the relative differences between the two reverse engineering systems. We found that we were able to successfully adapt the Acacia extractors for use in the PBS system, and that the conversion of extracted facts is straightforward once a suitable translation mechanism is in place. We note that, as observed by others [1, 16], the robustness of the extractors and quality of the extracted facts varies between tools, and that it is sometimes necessary to "tweak" the source code of the system being examined in order to get the extractor to process it correctly. Finally, we consider that this work represents a significant data point in the quest for seamless data exchange between reverse engineering environments.

REFERENCES

[1] Matt Armstrong and Chris Trudeau, "Evaluating Architectural Extractors", *Proc. of the 1998 Working Conference on Reverse Engineering* (WCRE-98), Honolulu HI, October 1998.

[2] Ivan Bowman, Michael W. Godfrey, and Ric Holt, "Connecting Architecture Reconstruction Frameworks", *Proc. of CoSET'99 — Symposium on Constructing Software Engineering Tools*, May 19999. Also published in *Journal of Information and Software Technology*, vol. 42, no. 2, February 2000.

[3] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture", *Proc. of the* 21^{st} *Intl. Conf. on Software Engineering* (ICSE-21), Los Angeles CA, May 1999.

[4] Ivan Bowman, Michael W. Godfrey, and Ric Holt, "Extracting Source Models from Java Programs: Parse, Disassemble, or Profile?", in preparation.

[5] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios, "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection", *IEEE Trans. on Software Engineering*, vol. 24, no. 9, September 1998.

[6] P. Devanbu, "A Language and Front-end Independent Source Code Analyzer Generator", *Proc. of the* 14^{th} *Intl. Conf. on Software Engineering* (ICSE-14), 1992.

[7] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The Software Bookshelf," *IBM Systems Journal*, vol. 36, no. 4, November 1997.

[8] Michael W. Godfrey and Qiang Tu, "Evolution of Open Source Software: A Case Study", submitted for publication, available from `http://plg.uwaterloo.ca/~migod/papers/`.

[9] Christine Hofmeister, Robert Nord, and Dilip Soni, *Applied Software Architecture*, Addison-Wesly Longman Inc., Reading MA, 2000.

[10] B. Laguë, C. Leduc, A. Le Bon, E. Merlo, and M. Dagenais, "An Analysis Framework for Understanding Layered Software Architectures", *Proc. of the 1998 Intl. Workshop on Program Comprehension* (IWPC '98), Ischia, Italy, June 1998.

[11] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski, "Metrics and Laws of Software Evolution — The Nineties View", *Proc. of the Fourth Intl. Software Metrics Symposium* (Metrics'97), Albuquerque NM, 1997.

[12] Eric H. S. Lee, "The Software Bookshelf for Mozilla", website, `http://swag.uwaterloo.ca/~ehslee/pbs/mozilla/R2/`.

[13] Eric H. S. Lee, "Mozilla: Its Extracted Software Architecture", in preparation.

[14] "The Mozilla Homepage", website, `http://www.mozilla.org/`.

[15] Hausi Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification", *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, December 1993.

[16] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S-C. Lan, "An Empirical Study of Static Call Graph Extractors", *ACM Transactions on Software Engineering and Methodology* (TOSEM), vol. 7, no. 2, 1998.

[17] “The Open Source Homepage”, website, `http://www.opensource.org/`.

[18] David Parnas, “Software Aging”, *Proc. of the* 16^{th} *Intl. Conf. on Software Engineering* (ICSE-16), Sorrento, Italy, May 1994.

[19] “The PBS Homepage”, website, `http://www-turing.cs.toronto.edu/pbs/`.

[20] Mary Shaw and David Garlan, *Software Architecture: Perspectives of an Emerging Discipline*, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[21] John B. Tran and R.C. Holt, “Forward and Reverse Repair of Software Architecture”, *Proc. of CASCON 1999*, Toronto, November 1999.

[22] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt, “Architecture Analysis and Repair of Open Source Software”, to appear in *2000 Intl. Workshop on Program Comprehension* (IWPC’00), Limerick, Ireland, June 2000.

[23] “The VIM Homepage”, website, `http://www.vim.org/`.

MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems

Stéphane Ducasse, Michele Lanza, Sander Tichelaar
Software Composition Group, University of Berne
Neubrückstrasse 12, CH – 3012 Berne, Switzerland
{ducasse,lanza,tichel}@iam.unibe.ch — http://www.iam.unibe.ch/∼scg/

Abstract

Surprising as it may seem, many of the early adopters of the object-oriented paradigm already face a number of problems typically encountered in large-scale legacy systems. The reengineering of those systems often poses problems because of the considerable size and complexity of such systems. In the context of the FAMOOS project we have developed a language independent environment called Moose which can deal with that complexity. This paper describes the architecture of Moose, the tools which have been developed around it and the industrial experiences we have obtained.

Keywords: *Reengineering, Reverse Engineering, Refactoring, Software Metrics, Object-Oriented Programming*

1 Introduction

Legacy systems are not limited to the procedural paradigm or languages such as COBOL. Although the object-oriented paradigm promised increased flexibility of systems and the ease in their evolution, even these systems get hard to maintain over time and need to be adapted to new requirements. The goal of the FAMOOS Esprit project was to support the evolution of such object-oriented legacy systems towards frameworks [6].

During the FAMOOS project we built a tool environment called MOOSE to reverse engineer and reengineer object-oriented systems. It consists of a repository to store models of software systems, and provides query and navigation facilities. Models consist of entities representing software artifacts such as classes, methods, etc. MOOSE has the following characteristics:

- It supports reengineering of applications developed in different object-oriented languages, as its core model is *language independent* which, if needed, can be *customized* to incorporate language specific features.
- It is *extensible*. New entities like measurements or special-purpose relationships can be added to the environment.
- It supports reengineering by providing facilities for analyzing and storing multiple models, for refactoring and by providing support for analysis methods such as metrics and the inference of properties of source code entities.
- Its implementation being fully object-oriented, MOOSE provides a complete description of the meta-model entities in terms of objects that are easily parameterized and/or extended.

These properties make MOOSE an ideal foundation for reengineering tools [3].

The outline of this paper is the following: Before presenting the specific aspects of MOOSE, we list the main characteristics that we expect from a reengineering environment. After presenting the architecture of MOOSE, we give an overview of its underlying meta-model and interchange format. We present how a modelled system can be navigated and queried. Then we show how MOOSE supports code refactorings. To give a more dynamic perception of MOOSE we show a typical use in the form of a short scenario. Finally we evaluate the environment regarding the requirements we previously listed and conclude.

2 Requirements for a Reengineering Environment

Based on our experiences and on the requirements reported in the literature [12, 8, 9], these are our main requirements for a reengineering environment:

Extensible. An environment for reverse engineering and reengineering should be extensible in many aspects:

- The meta-model should be able to represent and manipulate entities other than the ones directly extracted from the source code (e.g. measurements, associations, relationships, etc.).
- To support reengineering in the context of software evolution the environment should be able to handle several source code models simultaneously.
- It should be able to use and combine information from various sources, for instance the inclusion of tool-specific information such as run-time information, metric information, graph layout information, etc.
- The environment should be able to operate with external tools like graph drawing tools, diagrammers (e.g. Rational Rose) and parsers.

Exploratory. The exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities. The environment should be able to present the source code entities in many views, both textual and graphical, in little time. It should be possible to perform several types of actions on the views the tools provide such as zooming, switching between different abstraction levels, deleting entities from views, grouping entities into logical clusters, etc. The environment should as well provide a way to easily access and query the entities contained in a model. To minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct linkage to its source code. A secondary requirement in this context is the possibility to maintain a history of all steps performed by the reengineer and preferably allow him to return to earlier states in the reengineering process.

Scalable. As legacy systems tend to be huge, an environment should be scalable in terms of the number of entities being represented, i.e. at any level of granularity the environment should provide meaningful information. An additional requirement in this context is the actual performance of such an environment. It should be possible to handle a legacy system of any size without incurring long latency times.

In addition to these general requirements, the context of our work [6] forces us to have an environment that is able to support multiple languages.

3 Architecture

MOOSE uses a layered architecture (see Figure 1). Information is transformed from source code into a source code model. The models are based on the FAMIX meta-model [4, 5] which is described in section 4. The information in this model, in the form of entities representing the software artifacts of the target system, can be analyzed, manipulated and used to trigger code transformations by means of refactorings. We will describe the architecture of MOOSE starting from the bottom.

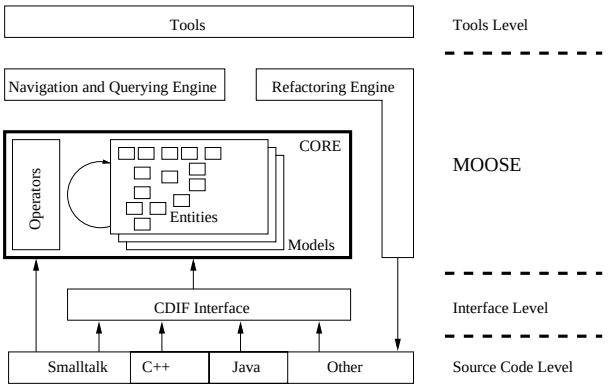


Figure 1. Architecture of Moose.

- *Extraction/Import.* MOOSE supports multiple languages. Source code can be imported into the meta-model in two different ways:

  1. In the case of VisualWorks Smalltalk – the language in which MOOSE is implemented – sources can be directly extracted via the meta-model of the SMALLTALK language.
  2. For other source languages MOOSE provides an import interface for CDIF files based on our FAMIX meta-model. CDIF is an industry-standard interchange format which enables exchanging models via files or streams. Over this interface MOOSE uses external parsers for source languages other than SMALLTALK. Currently C++, JAVA, ADA and other SMALLTALK dialects are supported.

- *Storage and Tools.* The models are stored in memory. Every model contains entities representing the software artifacts of the target system. Every entity is represented by an object, which allows direct interaction and querying of entities, and consequently an easy way to query and navigate a whole model. MOOSE can maintain and access several models in memory at the same time.

  Additionally the core of MOOSE contains the following functionality:

  – *Operators.* Operators can be run on a model to compute additional information regarding the

software entities. For example, metrics can be computed and associated with the software entities, or entities can be annotated with additional information such as inferred type information, analysis of the polymorphic calls, etc. Basically any kind of information can be added to an entity.

  - *Navigation facilities.* On top of the MOOSE core we have included querying and navigation support. This support is discussed in section 5.
  - *Refactoring Engine.* The MOOSE REFACTORING ENGINE defines language-independent refactorings. The analysis for a code refactoring is based on model information. The code manipulation which a refactoring entails, is being handled by language-specific front-ends. Section 6 describes the engine in more detail.

- *Tools Layer.* The functionality which is provided by MOOSE can be used by tools. This is represented by the top layer of figure 1. Some examples of tools based on MOOSE are described in section 7.

4 A Language Independent Meta-model

MOOSE is based on the FAMIX meta-model [4, 5]. FAMIX provides for a language-independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our tools. It is *language independent*, because we need to work with legacy systems in different implementation languages (C++, JAVA, SMALLTALK, ADA). And it is *extensible*: since we cannot know in advance all information that is needed in future tools, and since for some reengineering problems tools might need to work with language-specific information (e.g. to analyse include hierarchies in C++), we allow for language plug-ins that extend the model with language-specific features. Next to that, we allow tool plug-ins to extend the model to store, for instance, analysis results or layout information for graphs. Figure 2 shows the core of the FAMIX model. It consists of the main object-oriented entities, namely Class, Method and Attribute. In addition there are the associations InheritanceDefinition, Invocation and Access. An Invocation represents a Method calling another Method and an Access represents a Method accessing an Attribute. These abstractions are needed for reengineering tasks such as dependency analysis, metrics computation and reorganisation operations. The complete model consists of much more information, i.e. more entities such as functions and formal parameters, and additional relevant information for every entity. The model does not contain any source code. The complete specification of the model can be found in [5].

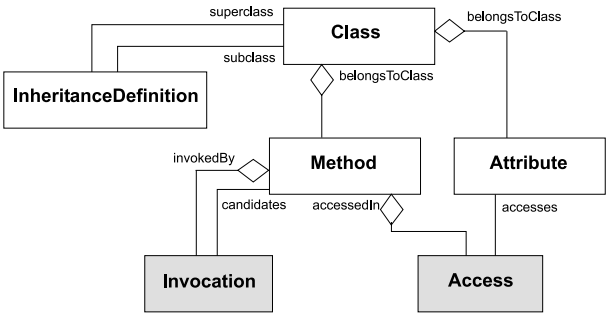


Figure 2. Core of the FAMIX model.

Information exchange with CDIF

To exchange FAMIX-based information between different tools we have adopted CDIF [2] . CDIF is an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF are, that firstly it is an industry standard, and secondly it has a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF framework supports the extensibility we need to define our model and plug-ins. As shown in Figure 1 we use CDIF to import FAMIX-based information about systems written in JAVA, C++ and other languages. The information is produced by external parsers such as SNiFF+ [15, 16]. Next to parsers we also have integrations with external environments such as the Nokia Reengineering Environment [6].

5 Navigation and Querying

One of the challenges when dealing with complex meta-models is how to support their navigation and facilitate easy access to specific entities. In the following subsections we present two different ways of querying and inspecting source code models in MOOSE.

5.1 Programming Queries

The fact that the meta-model in MOOSE is fully object-oriented together with the facilities offered by the Smalltalk environment, it is simple to directly query a model in MOOSE. We show two examples. The first query returns all the methods accessing the attribute name of the class Node.

```
(MSEModel currentModel
   entityWithName: #'Node.name')
      accessedByCollect:
         [ :each | MSEModel currentModel
             entityWithName: each accessedIn ]
```

The second query select all the classes that have more than 10 descendants.

```
MSEModel currentModel allClassesSelect:
  [ :each | each hasProperties and:
     [ (each hasPropertyNamed: #WNOC) ifTrue:
        [(each getNamedPropertyAt: #WNOC) > 10]]]
```

Note that these queries resemble SQL queries on model information stored in a database [10]

5.2 Querying using the MOOSE EXPLORER

Reengineering large systems brings up the problem of how to navigate large amounts of complex information. Well-known solutions are code browsers such as the Smalltalk one, which have been sufficient to support code browsing, editing and navigating a system by the way of senders and implementers. However, for reengineering these approaches are not sufficient because:

- The number of potentially interesting entities and their interrelationships is too large. A typical system can have several hundreds of classes which contain in turn several thousands of methods, etc.
- All entities need to be navigable in a *uniform way*.
  - In the context of reengineering no entity is predominant. For example, attribute accesses can be extremely important to analysis methods but in other cases completely irrelevant.
  - In presence of an extensible meta-model, the navigation schema should take into account the fact that new entities and relationships can be added and should be navigable as well.

MOOSE EXPLORER proposes an uniform way to represent model information (see figure 3). All entities, relationships and newly added entities can be browsed in the same way. From top to bottom, the first pane represents a current set of selected entities. Here we see all the attributes of the current model. The bottom left pane represents all the possible ways to access other entities from the currently selected ones. Here, from the selected attribute name of the class Node the methods that access it are requested. The resulting entities are displayed in the right bottom pane and can then be further browsed. 'Diving' into the resulting entities puts them as the current selection in the top pane again, which allows for further navigation through the model.

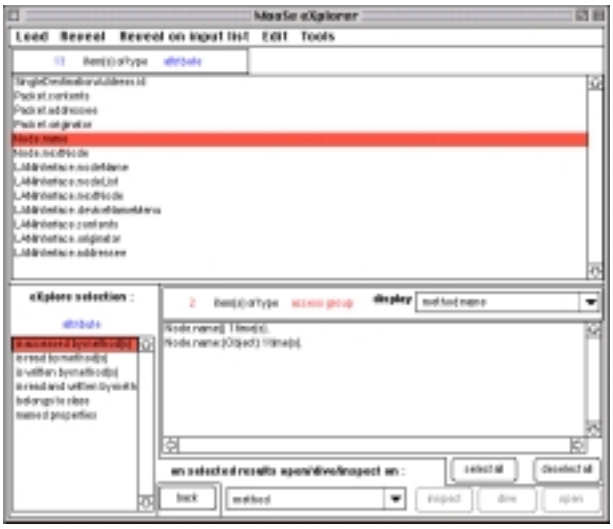


Figure 3. MOOSE EXPLORER: navigating a meta-model in an uniform way.

6 Refactoring

The MOOSE REFACTORING ENGINE closes the reengineering circle. While the MOOSE core provides for a repository and querying and navigation support, the MOOSE REFACTORING ENGINE provides support for doing actual code changes. Refactoring [7] is about making changes to code to improve its structure, simplicity, flexibility, understandability or performance [1] without changing the external behaviour of the system. The MOOSE REFACTORING ENGINE provides functionality similar to the Refactoring Browser [14] for Smalltalk, but for multiple implementation languages.

The MOOSE REFACTORING ENGINE does virtually all of the analysis — needed to check the applicability of a refactoring and to see what exactly has to be changed — using the language-independent FAMIX model. The language dependence can be kept on a minimal level, because firstly the refactorings are very similar for the different languages, and secondly, FAMIX is designed to capture these commonalities as much as possible. For instance, FAMIX supports multiple inheritance, which covers Smalltalk's single inheritance, C++'s multiple inheritance and Java's classes and interfaces. Language extensions (see section 4) cover most of the remaining issues, for instance, to figure out if a class entity in MOOSE represents a class or an interface in Java.

Of course, changing the code is language-specific. For every supported language a component has to be provided that performs the actual code changes directly on the source code. Currently the MOOSE REFACTORING ENGINE is a prototype with language front-ends for Smalltalk and Java. For Smalltalk we use the Refactoring Browser [14] to change the code, and for Java we currently use a text-based approach based on regular expressions. Although the text-based approach is more powerful than we initially expected, we plan to move to an abstract syntax tree based approach in the future.

A set of language-independent refactorings together with

the analysis support of MOOSE itself provides for a powerful combination of using analysis to drive (semi-)automated code improvements. This is illustrated by the scenario in section 8.

7 Foundation for other tools

MOOSE serves as a foundation for other tools. It acts as the central repository and provides services such as metric computation and refactorings to the reengineering tools built on top of MOOSE. At this point in time the following tools have been developed:

- CODECRAWLER supports reverse engineering through the combination of metrics and visualization [11, 3] (see Figure 4). Through simple visualizations which make extensive use of metrics, it enables the user to gain insights in large systems in a short time. CODECRAWLER is a tool which works best when we approach a new system and need quick insights to get information on how to proceed. CODECRAWLER has been successfully tested on several industrial case studies.
- GAUDI [13] combines dynamic with static information. It supports an iterative approach creating views which can be incrementally refined by extending and refining queries on the repository, while focusing on dynamic information.

The following tools are currently under development:

- The MOOSE REVEALER is used to detect entities which fulfill certain properties. At the basic level these may be abstract classes, empty methods, etc. At a higher level of complexity it addresses design problems such as unused attributes or big classes which could be split by identifying clusters of methods or attributes.
- The MOOSE FINDER is a tool that allows to compose queries based on different criteria like entity type, properties or relationships, etc. A simple query finds entities that meet certain conditions. Such a query can in turn be combined with other queries to express more complex ones. The MOOSE FINDER is currently being extended in order to handle multiple models in the context of software evolution.
- The MOOSE DESIGN FILTER can use the meta-model information to communicate with Rational Rose, in order to generate design views on the code.

Not only does MOOSE serve as the base for all those applications providing them a number of functionalities like the metrics framework, the repository also serves as common interface between those tools.

Except for providing the foundation for our own tools, MOOSE also interfaces with external tools. One example is the Nokia Reengineering Environment [6].

8 Scenario

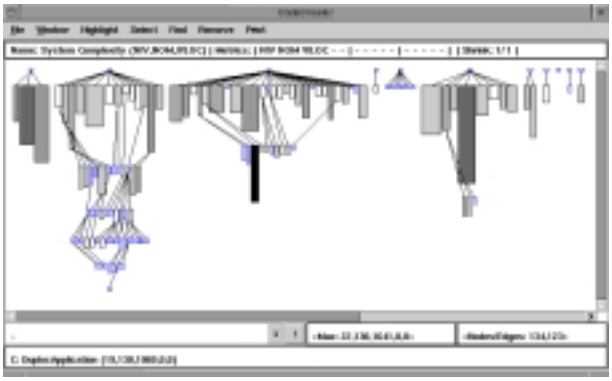


Figure 4. CODECRAWLER **showing an inheritance tree view of a system. The width of the nodes represents the number of methods, the height represents the number of instance variables.**

In this section we present a typical scenario of how the MOOSE environment can be used. It shows three different tools based on MOOSE, and their interaction to detect a problem, analyze it and finally resolve it by changing the code. Note that the scenario is partly hypothetical in the sense that the MOOSE REVEALER is in its early stages of development and that its capabilities are not yet tested in real world cases.

We start with CODECRAWLER. Figure 4 shows a screenshot of this tool. In this case the bigger boxes denote bigger classes in the inheritance hierarchy. The classes are bigger, in terms of number of methods (x-axis) and number of attributes (y-axis). In this way CODECRAWLER points us to possible problems in a software system, as big classes might imply a wrong distribution of responsibilities. We will focus on the tall gray class on the right side of the drawing.

In the second phase we use the MOOSE REVEALER to analyze our possible problem. In this case the MOOSE REVEALER finds out that the class can be split in two pieces, because it finds two groups of methods that have a strong internal cohesion, but do not really depend on the other group. The MOOSE REVEALER proposes the user to split the class in a superclass and a subclass, both with one group of methods. If the user decides that the proposed solution

is a good idea, he or she can trigger the MOOSE REFACTORING ENGINE to implement the proposed change. The MOOSE REFACTORING ENGINE initiates a series of refactorings: it creates a new superclass, and pulls up the methods of one of the groups into this new class, while updating all the references to these methods and checking if the changes do not have any unwanted effect on the system (the changes should be behaviour preserving).

The scenario shows how powerful the combination of metrics, visualization, FAMIX-based analysis and refactorings can be. Of course, not every big class can be nicely split (and quite often there is a good reason to have a specific big class). Currently we are researching how far we can get in finding possible solutions to potential problems. In the end, however, only the developer can decide if a potential problem is really a problem and if the proposed solution is indeed a good and viable solution.

The fact that most of the analysis is based on the language-independent representation of software in MOOSE, makes the scenario applicable for every language supported by MOOSE and the MOOSE REFACTORING ENGINE.

9 Validation and Evaluation

MOOSE and its tools have been validated in a few industrial experiences. The idea was that members of our team went to work on the industrial applications in a 'let's see what they can tell us about our system' way. There was no training of the developers with our tools. The common point about those experiences was that the subject systems were of considerable size and that there was a narrow time constraint for all experiences we describe below:

1. A very large legacy system written in C++. The size of the system was of 1.2 million lines of code in more than 2300 classes. We had four days to obtain results.

2. An medium-sized system written in both C++ and JAVA. The system consisted of about 120,000 lines of code in about 400 classes. The time frame was again four days.

3. A large system written in SMALLTALK. The system consisted of about 600,000 lines of code in more than 2500 classes. This time we had less than three days to obtain results. Parsing and storing the complete system took less than 5 minutes on a PC Pentium III 500Hz.

The fact that all the industrial case studies where under extreme time pressure lead us to mainly get an understanding of the system and produce overviews [3]. We were also able to point out potential design problems and on the smallest case study we even had the time to propose a possible redesign of the system. Taking the time constraints into account, we obtained very satisfying results. Most of the time, the (often initially sceptical) developers were surprised to learn some unknown aspects of their system. On the other hand, they typically knew already about many problems we found.

We learnt that, in addition to the views provided by our tools, code browsing was needed to get a better understanding of specific parts of the applications. Combining metrics, graphical analysis and code browsing proved to be an successful approach to get the results described above. The obvious conclusion is that tools are necessary but not sufficient.

Memory issues

Up to now we did not have problems regarding the number of entities we loaded into the code repository. The maximum number of entities we loaded was around 250,000 in the third industrial case, which was the limit on the available computers. Surpassing 300,000 entities made the environment swap information to the hard disk and back. The code repository might run into problems with multi-million line projects. For that reason we have designed the code repository to support a possible database mapping easily. In that sense the design of the code repository is more database-oriented (with, for instance, a global entity manager than object-oriented.

In addition, the following considerations have to be taken into account when speaking about memory problems. First, the amount of available memory on the used computer system is, of course, an important factor. Secondly, we have never even tried to optimize our environment neither in access speed nor in memory consumption, because so far we did not really have problems in these areas. Therefore, there is some room for improvement, would it be needed in the future. A third aspect is that tools that make use of the repository need some memory of their own as well. For instance, CODECRAWLER needs to create a lot of additional objects (representing nodes and edges) for the purpose of visualization.

The requirements revisited

In section 2, we listed three properties which a reengineering environment should possess. We will now list those properties and discuss how MOOSE evaluates in that context. In section 2 we stated that such an environment should be:

1. **Extensible.** The extensibility of MOOSE is inherent to the extensibility of its meta-model. Its design allows for extensions for language-specific features and

for tool-specific information. We have already built several tools which use the functionalities offered by MOOSE.

2. **Exploratory.** MOOSE is an object-oriented framework and offers as such a great deal of possible interactions with the represented entities. We implemented several ways to handle and manipulate entities contained in a model, as we have described in the previous sections.

3. **Scalable.** The industrial case studies presented at the beginning of this section have proved that MOOSE can deal with large systems in a satisfactory way: we have been able to parse and load large systems in a short time. Since we keep all entities in memory we have fast access times to the model itself. So far we have not encountered memory problems: the largest system loaded contained more than 250,000 entities and could still be held completely in memory without any notable performance penalties.

10 Conclusion and Future Work

In this paper we have presented the MOOSE reengineering environment. First, we have defined our requirements for such an environment and afterwards we have introduced the architecture of MOOSE, its meta-model and the different tools that are based on it.

The facilities of MOOSE for storing, querying and navigating information and its extensibility make it an ideal foundation for other tools, as shown by tools such as GAUDI and CODECRAWLER. Next to that, the environment has proven its scalability and usability in an industrial setting.

Future work includes further development of our MOOSE-based tools, using them to explore in more detail topics such as design extraction, steering of refactorings based on code duplication detection or other kinds of analysis, and evaluating system evolution. Furthermore, we are working on providing extended support for fine-grained analysis by means of composed queries. Next to that, we plan to introduce *classifications* or *groupings* of entities to support higher level views of systems.

Acknowledgements

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975 (FAMOOS).

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] C. T. Committee. Cdif framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, Jan. 1994. See http://www.cdif.org/.

[3] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[4] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, LNCS 1723, Kaiserslautern, Germany, Oct. 1999. Springer-Verlag.

[5] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. technical report, University of Berne, Aug. 1999.

[6] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Berne, Oct. 1999. See http://www.iam.unibe.ch/~famoos/handbook.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. *Automated Software Engineering*, 3(1-2), June 1996.

[9] R. Kazman. Tool support for architecture analysis and design, 1996. Proceedings of Workshop (ISAW-2) joint Sigsoft.

[10] R. Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, Apr. 1999.

[11] M. Lanza. *Combining Metrics and Graphs for Object Oriented Reverse Engineering*. Diploma thesis, University of Bern, Oct. 1999.

[12] G. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36.

[13] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, Sept. 1999.

[14] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[15] TakeFive Software GmbH. *SNiFF+*, 1996.

[16] S. Tichelaar and S. Demeyer. Sniff+ talks to rational rose – interoperability using a common exchange model. In *SNiFF+ User's Conference*, Jan. 1999. Also appeared in the "Proceedings of the ESEC/FSE'99 Workshop on Object-Oriented Re-engineering (WOOR'99)" – Technical Report of the Technical University of Vienna (TUV-1841-99-13).

Integrated Personal Work Management in the TkSee Software Exploration Tool[1]

Timothy C. Lethbridge

School of Information Technology and Engineering, 150 Louis Pasteur, University of Ottawa, K1N 6N5 Canada. Tel +1 613 562-5800 x6685, Fax +1 613 562-5187, http://www.site.uottawa.ca/~tcl tcl@site.uottawa.ca

Abstract

Typical tools for software maintenance help with searching the code, building models of that code, making changes to it, and keeping track of the changes. However, since maintenance is a complex process, these tools often require the maintainer to juggle numerous windows containing different types of information, and to very carefully manage their work so that nothing is forgotten. In other words, few tools adequately support the process of managing a maintainer's context – his or her plans and mental models for solving a problem. In this paper we show how the TkSee tool facilitates *personal work management* in software maintenance. It does this using two levels of hierarchies which provide one-click access to key information, using a single-window interface. TkSee is being developed by studying maintainers and then providing facilities that address inefficiencies in their work practices.

Keywords: Personal work management, software maintenance, browsing history, searching in source code.

1. Introduction

The goal of our research is to improve the productivity of software engineers (SEs) performing maintenance. To achieve this, we study SEs in the field, discover potential inefficiencies in their work practices, design tools to reduce these inefficiencies and then evaluate the tools to determine their effectiveness. Our field studies have taken place at a telecommunications company where SEs are developing and maintaining a system that is about 18 years old and contains millions of lines of code.

In this paper, we address the issue of *personal work management* which encompasses many of the activities and problems we have observed in our studies. By personal work management, we mean organizing and keeping track of the problem-solving context in addition to software artifacts such as source code and documentation. The context includes: 1) tasks and subtasks one is doing or planning; 2) information one is using to perform these tasks; and 3) one's overall mental model of the problem, of potential solutions to the problem, and of the system as a whole.

For the purposes of this paper, we restrict our attention to situations where SEs are working with the source code itself to solve maintenance problems. Several other researchers are also studying maintainers; for example, Litman et al. [1] look at their mental models; Boehm-Davis et al. [2] discuss the influence of program structure on maintenance, while Bendifallah and Scacchi [3] study general maintenance work practices. Many other researchers are also conducting empirical studies into other aspects of the software engineering process, for example D'Astous [4] studies SEs in group meetings.

This paper should be of interest to software tool developers and researchers as well as the users of those tools, the SEs themselves. We first describe aspects of our research methodology in more detail and highlight some of our findings regarding tasks performed by SEs as well as problems they face. Then we describe TkSee, a tool designed explicitly to address these problems in an integrated way. Finally, we compare TkSee with certain other software engineering tools.

2. Techniques used to gather data

We have been fortunate to have been able to perform field studies with a very enthusiastic team of software developers over a five-year period. During this period we have conducted several different data-gathering exercises [5] [6] and also released several versions of TkSee. Elsewhere [7], we describe our experiences establishing a solid industry-university research relationship.

When developing a software tool like TkSee, there is a temptation in both industry and academia to implement any new 'good idea' as a feature. We have tried to resist this temptation by making sure we first gather solid evidence that each feature is really needed. This is for two reasons: Firstly, we want to follow a good scientific and engineering methodology – basing our work on solid evidence. Secondly, we want to develop tools that will be accepted by the users – trying

[1] This research is sponsored by the Consortium for Software Engineering Research (CSER) and supported by NSERC and Mitel Corporation.

to get users to change their work practices without clear benefit is a recipe for shelfware.

We have used the following techniques to gather our evidence:

- Interviews
- Surveys
- Asking the SEs to draw diagrams of their mental models
- Simple observation
- Synchronized shadowing (systematic note-taking using laptop computers)
- Logging of tool use
- Heuristic evaluation of usability.
- Videotaped evaluation of tool use

The first six techniques are part of the repertoire of empirical studies and can be used before the tool is developed as well as afterwards. The latter two techniques are most commonly used in the human-computer interaction community, but are also excellent for refining knowledge about what aspects of tools work and what aspects do not [8] [9].

TkSee has undergone numerous changes in response to data from these studies. It has been continuously used since 1996 by various SEs, several of whom use it on an everyday basis. It is mostly used by new employees who have joined the group since it was first deployed and has, according to the company [7], substantially reduced the total time required for these new employees to become productive.

3. Some observations from our field studies

During our fieldwork, we have studied SEs working primarily in a Unix™ environment; they use Emacs and other Unix tools, especially grep and certain in-house facilities. We have also observed people using TkSee itself and some competing products (e.g. Source Navigator™ [10]).

Several important results have arisen from our field studies:

Firstly, we observe that the most important single class of activity performed by SEs is *searching*: Software engineers search in several ways for many kinds of search targets. Searches are either global, i.e. searching the entire system using a tool such as 'grep'; or local, i.e. searching within a file. The information sought includes the *definitions* of entities and the *uses* of entities. The entities include variables, types, routines or files and the seeker might be interested in either *specific lines* of code or merely the *modules* containing definitions or uses. The seeker might already know the name of the entity being sought, or may have to make an educated guess as to what its name might be.

Secondly, we note that considerable time is spent *manipulating the results* of searches; such manipulations include:

- Copying the name of something found from one tool or subtool and pasting it into another.
- Keeping lists of search results which can act as 'to-do' lists; either of entities to be changed in some way, or issues to investigate further. These lists are often kept on paper or in simple text files. Not all of the subtask lists come from search results, but many do.
- Developing miniature mental models of the problem at hand, using the search results as evidence. These mental models can be architectural (e.g. describing what connects to what), or represent control flow or data flow. They are normally local in scope – little attempt is generally made to understand the system at a high level.

Thirdly, we note considerable *context switching* on the part of the software engineers. Context switches are of two types, interruptions and drilling-down. In both cases, the SEs temporarily suspend the current task and must resume it later, recalling their mental models and lists of intended subtasks. Interruptions occur when the SE must attend a meeting, go to lunch, answer the phone or stop work for the day. Drilling down occurs when the SE notices a subproblem that must be solved.

In all three tasks described above, we have noted various inefficiencies and difficulties. We describe these in the next section.

4. Difficulties faced when searching and keeping track of work

When studying SEs, we notice the following key classes of problems they face – all of these relate to not having fast enough access to desired information:

4.1 Consuming energy and time by having to switch among too many windows and/or tools

SEs who use Unix tools often have to jump from tool to tool – for example it is common for SEs to run different grep sessions in different shells, and to have several text editor sessions and other tools open. Some commercial program comprehension tools require the user to open many windows in normal use. This jumping from window to window can be time consuming and it requires mental energy to remember where everything is located. Users of Emacs are partly able to conquer this problem by using various 'buffers' available in that tool, however they still have to switch among buffers to access different types of data.

4.2 Losing information previously found

As noted earlier, our observational data show that searching is one of the key activities performed by SEs. However, we see many instances where SEs perform

the same search repeatedly since there are few facilities for storing their results.

Some SEs save search results by writing them on paper (a clear inefficiency), others use Emacs buffers or separate files. However, neither of these solutions is entirely satisfactory due to the overhead of doing the saving, and then finding the data again later. In otherwise one's ability to organize the results is weak.

4.3 Losing track of what they have to do and their mental models

The SEs constantly form and refine plans and models, often based on search results. Since the SEs have to frequently switch contexts, they then have to recall their plans and models upon returning to an earlier context – this proves somewhat error-prone and certainly is time-consuming.

5. An overview of TkSee

In this section we first describe the functions of TkSee; then we describe how TkSee attempts to overcome the problems described above.

We will focus on the user interface of TkSee; other aspects of the design of TkSee are described in [11].

5.1 The panes in the TkSee window

Figure 1 shows a high-level view of TkSee's main window; Figure 2 is a screen dump of an active session. All the activity in TkSee occurs in this one window, plus a few pop-up dialog boxes. The main window contains three panes which can be resized: The information pane, the exploration pane and the history pane.

The *information pane*, on the right of the TkSee window, can contain virtually any type of information – TkSee is designed to allow new subtools to be integrated such that they display their output in this pane. Typical types of information displayed in the pane are files, information about the use of variables and types, as well as data about problem reports from a configuration management system. When a file is displayed, a routine or specific line of code within that file can be highlighted; also, the code can be displayed statically or else be actively debugged. In the latter case, there are pointers to the currently executing statement and to breakpoints.

Many operations are available in the information pane, including the ability to select any text and search for the selected text either locally within the pane, or globally. Global searches place the results in the exploration pane. If the information pane contains an active debugging session, then TkSee makes available standard debugging operations such as setting breakpoints and stepping through the code.

The *exploration pane* contains a graph whose nodes are the types of information that can be displayed in the information pane, and whose arcs are the relationships between these types of information. Selecting a node causes corresponding information to appear in the information pane. Typical arcs include the 'calls', 'defines', 'refers to' and 'includes' relationships. The user initiates an exploration by performing a global search, extracting a subset of nodes from another exploration, or starting to debug a program. Once some nodes are selected, various queries cause new arcs and nodes to appear; the user can also delete nodes that are not of interest. The exploration graph is generally a true hierarchy, so the standard appearance of the nodes is in the form of an indented list (we plan to add a graphical node-and-arc alternative in a future release).

An exploration pane thus serves as the user's personal 'sandbox' for manipulating search results and building a mental model that helps with his or her current subtask.

The *history pane* contains a hierarchy with one node for each exploration. Selecting a history node replaces the contents of the exploration pane, and consequently also replaces the contents of the information pane. A new history node is added whenever an exploration is initiated as described above. The user can rename any exploration if they wish so they can remember its purpose – e.g. it might represent a particular mental model, or a significant list of things to do for a task.

History nodes (explorations) are only deleted at the explicit request of the user. If the user reverts to an earlier exploration (i.e. he or she returns to an earlier subtask), new history nodes will be considered subnodes of that earlier exploration. The history pane thus forms a hierarchy.

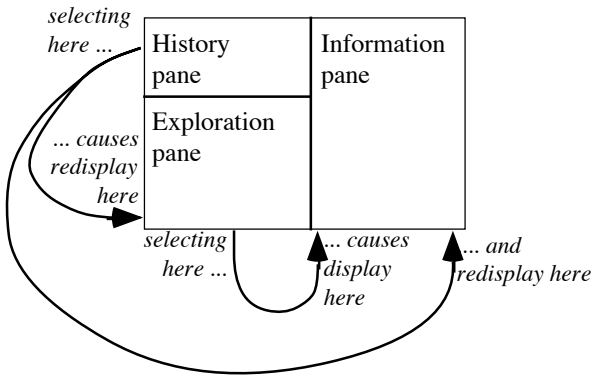


Figure 1: The three main panes in TkSee and the effect of selecting an item in each pane.

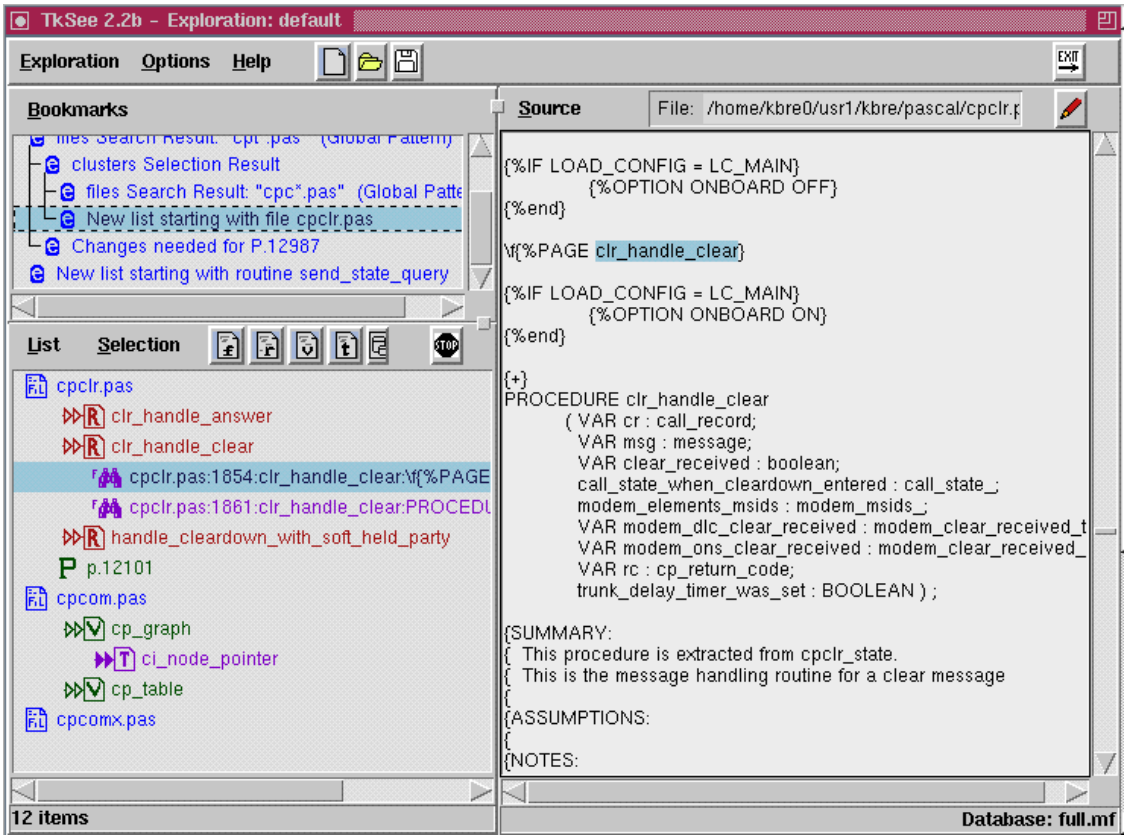


Figure 2: An example screen dump of TkSee. The exploration (left) shows some files (with '.pas' extension), some routines ('R' icon) and variables ('V' icon) in these files at one level of indentation; a problem report ('P' icon) related to the file, and a selected line of code resulting from a grep search in a routine (binoculars icon). The source code for the selected routine is displayed in the information pane at the right. The top-left history pane shows the current task, highlighted by the 'e' icon.

5.2 The twin hierarchies of TkSee

It is a central hypothesis of the design of TkSee that personal work management can be improved by making the exploration and history panes appear as hierarchies. Alternative presentations of these types of information in other tools include separate windows, separate subwindows, or simple lists.

TkSee explorations are hierarchies of *heterogeneous* information. These structures serve multiple functions: 1) They allows related information to be kept together (one can easily see which query led to which subsequent query by looking at levels of indentation). 2) They save search results so they are not lost, while allowing un-needed 'hits' to be deleted. 3) They allows one-click access to details of each exploration node – i.e. by displaying the details in the information pane. 3) They allow the SE to maintain a list of things to do or a structured mental model of some aspect of the system's design.

TkSee history hierarchy elements allow one-click access to any of a set of complete explorations. The SE can easily click on different history items to jump back and forth among tasks without losing context. The fact that the history is a hierarchy parallels the fact that SEs work on a hierarchy of tasks and subtasks.

The following summarizes what we believe to be the theoretical benefits of using hierarchies as the main visual organizing technique:

• *Related information is kept close together:* When using simple indentation to show the relationship between one node and another, the two nodes are as physically close as possible. We posit that this reduces the effort to keep track of information and helps users focus their attention. In particular it helps solve the problems described in sections 4.1 (too much switching among windows) and 4.2 (losing information previously found).

• *One representation for any kind of relationship:* Many software tools show different kinds of relationships in different ways. They might use hierarchies for some information, but use separate windows or tools in other cases. We find that hierarchies provide a unifying representation capable showing practically any information.

• *Hierarchical data is modelled naturally:* Much of the information related to software systems tends to be

hierarchical in nature: Examples from software architecture which can appear in the exploration pane include inheritance hierarchies, routine call hierarchies and file inclusion hierarchies. Task hierarchies, appearing in the history pane, are similarly hierarchical in nature. Note that the TkSee hierarchies are not *exclusively* used to show naturally hierarchical relationships since arcs (indentation) can be used to show any arbitrary relationship.

• *Heterogeneous hierarchies are readily usable and understandable:* Early prototypes of TkSee showed that users understand the heterogeneous hierarchies. Any usability problems they discovered related to other aspects of the interface.

• *Hierarchies are easily manipulable:* The user can easily delete and rearrange nodes in a hierarchy to build a model of just the relationships among information he or she is interested in.

5.3 Other aspects of TkSee that facilitate personal work management

The following are two other features of TkSee that attempt to solve the problems listed in section 4, and thus facilitate personal work management. These features work synergistically together with the twin hierarchies.

The single window: Having a single window in TkSee was a conscious decision designed to combat the problems of losing information and jumping among tools. We recognize the risk of losing information *within* the single window – this is combated by the other features listed below.

Information pane plugins: TkSee can be enhanced by adding different types of information which are displayed as nodes in an exploration, and presented in detail in the information pane. This allows us to integrate various tools, reducing the need for the user to switch among windows.

For example, the ability to display lines of code is implemented by simply calling standard Unix grep – TkSee's value-added in this case is the *management* of the results returned by grep. Two lines returned by grep, that resulted from searching for items in the exploration pane, can be seen in Figure 2.

As other examples of plugins, we have recently integrated standard Unix debuggers into TkSee and are integrating separately-designed facilities for analysing traces and clustering source code to recover architecture. As with grep, these tools merely add different types of nodes to an exploration.

6. Comparison of TkSee features to those of certain other tools

TkSee's facilities have parallels in other tools although no tool has made quite the same design choices as TkSee designers. In this section we compare TkSee to Emacs, to web browsers and to Source navigator, a commercial source code exploration tool.

6.1 Comparison to Emacs

Emacs [12] is a highly functional editor with some facilities for exploration and the ability to be expanded easily to provide new functions. For Unix programmers it is often the environment of choice, although it requires a considerable investment of time before one becomes a expert.

TkSee, on the other hand, is strictly an exploration tool; if a user wants to edit a file displayed in the information pane, he or she must click on an icon that opens the file in the editor of the users' choice (which could be Emacs). Since TkSee is being developed primarily for research into program comprehension, we don't want to spend time giving it editor facilities – we would never be able to give it the power of Emacs.

TkSee is, unlike Emacs, designed to be usable by beginners with little training. In fact, our goal is to make it so intuitive that documentation is not needed. Our user documentation is therefore very minimal.

Like users of TkSee, users of Emacs often perform all their work in a single-window, using Emacs' 'buffers' to store information they want to revisit; TkSee's explorations provide similar facilities but with several advantages:

- Explorations contain heterogeneous information organized hierarchically..
- Explorations themselves are organized using the history pane - this hierarchy of hierarchies gives considerable organizational power.

Unlike Emacs, TkSee currently doesn't allow users to open more than one information pane at once – something that would be useful, for example, if one wanted to compare two files. However, one can rapidly flip back and forth between files by moving the cursor between their names in the exploration pane.

It has been suggested that we could build TkSee functionality into Emacs, and a future project might endeavour to do that. However we are moving in the direction of supporting various kinds of graphical views in TkSee, something the text-oriented Emacs cannot readily handle.

Among TkSee users, most use Emacs as well, suggesting that the functionality of the two tools is complementary.

6.2 Comparison to Web Browsers

Web browsers are widely used to explore vast information spaces, so why not just render software information as html, and use a web browser's navigation facilities in place of TkSee?

The information pane of TkSee is very similar to a web page in the sense that it can contain many different

types of information, can be searched internally and has hyperlinks.

The main power of TkSee for personal work management, however, comes from its exploration and history panes. These are quite different from what is available in a web browser, unless you encapsulated TkSee functionality in a Java applet. Table 1 provides a detailed comparison of the different types of work management capabilities in the two environments – TkSee takes some features from web browser bookmarks and history and combines them with querying capabilities and one-click access to information.

Several other researchers [13] [14] [15] have suggested how web browser history mechanisms could be improved.

	TkSee History	**TkSee Exploration**	**Web Browser History**	**Web Browser Bookmarks**
Nature of an individual node	An **'exploration'** (next column), a heterogeneous graph of information created and edited by the user	Could represent a subsystem, file, routine, variable, line of code … or **anything**	A **'web page'** (can contain anything)	A **'web page'**, or a manually specified category of web pages
Reason for presence of a node	A **task or subtask** of the user	Information **resulting from a query**; may or may not be interesting or have been visited	Information **visited**	Information visited that is **interesting**
Structure of a *set* of nodes	A **hierarchy**	Typically a **hierarchy**, but can be a **graph**	A **list**	By default a **list**, but can be a **hierarchy**
Nature of child nodes	Subtasks	Information in any relationship to parent (e.g. 'calls', 'includes')	n/a	Non-leaves are categories; leaves are web pages
Upon revisiting a node	Redisplay the exploration; the user can edit the exploration	Redisplay details of the unit of information	Redisplay web page	Redisplay web page
A new node is created…	**Automatically** when a query replaces the current exploration graph; or **manually** if the user chooses to extract a subgraph of an exploration	**Manually** in response to a query (many nodes may be created at once)	**Automatically** on visiting a page	**Manually**, whenever the user is interested
Ability to delete a node	Yes	Yes	No	Yes
Operations on *sets* of nodes	Deletion only	Queries to create children of multiple parents, deletion, extraction to create new history	None	Deletion only
Ability to rename a node	Yes (to attach more meaningful name)	No	No	Yes (needed when web page title is poor)
Persistence	Permanent (users may have several files or use the default)	Permanent (stored in an exploration)	Typically current session only	Permanent (users may have several files or use the default)

Table 1: 'History' and 'Bookmark' facilities used for personal work management in TkSee and typical web browsers.

6.3 Comparison to Source Navigator

Source Navigator is a commercial source code browsing tool. In fact, we often use it to maintain TkSee. The reason we do not use TkSee to maintain itself is because we do not yet have a parser for Tcl/Tk, the language in which TkSee is written.

Source Navigator shares many facilities with TkSee: It allows one to browse many different types of software objects and relationships. It has the advantage of allowing one to edit code directly and, since it is a commercial tool, has a wide range of very robust parsers.

Source navigator has panes that resemble TkSee's exploration pane in the sense that they contain certain types of hierarchies of information. However it does not present all types of information in the same hierarchy, nor does it have one-click access to a hierarchical history of explorations. This forces the maintainer to open several different windows to access information.

6.4 Comparison to integrated development environments (IDEs)

Many compilers today come with integrated development environments. There is a long history of research into such facilities dating back to the early days of Lisp and Smalltalk [16] [17 [18] [19].

In addition to providing a unified tool for the editing, configuration management and compilation process, many of these provide program exploration facilities similar to Source Navigator such as displaying call hierarchies and variable-use cross reference information.

Most IDEs, however, use different windows to display different types of information. Furthermore, we have found no IDE that effectively targets the difficulties we identified in which users have trouble keeping track of their search results, their mental models of some small aspect of the system and their hierarchy of tasks.

7. Evaluation

Evaluation is an essential aspect of research into software tools. We are using several techniques to evaluate the use of TkSee in a real industrial environment. In particular, we want to determine whether the distinctive features of TkSee – those oriented around personal work management – in fact provide a significant advantage over features in alternative tools. At the current time we are still in the early phases of this evaluation.

Our first step has been to evaluate TkSee's usability and remove any superficial problems that might mask more important advantages or disadvantages. We have so far completed two cycles of usability evaluation; we describe this process elsewhere [8] [9].

Our second evaluation technique is to monitor ongoing use of the tool – we do this by logging each invocation of each command. So far, TkSee has been used on a discretionary basis by over 20 different people, some of whom have used it for several years on an every-day basis. Management reports that the time taken by new employees to learn about the subject software has dropped considerably [7]. We consider this to be good evidence that TkSee, as a whole, is probably more useful than other tools that the developers have available. Therefore we have achieved our goal of making software engineers more productive, at least in the context of the particular group with which we are working.

Our next step is to analyse the detailed use of specific TkSee features to discover which ones contribute to the success, and how much. We plan to do this using questionnaires, more detailed analysis of logs and observation.

Our overall three-step evaluation approach is described in more detail elsewhere [20].

8. Conclusions

TkSee's twin hierarchies – its history pane and its exploration pane – provide robust, simple-to-learn personal work management capabilities for the software maintainer.

By personal work management, we mean organizing three aspects of the context surrounding ones work: Firstly, keeping track of tasks and subtasks the maintainer must perform, despite frequent context-switching. Secondly, organizing information one is using to solve one's problem, particularly information one has retrieved through various search or query mechanisms. Thirdly, allowing manipulation of one's personal mental models of the system and the problem.

We define an *exploration* to be heterogeneous graph, normally a hierarchy, whose nodes represent any kind of information related to software including files, variables and problem reports. A new exploration is first created as a result of performing a search. The maintainer then incrementally modifies the exploration by selecting nodes and deleting them or performing further queries. Over time the exploration comes to represent part of the maintainer's personal mental model of the system or a list of things to do. TkSee constantly displays an exploration in its bottom-left pane; a simple click on a node displays more details, such as the source code, in the right-hand information pane.

A set of explorations is maintained in the history pane. This set can typically represents a hierarchy of tasks, and provides one-click task switching – selecting a history item replaces the contents of both the exploration and information panes.

Many other tools provide some of the features of TkSee, but we believe that TkSee is the first software

engineering tool to provide the twin-hierarchy approach to improve software engineers' personal work management.

TkSee has been enthusiastically used by a variety of users on a large maintenance project for several years. We are continuing to develop TkSee and to experiment with it to learn more about how software engineers can be made more productive.

Acknowledgements

We would like to thank the Mitel employees and management who made this work possible, Janice Singer at the National Research Council of Canada who collaborates in the studies of software engineers, and the many students and research associates who have developed TkSee over the years.

References

[1] Litman, D., Pinto, J., Letovsky, S. & Soloway, E. (1996). "Mental models and software maintenance," *Proc. Empirical Studies of Programmers: First Workshop.*, 1996.

[2] Boehm-Davis, D. Holt, R., & Schultz, A. (1992). "The role of program structure in software maintenance", *Int. J. of Man Machine Studies*, 36, 21-63.

[3] Bendifallah, S., & Scacchi, W. (1987). "Understanding software maintenance work", *IEEE Trans. Software Engineering*, 13(3), 311-323.

[4] D'Astous, P., and Robillard, P. (2000). "Protocol analysis in software engineering studies". *Empirical Studies in Software Engineering*, El-Emam, K. and Singer, J. Eds., MIT Press, in press.

[5] Singer, J., and Lethbridge T.C. (1998), "Studying Work Practices to Assist Tool Design in Software Engineering", *proc. 6th IEEE International Workshop on Program Comprehension*, Italy, pp. 173-179.

[6] Lethbridge, T.C. and Singer, J. (2000). "Experiences Conducting Studies of the Work Practices of Software Engineers", Erdogmus, H. and Tanir, O. Eds, *Advances in Software Engineering*. Springer-Verlag, in press.

[7] Lethbridge, T.C., Lyon, S., and Perry, P.. (2000). "The Management of University-Industry Collaborations Involving Empirical Studies of Software Engineering", El-Emam, K. and Singer. J Eds, *Empirical Studies in Software Engineering*. MIT Press, in press.

[8] Lethbridge, T.C. and Herrera, F. (2000). "Towards Assessing the Usefulness of the TkSee Software Exploration Tool: A Case Study", Erdogmus, H. and Tanir, O. Eds, *Advances in Software Engineering*. Springer-Verlag, in press.

[9] Herrera, F. (1999), "A Usability Study of the TkSee Software Exploration Tool" (1999), M.Sc. Thesis, Computer Science, University of Ottawa

[10] Cygnus Corporation, maker of Source Navigator (now part of Red Hat). http://www.cygnus.com/sn/

[11] Lethbridge, T. and Anquetil, N., (1997), "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study", *University of Ottawa, Computer Science Technical report* TR-97-07. http://www.site.uottawa.ca/~tcl/papers/Cascon/TR-97-07.html

[12] Halme H. and Heinanen J. (1988), "GNU Emacs as a dynamically extensible programming environment", Software-Practice & Experience, 18 (10), Oct. pp. 999-1009.

[13] Tauscher, L. and Greenberg, S., (1997), "How People Revisit Web pages: Empirical findings and Implications for the Design of History Systems", *International J. of Human-Computer Studies* 47, 1 July 1997. p. 97-137.

[14] Kandogan, Eser. Shneiderman, Ben. (1997) "Elastic windows: A hierarchical multi-window World-Wide Web browser", *UIST (User Interface Software and Technology): Proceedings of the ACM Symposium*. ACM, New York, NY, USA.. pp. 169-177

[15] Hirsch FJ. "Building a graphical Web history using Tcl/Tk." (1997), *Proceedings of the Fifth Annual Tcl/Tk Workshop*. USENIX Assoc. pp.159-60.

[16] Konsynski, B.R., Kottemann, J.E., Nunamaker, J.F. Jr. and Stott J.W. (1984), "PLEXSYS-84: an integrated development environment for information systems", J. Management Information Systems, 1 (3), Winter 1984-1985, pp. 64-104

[17] Ueda Y. (1989), "The Lisp programming environment", Journal of the Information Processing Society of Japan", 30 (4), pp.314-325.

[18] Schefstrom, D. (1989), "Building a highly integrated development environment using preexisting parts", Information Processing 89. Proceedings of the IFIP 11th World Computer Congress. North-Holland. 1989, pp. 345-350.

[19] Maguire, L.P, McGinnity, T.M. and McDaid, L.J. (1999), "Issues in the development of an integrated environment for embedded system design: User needs and commercial products", Microprocessors & Microsystems, 23 (4), Oct., pp. 191-197.

[20] Lethbridge, T.C. (2000), "Evaluating a Domain-Specialist Oriented Knowledge Management System", International Journal of Human-Computer Studies., to appear.

Displaying and Editing Source Code in Software Engineering Environments

Michael L. Van De Vanter[1] and Marat Boshernitsan[2]

[1]*Sun Microsystems Laboratories*
901 San Antonio Avenue
Palo Alto, CA 94303 USA
Tel +1 650 336-1392, Fax +1 650 969-7269, Email michael.vandevanter@sun.com

[2]*Department of Computer Science*
University of California at Berkeley
Berkeley, CA 94720-1776 USA
Tel +1 510 642-4611, Fax +1 510 642-3962, Email maratb@cs.berkeley.edu

Abstract

Source code plays a major role in most software engineering environments. The interface of choice between source code and human users is a tool that displays source code textually and possibly permits its modification. Specializing this tool for the source code's language promises enhanced services for programmers as well as better integration with other tools. However, these two goals, user services and tool integration, present conflicting design constraints that have previously prevented specialization. A new architecture, based on a lexical representation of source code, represents a compromise that satisfies constraints on both sides. A prototype implementation demonstrates that the technology can be implemented using current graphical toolkits, can be made highly configurable using current language analysis tools, and that it can be encapsulated in a manner consistent with reuse in many software engineering contexts.

Keywords: Program editor, software engineering tool integration, language-based editing

1. Introduction

Any interactive software engineering tool that deals with programs inevitably displays source code for a human to read and possibly modify[1]. The technology for doing this, however, has changed little in twenty years, despite a compelling intuition that specializing the technology for programming languages might increase user productivity substantially. In contrast, consider how word processing systems have evolved beyond simple text editors during those same twenty years.

Extensive research, numerous prototypes, and more than a few commercial attempts have failed to deliver practical language-based editing for source code. Programmers find such systems difficult and unpleasant when compared with simple text editors. Tool builders find that implementations are fragile and place high demands on supporting infrastructure.

Language-based editing will only succeed in practice when it addresses the real goal: to help programmers program in the context of existing skills and tools. This translates to two sets of requirements, often conflicting, for an editor:

- *Programmer's perspective*: the editor must make reading and writing source code easier and more rewarding.
- *Tool builder's perspective*: the editor must reliably share information with other tools, for which it may act as a user interface, and it must be packaged for reuse (portable, highly configurable, and embeddable).

The CodeProcessor[2] is an experimental tool for editing source code, under development at Sun Microsystems Laboratories. It is based on technology that strikes a balance among apparently competing requirements. It is text oriented, but fundamentally driven by language technology. It can make its language-oriented representation (configured by declarative specifications) available to other tools, and can be embedded in other GUI contexts. The key architectural choice is a lexically-oriented intermediate representation for source code that addresses both usability and integration with other tools.

1. We do not address purely graphical programming languages, although some of the issues are similar.

2. "CodeProcessor" is an internal code name for this prototype; it is intended to suggest a specialization of simple text editing for source code, much as word- and document-processors are specialized for natural language documents.

Experience suggests that simple usability testing, better GUI design, or new algorithms would not have produced this design. Rather, it resulted from rethinking the tasks, skills, and expectations of programmers, and from then finding ways to address them: using existing language technology and within the context of practical software engineering tools. The result is an architecture that is different, though not necessarily more complex, than those tried in the past.

This paper presents an overview of the CodeProcessor and the design choices it embodies. Section 2 reviews requirements, and Section 3 discusses how previous technologies have failed to meet them all. Section 4 offers a new look at the design trade-offs needed when combining text editing and language support, and shows how this analysis leads to a solution. Sections 5 and 6 describe the two complementary and mutually dependent aspects of the CodeProcessor's design: architecture and user-model. Finally Section 7 reviews implementation status, followed by related work and conclusions.

2. Design goals

The requirements mentioned in the introduction, and discussed in more detail here, reflect different perspectives: programmers and tool designers. Past failures result from neglecting one point of view or the other; Sections 3 and 4 will show how they can be reconciled.

2.1. No training

All available evidence shows that programmers read programs textually; they also have "structural" understanding, but it is highly variable and not based on language analysis [10][12]. Programmers have deeply ingrained work habits as well as motor-learning that involves textual editing; they will only accept a tool that is familiar enough for immediate and comfortable use without special training.

This need not, however, prohibit advanced functionality. Consider how users experienced with simple text editors find the transition to word processors smoothed by familiar text entry and cursor commands. By analogy, language-based editing services should be layered carefully onto basic text editing behavior, imposing no (or barely noticeable) restrictions.

2.2. Enhance reading and writing

Additional editing services derive from specialization for the tasks confronting programmers. A familiar example is automatic indentation of source code lines. This service is based loosely on linguistic structure, and it helps both reading (visual feedback on nesting) and writing (saving tedious keystrokes). This particular service can be delivered in a simple text editor, but it can and should be taken much further.

Research shows that high quality, linguistically-driven typography measurably improves reading comprehension [3][19]. In many environments, reading is still the dominant task for programmers, even while writing code [9][31]. Good designs for program typography are available (for example the paper-based publication designs by Baecker and Marcus [3]), yet rarely used.

Also highly important, is special support (both reading and writing) for program comments. Transparent to conventional language tools, comments are tedious to format but crucial for readers.

Although specialized enhancements are important, it is absolutely essential that they not make things worse. Any intrusion on text editing must respect the "balance of power" between user and tool. This can be delicate even in the simplest of cases, for example auto-indentation mechanisms that programmers find helpful but "not quite right."

Nowhere has intrusiveness been more problematic than in treatment of fragmentary and malformed source code. This is, of course, the normal state for programs under development. Unfortunately, language-based editors typically treat such situations as user "errors" and encourage or require corrective action. The real "error" is that the tools fail to model what the user is really doing [14] and cannot function usefully until rescued. Editing tools must function without interruption in any context.

2.3. Access to linguistic structure

Software engineering tools (for example analyzers, builders, compilers, and debuggers) generally operate over structural source code representations such as abstract syntax trees. An editing tool is most easily integrated with other tools if it can share such representations, but as Section 3.1 discusses, this presents severe design challenges for a tool whose job is to display and permit modification to source code in terms of text.

2.4. Configuration and embedding

Finally, as software engineering tools evolve, emphasis shifts from standalone editing systems to specialized tools that must work with other tools. A tool for source code editing must be well encapsulated, somewhat like a GUI component, and not demand complex support such as a particular kind of source code repository. Reflecting the reality that practical software engineering involves many languages, it should be easily configured via language specifications. In order to be used as an interface by many other tools, an editing tool must have a visual style that is easily configured for different contexts and tasks.

3. The design space

At the heart of a specialized editing tool is an internal representation for source code. Conventional choices, depicted in Figure 1, are divided by a gulf between fundamentally different approaches: one oriented toward usability and one toward higher level services.

Figure 1: Design choices for program editors

3.1. Pure designs

At the far right of the diagram are "structure editors" [4][6][8][18], so called because of internal representations closely related to the tree and graph structures used by compilers and other tools. This greatly simplifies some kinds of language-oriented services, but it requires that programmers edit via structural rather than textual commands. Behind this approach is a conjecture, articulated by Teitelbaum and Reps, that programs are intrinsically tree structured, and that programmers understand and should manipulate them that way [25]. Unfortunately, years of failed attempts [11], combined with research on program editing [17] and on how programmers really think about programs [13][22] have refuted that conjecture. From a tool integration perspective, the advantages of complete linguistic analysis are offset by its fragility (in the presence of user editing) and context-dependency (the meaning of code in many languages depends potentially on all the other code with which it will run). Few structure editors are in use today

At the far left are simple text editors with no linguistic support. Editing is simple and familiar, but there is no real specialization for source code. Integrating a simple text editor with software engineering tools requires complex mappings between structure and text, but these typically result in restrictive and confusing functionality, fragile representations (for example, where the identity of structural elements is not preserved during editing operations), or both [27].

3.2. Modified designs

Subsequent efforts in language-based editing can be viewed as attempts to bridge this gulf. Some structure editors allow programmers to "escape" the structure by transforming selected tree regions into plain text [21], but usability problems persist. The complex unseen relationship between textual display and internal representation makes editing operations, both structural and text escapes, confusing and apparently unpredictable [27] because of "hidden state." Textual escapes make matters with a confusing and distracting distinction between those parts of the program where language-based services are provided and those where they are not. Often language services and tools stop working until all textual regions are syntactically correct and promoted back into structure.

At the left side of Figure 1 are widely used code-oriented text editors such as Emacs [23]. These use a purely textual representation, assisted by ad-hoc regular expression matching that recognizes certain language constructs. The structural information computed by simple text editors is, by definition, incomplete and imprecise. It therefore cannot support services that require true linguistic analysis, advanced program typography for example. Simple text editors typically provide indentation, syntax highlighting[1] and navigational services that can tolerate structural inaccuracy. A malformed program will, at worst, be incorrectly highlighted.

A few text editors perform per-line lexical analysis with each keystroke, but the information has never been fully exploited and the lack of a true program representation leads to confusion in the inevitable presence of mismatched string quotes and comment delimiters.

3.3. Inclusive designs

A more inclusive approach is to maintain both textual and structural representations. Although this approach promises a number of advantages [5][26], it is difficult to keep the representations consistent and it has not been demonstrated that the cost and complexity are justified.

4. Finding the middle ground

Section 3 described a fundamental design tension:

- It is desirable to maintain a linguistically accurate program representation, updating it on every modification, however small.
- The greater the degree of structural sophistication, the more fragile the representation is in the presence of unrestricted textual editing, and the more room there is for confusing behavior and inconsistency between what's seen and what's represented internally.

In summary, an ideal representation would be closely related to displayed text, but would also reflect linguistic structure at all times. What's needed is a compromise

1. "Syntax highlighting" is an unfortunate misnomer, since pattern-matching is considerably weaker than syntactic analysis. It would be more accurate to call it "unreliable keyword, string, and comment recognition".

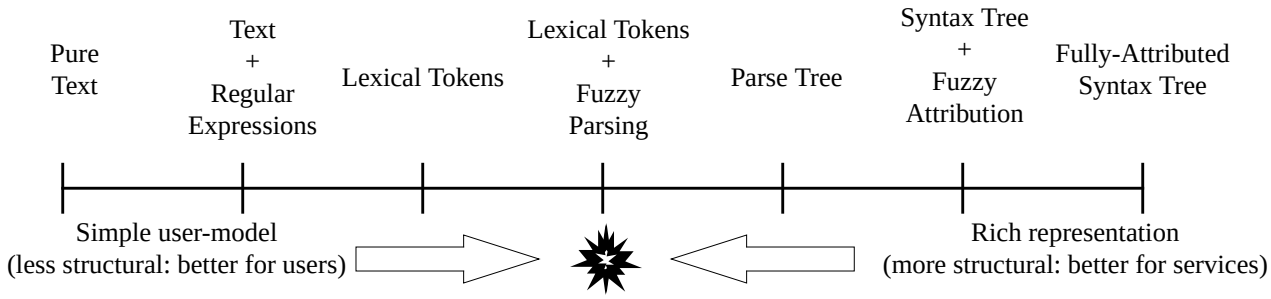


Figure 2: Additional choices for program representation and analysis

somewhere in the middle of Figure 1, where the amount of language analysis performed is as simple (and localized) as possible, but also as useful as possible.

A compromise can be found by taking a closer look at language analysis: both the internal engineering of compilers, and the formal language theory behind it. A typical compiler analyzes textual programs in phases, shown below. Each stage is driven by a different kind of grammar

lexical analysis → parsing → static semantic analysis

(corresponding approximately to types 3, 2, and 1 in the Chomsky grammar hierarchy) and uses a corresponding kind of analyzer [29]. Programming languages are often designed around this grammatical decomposition, and batch-oriented compilers benefit from the simplicity and formal foundations of separate phases.

This decomposition reveals additional choices, depicted in Figure 2, for analyzing and representing programs being edited. Possible representations include the standard products of each phase: lexical token stream, parse tree, and attributed tree respectively. Intermediate choices include partial analysis of the next grammatical level: regular expression matching is a partial lexical analysis, fuzzy parsing is a partial syntactic analysis which recognizes only certain features of the context-free syntax (e.g. nested parenthesis or context-dependent categorization of identifiers into function and variable names), and partial semantic attribution that can be used for computing limited amounts of semantic context. Partial analyses are often simpler to implement (fuzzy parsing can be performed through a simple pattern matching on the token stream) and more forgiving of inconsistencies in the representation.

An important distinction among the three analysis phases concerns the scope of cause and effect. Static semantic analysis (closely related to Chomsky's context-sensitive syntax) at each point in a program depends potentially upon the entire program. Parsing (context-free syntax) depends only on the enclosing phrase, but assumes that program is well formed. Lexical analysis (regular syntax) depends only on adjacent tokens, making it highly suitable for the inner loop of an editor.

Thus the lexical representation, not used in any prior systems, emerges as a promising compromise:

- It is a stream, not a tree, and thus bears a close relationship to textual source code;
- The analysis needed to update the representation after each edit usually requires only local context;
- It is suitable for program fragments;
- It has enough linguistic information to provide many language-based services, including more robust implementation of familiar services such as indentation, parenthesis and bracket matching, procedure or method head recognition, etc.; and
- It is a language representation suitable for integration with other tools, including complete language analyzers. Further analysis, for example parsing, could be folded into the CodeProcessor if added carefully, but at some additional cost in complexity.

Although this approach is promising, a number of design questions remain:

- Can the textual display and behavior be made to look and feel familiar enough that it requires no training?
- To what degree can the display be specialized for programs using only lexical information?
- Can such a fine-grained typographical display be implemented using current toolkit technology and made configurable?
- Can the lexical token representation be made robust in the presence of partially typed and badly formed tokens? In particular, how can "bracketed" tokens such as string literals be managed when one of the brackets (double quotes for strings) is missing?
- What specialized support for comments and other, possibly non-textual, annotations is possible?
- How can a description-driven lexical analyzer be adapted to update the representation after each keystroke?

Solutions appear in the following two sections, which summarize respectively the two mutually dependent aspect of the CodeProcessor's design: architecture/implementation and user-model. The architecture is presented first in Section 5, although many aspects were driven by the user-model design described in Section 6.

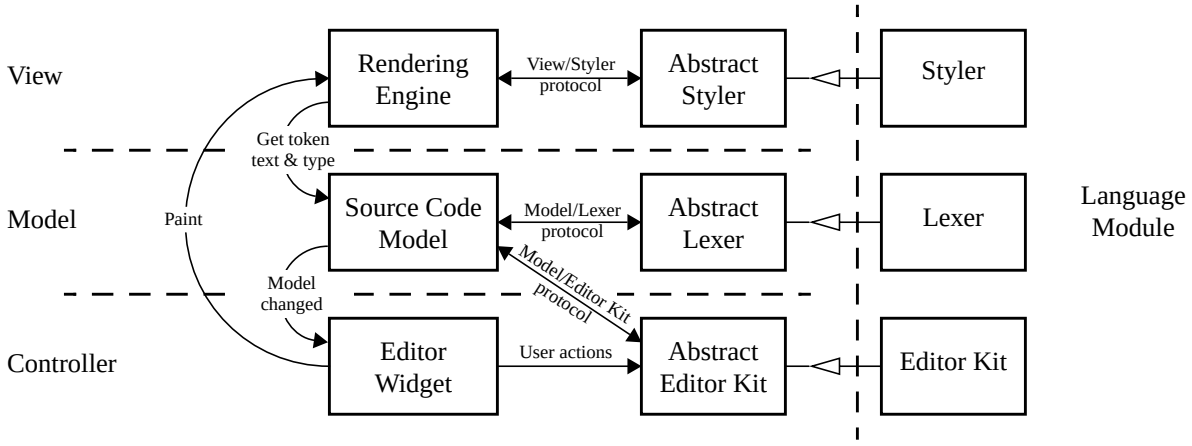


Figure 3: CodeProcessor Architecture

5. Architecture

The CodeProcessor's architecture, depicted in Figure 3, is based on the Model-View-Controller design paradigm. This choice is not accidental: in addition to being a natural architecture for display and editing, it also reflects the design of the Java™ Foundation Classes (JFC) "Swing" toolkit and its text framework [30] which was used to implement the current prototype. Multi-lingual behavior is supported by separating each of the three core modules into two components: one implementing the language-independent functionality, and the other (collectively known as a *Language Module*) providing language-sensitive features for a particular language. In the CodeProcessor this separation is achieved by subclassing, but other decompositions are also possible.

The remainder of this section describes each of the major design constituents in order.

5.1. The Controller

The *Controller* is manifested through two closely related components: the *Editor Widge*t and the *Editor Kit*. The Editor Widget is responsible for dispatching window system events and making the CodeProcessor a fully functional member of the JFC widget family. The Editor Kit implements the intricate editing behavior described in Section 6.2.

Much of Editor Kit's functionality is language-independent; some, however, may be custom-tuned for each particular language, for example adding keyboard shortcuts for inserting language constructs.

The primary responsibility of the Editor Kit is to implement user actions that require taking the context of the action into the consideration. Some actions, such as cursor movement commands, require no changes to the source code model; their execution depends only on the context (tokens) surrounding the cursor. Other actions, such as insertions and deletions, may depend not only on the modification context, but also on the state *after* the modification, since certain nuances of the user-model require "looking into the future."

To facilitate this, the Editor Kit commences a two-stage modification process upon any potential change. First, the source code model is requested to consider the effects of the change *without* modifying the underlying content. This produces an object describing the change in terms of a model transformation that needs to take place. When the Editor Kit regains control it examines the transformation, either discarding it, if it has no effect or is not valid, or applying it to the model.

5.2. The Model

As discussed in Section 4, source code is represented as a sequence of lexical tokens, although this representation is extended in several crucial ways. This representation allows for much-needed flexibility, as it both supports the required user-model, and fits naturally with the incremental lexical analysis algorithm.

The lexical analysis algorithm, developed by Tim Wagner [28], is fully general: it supports unbounded contextual dependency and multiple lexical states. Moreover, incrementality can be crafted onto existing batch lexers that conform to a simple interface. For instance, the current prototype's lexer for the Java programming language is generated by the JavaCC tool [16] from a readily available lexical specification; the specification is extended to include various categories of irregular lexemes created during editing, as discussed in Section 6.1.

Figure 4 depicts the modification of a model after insertion of the characters "`=x`" into a fragment containing the four tokens '`a`', '`+`', '`c`', and '`;`' with cursor initially between '`+`' and '`c`'. Figure 4a represents the

content immediately prior to the modification, 4b -- the transformation resulting from considering given modification, and 4c -- the content after the suggested transformation has been applied.

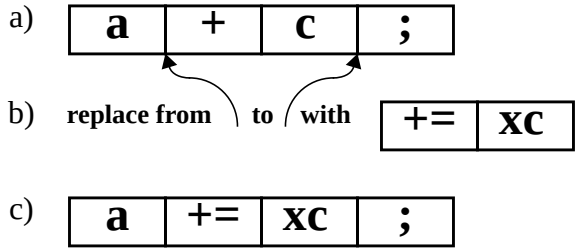


Figure 4: Example model update

The source code model is also responsible for adding and removing "separators," special non-linguistic tokens whose role in the user-model is described in Section 6.2. Other non-linguistic tokens include comments, line breaks, and other layout directives.

A significant advantage of the model, from the perspective tool integration is that it enables *stable* references to source code structure: during any kind of editing, the *identity* of unaffected tokens is guaranteed.

5.3. The View

The rendering mechanism displays source code in accordance with the requirements outlined in Section 6.1. The typographically-enhanced display is facilitated by assigning stylistic properties to each token by means of the *Styler* component. The Styler lends itself to being automatically-generated, although the current implementation uses hand-written Stylers.

Stylers can also be used to export human-readable source code from the CodeProcessor by rendering into a character stream, dropping stylistic information that cannot be represented. Appropriate formatting can be achieved by Stylers optimized for text output.

5.4. Representing embedded structures

Programming languages commonly include embedded syntactic structures that have distinct lexical rules, most notably comments and strings. Embedded structures are supported by nested editors with transparent boundaries (behavioral considerations are presented in Section 6.3). The only requirements for this support, easily met by all embedded language structures we have encountered, are that they have well-defined linguistic boundaries and that their contents be tokenized as a single entity by the language lexer.[1]

This architecture permits utilization of any editors in the JFC text framework, including the CodeProcessor recursively. The mapping from token types to editor types is performed by the Language Module; this module in the current prototype uses the standard JFC text editor for comments and a token-based CodeProcessor editor for strings and character literals.[2]

6. Functionality and user-model

This section presents an overview of the CodeProcessor's functional behavior as well as the user-model experienced by the programmer.

6.1. Advanced program typography

The CodeProcessor is visually distinguished by its advanced typographical "styles," implemented by the view architecture described in Section 5.3. These styles approximate designs by Baecker and Marcus [3] and are updated with each keystroke as the source code is being incrementally reanalyzed. Alternate styles for each language can be selected dynamically, either to suit individual preference or as required by particular tools driving the display. The style appearing in Figure 5 is configured

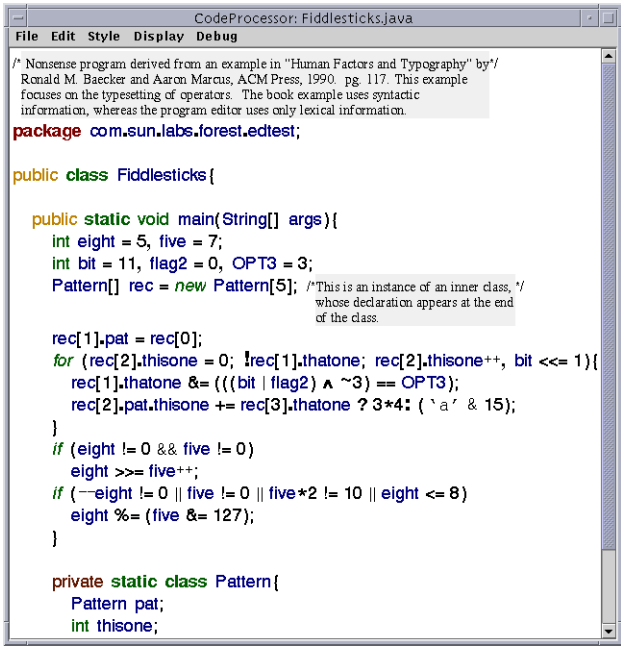


Figure 5: Example CodeProcessor display

by 123 token categories to which are assigned 61 separate token styles.[3] Each token style specifies type face, size relative to a base, style (plain, bold, italic), foreground and

1. If the nested editor is, in fact, another instantiation of the CodeProcessor, the contents of an embedded structure may be further tokenized by the nested lexer.

2. Both strings and character constants afford a simple lexical description that recognizes character escapes such as \n, \t, etc. This lets us, for example, highlight legal escapes so that they are distinguishable from the rest of the text, as well as indicate which ones are invalid.

3. Much of the stylistic detail is required as compensation for the absence of type faces suitable for programs [3].

background colors, baseline elevation, and both left and right boundary specifications used to compute display spacing between adjacent tokens. Token styles can also specify alternate display glyphs, for example to display ligatures.

In a departure from the Baecker and Marcus designs, which require well-formed programs, CodeProcessor styles reveal that certain tokens are lexically incomplete (for example "`0x`") or badly formed (for example "`08`"), based on lexical grammars extended to include such tokens. The CodeProcessor treats such tokens as legitimate in every other respect.

Although the Baecker and Marcus designs require full program analysis, a surprising amount of the visual detail can be achieved using only lexical information. Indentation requires fuzzy parsing in the style of many text editors. More visual features could be added through other kinds of fuzzy parsing, for example adjusting operator spacing based on expression depth.

Horizontal spacing between tokens is computed from the source code, not affected by presses on the space bar. This improves legibility and saves keystrokes, much in the same way that conventional auto-indentation works at the beginning of each line. We anticipate adding a tab-like mechanism to the current prototype that gives programmers some ability to impose vertical alignment.

6.2. Editing behavior

The CodeProcessor behaves like a code-oriented text editor in most respects. Where it differs, the behavior has been designed so that it appears to do the right thing when used as a text editor. Preliminary experience with the CodeProcessor's user-model suggests that programmers find descriptions of the behavior confusing, but the behavior itself unremarkable.

Some behaviors are completely conventional. Indentation is automatic. Line breaks are explicitly entered and deleted by the programmer.[1] Typing text within comments and language tokens (especially string literals) is likewise conventional, with the notable exception that programmers can easily type multi-line comments (and perhaps eventually strings), as shown in Figure 5.

Non-standard behavior appears in and around token boundaries. To first approximation, token boundaries are determined purely by the lexical analyzer. When the cursor rests between two tokens it is displayed midway between them; pressing the space bar silently does nothing.

However, not all boundaries can be unambiguously computed, for example between keywords. Here the CodeProcessor automatically inserts a "separator" token. This behaves somewhat like a "smart space" in a word processor: no more than one can be present between adjacent lexical tokens. The cursor can rest on either side of a separator; deleting a separator is treated as a request to join surrounding lexical tokens (if they could not be joined, there would have been no separator present). Separators often come and go as the lexical categories of adjacent tokens are changed by editing, but since they are behavioral rather than visual, this is not distracting.

String literals and comments receive special treatment, as described in the following section. Additional subtleties in the user-model, beyond the scope of this paper, are required so that "the right thing" appears to happen at all times.

6.3. Nested editors

The user-model for editing programs described in the previous section is inappropriate in certain regions. The contents of string literals obey different grammars than surrounding code, and the contents of comments are not analyzed at all.

Such regions receive special support in the CodeProcessor, beginning with behavior that preserves their boundaries during all normal editing. This has the flavor of structure editing, but it solves a number of traditional problems with boundary confusion; potentially confusing behavior can smoothed over with careful design.

Having guaranteed boundary stability for these regions, the CodeProcessor can then provide specialized behavior in a straightforward way. Specialized editors are simply embedded to match the model: one kind for strings, another for character literals, yet another for plain text comments. More can be added, for example to support HTML or graphical comments. Although this has something of the flavor of a compound document system, it is specialized for source code and designed so that the boundaries are no more obtrusive than absolutely necessary. For example, the text cursor moves smoothly across boundaries between code and embedded structures.

6.4. The Programmer's Experience

The net result of these behaviors is by design an editing experience that is visually rich but otherwise unobtrusive. Nearly all familiar keystroke sequences have their intended effect, with the added bonus of fine-grained visual feedback. Time wasting efforts at whitespace management, for example deciding where to insert spaces and how to align multi-line comments, become as unnecessary as manual indentation. This frees the programmer to concentrate more completely on the task at hand: understanding and writing code. Furthermore, the rich display engine creates new opportunities for tools to present information by modulating the source code display to suit the task at hand.

1. The CodeProcessor does not break lines, but it would be helpful to add a linguistically driven mechanism for "wrapping" lines wider than the available window.

7. Implementation status

Initial design of the CodeProcessor was carried out at Sun Labs by the first author in the Spring of 1993. A prototype using C++, the *lex* analyzer, and the Interviews graphical toolkit [15], was demonstrated later that year as part of a larger programming environment project. An evolution of the first prototype, using the Fresco toolkit [7] (itself an evolution of Interviews) was completed and demonstrated in early 1995, at which time work ceased with the conclusion of the project. The design was then shelved, awaiting more suitable infrastructure than was available at that time.

The second author commenced a reimplementation of the CodeProcessor design during a summer internship at Sun Labs in 1998, adding recent improvements in incremental lexing technology and adapting the recently developed text framework from the JFC swing toolkit [30]. This prototype, which will be subject to further refinement and evaluation, is substantially complete, with the exception of automatic indentation and other services not part of the core design.

8. Related work

Emacs [23] is an augmented text editor of the kind described in Section 3.2. Its editing *modes* add specialized behavior and text coloring via pattern matching, but they fall short of the CodeProcessor's requirements. Weak encapsulation of its internal representation, as well as insufficient model-controller separation, makes reliable representation and manipulation of structural information difficult, if not impossible. Language analysis is limited to (unreliable) regular expression matching of fewer than ten lexical constructs. Rendering and layout, even in the more recent XEmacs [32], does not meet the CodeProcessor's demands. The editors embedded in many commercial integrated development environments have basic text editing and display functionality comparable to Emacs.

Numerous structure editors, mentioned in Section 3.1, were built in research environments, for example Centaur [6], Gandalf [18], Mentor [8], and PSG [4]. All had acknowledged usability problems [11].

The commercialized Synthesizer Generator [21] is a notable example of the modified structure editors described in Section 3.2, but was still plagued by confusing behavior [27] and by restrictions on editing.

The Pan system [5] is characteristic of the inclusive designs described in Section 3.3. It permitted unrestricted text editing, performed full incremental language analysis on demand, and provided semantic feedback. Although some attention was paid to usability [26], the implementation was enormously complex and offered no language-related advantages during textual editing. Important features such as comments received no special support at all.

Several elements of the CodeProcessor's design subsequently appeared in the Desert environment, including attention to usability, adoption of advanced typesetting, and the choice of a token-based representation [20]. FRED, the Desert editor, performs language analysis via integration with the FrameMaker document processing system [1]. This limits FRED's ability to support fine-grained language-based behavior due to the lack of appropriate abstractions in the Frame Developer's Kit API [2]. Moreover, reliance on a sizable document processing system reduces the likelihood of embedding FRED elsewhere.

9. Conclusions

We have designed and prototyped source code editing technology that addresses the full spectrum of requirements faced by designers of software engineering tools. This technology matches programmers' skills and expectations, and brings to bear the power of language-based technology in support of both the people and other tools in the environment. Meeting these often conflicting requirements required both a new user-model for its behavior as well as a new architecture. Its construction stretches the limits of the existing infrastructure.

History tells us that less ambitious designs will fail. Some language-oriented technology can be grafted onto simple text editors, but insufficiently rich representations limits their power and accuracy. Some usability compromises can be made to language-oriented structure editors, but the fundamental architecture dooms their usability.

A lexical-based architecture by itself would also fail, since a naive user-model would suffer many of the ills of tree-oriented editors. Likewise, the new user-model by itself would fail, since the mismatch between it and existing representations would preclude adequate implementations.

The CodeProcessor performs enough linguistic analysis to permit useful tool integration, as well as useful language-based services such as high-quality on-the-fly typography. At the same time its fundamental behavior is textual, permitting easy adoption by programmers, and it includes specialized support that simplify and extend comment management significantly.

Designing tools that are both powerful and effective is difficult, and the more "low level" the tool, the more demanding are the user requirements. Starting with these requirements, however, and embracing the notion that powerful tools must above all fit with programmers skills, expectations, and tasks, gives hope that benefits of software development technology can actually make a difference in the way people work.

10. Acknowledgments

The reimplementation of this design was made possible by support from Mick Jordan, Principal Investigator of the Forest Project at Sun Microsystems Laboratories. Yuval Peduel made helpful comments on early drafts of

this paper, and we thank the anonymous reviewers for their constructive suggestions as well.

11. Trademarks

References

[1] Adobe Systems Incorporated, Adobe FrameMaker, http://www.adobe.com/products/framemaker/

[2] Adobe Systems Incorporated, Frame Developer's Kit, http://partners.adobe.com/asn/developer/framefdk/fdkguide.html

[3] Ronald M. Baecker and Aaron Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley Publishing Co. (ACM Press), Reading, MA, 1990.

[4] Rolf Bahlke and Gregor Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Transactions on Programming Languages and Systems* **8**,4 (October 1986), 547-576.

[5] Robert A. Ballance, Susan L. Graham and Michael L. Van De Vanter, "The Pan Language-Based Editing System," *ACM Transactions on Software Engineering and Methodology* **1**,1 (January 1992), 95-127.f

[6] P. Borras, D. Clemént, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual, "CENTAUR: the system," *Proceedings ACM SIGSOFT '88: Third Symposium on Software Development Environments*, November 1988, 14-24.

[7] Steve Churchill, "C++ Fresco: Fresco tutorial," *C++ Report*, (October 1994).

[8] Véronique Donzeau-Gouge, Gérard Huet, Giles Kahn and Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe and Erik Sandewall (editors), McGraw-Hill, New York, NY, 1984, 128-140.

[9] Adele Goldberg, "Programmer as Reader," *IEEE Software* **4**,5 (September 1987), 62-70.

[10] Robert W. Holt, Deborah A. Boehm-Davis and Alan C. Schultz, "Mental Representations of Programs for Student and Professional Programmers," in *Empirical Studies of Programmers: Second Workshop*, Gary M. Olson, Sylvia Sheppard and Elliot Soloway (editors), Ablex Publishing, Norwood, New Jersey, 1987, 33-46.

[11] Bernard Lang, "On the Usefulness of Syntax Directed Editors," in Advanced Programming Environments, Lecture Notes in Computer Science vol. 244, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), Springer Verlag, Berlin, 1986, 47-51

[12] Stanley Letovsky, "Cognitive Processes in Program Comprehension," in *Empirical Studies of Programmers*, Elliot Soloway and Sitharama Iyengar (editors), Ablex Publishing, Norwood, New Jersey, 1986, 58-79.

[13] Stanley Letovsky and Elliot Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software* **3**,3 (May 1986), 41-49.

[14] Clayton Lewis and Donald A. Norman, "Designing for Error," in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper (editors), Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 411-432.

[15] Mark A. Linton, John M. Vlissides, and Paul R. Calder, "Composing user interfaces with InterViews," *Computer*, **22**,2 (February 1989), 8-22.

[16] Metamata, Inc. "JavaCC - The Java Parser Generator: A Product of Sun Microsystems," http://www.metamata.com/JavaCC/

[17] Lisa Rubin Neal, "Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors," P*roceedings SIGCHI Conference on Human Factors in Computing Systems*, Toronto, Canada, April 1987, 99-102.

[18] David Notkin, "The GANDALF Project," *Journal of Systems and Software* **5**,2 (May 1985), 91-105.

[19] Paul Oman and Curtis R. Cook, "Typographic Style is More than Cosmetic," *Communications of the ACM* **33**,5 (May 1990), 506-520.

[20] Steven P. Reiss, "The Desert Environment," *ACM Transactions on Software Engineering and Methodology* **8**, 1 (October 1999), 297-342.

[21] Thomas Reps and Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Springer Verlag, Berlin, 1989. Third edition.

[22] Elliot Soloway and Kate Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering* **SE-10**,5 (September 1984), 595-609.

[23] Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* **16**,6 (June 8-10 1981), 147-156.

[24] Gerd Szwillus and Lisa Neal (editors), *Structure-Based Editors and Environments*, Academic Press, 1996.

[25] Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM* **24**,9 (September 1981), 563-573.

[26] Michael L. Van De Vanter, Susan L. Graham and Robert A. Ballance, "Coherent User Interfaces for Language-Based Editing Systems," *International Journal of Man-Machine Studies* **37**,4 (1992), 431-466, reprinted in [24].

[27] Michael L. Van De Vanter, "Practical Language-Based Editing for Software Engineers," in *Software Engineering*

and Human-Computer Interaction: ICSE '94 Workshop on SE-HCI: Joint Research Issues, Sorrento, Italy, May 1994, Proceedings, Lecture Notes in Computer Science vol. 896, Richard N. Taylor and Joelle Coutaz (editors), Springer Verlag, Berlin, 1995, 251-267.

[28] Tim A. Wagner, *Practical Algorithms for Incremental Software Development Environments*, UCB/CSD-97-946, Ph.D. Dissertation, Computer Science Division, EECS, University of California, Berkeley, December 1997.

[29] William M. Waite and Gerhard Goos, *Compiler Construction*, Springer-Verlag, 1984.

[30] Kathy Walrath and Mary Campione, *The JFC Swing Tutorial: A Guide to Constructing GUIs*, Addison-Wesley, 1999.

[31] Terry Winograd, “Beyond Programming Languages,” *Communications of the ACM* **22**,7 (July 1979), 391-401

[32] XEmacs, http://www.xemacs.org

SESSION 2

INTEGRATION, INTEROPERABILITY, AND DATA INTERCHANGE

Construction of an Integrated and Extensible Software Architecture Modelling Environment

John Grundy[1]

[1] *Department of Computer Science, University of Auckland,*
Private Bag 92019, Auckland, New Zealand,
Phone: +64-9-3737-599 ext 8761, Fax: +64-9-3737-453, Email: john-g@cs.auckland.ac.nz

Abstract

Constructing complex software engineering tools and integrating them with other tools to form an effective development environment is a very challenging task. Difficulties are exacerbated when the tool under construction needs to be extensible, flexible and enhanceable by end users. We describe the construction of SoftArch, a novel software architecture modelling and analysis tool, which needs to support an extensible set of architecture abstractions and processes, a flexible modelling notation and editing tools, a user-controllable and extensible set of analysis agents and integration with OOA/D CASE tools and programming environments. We developed solutions to these problems using an extensible meta-model, user-tailorable notation editors, event-driven analysis agents, and component-based integration with process support, OOA/D, code generation and reverse engineering tools.

Keywords: software engineering tools, software architecture, modelling notations, analysis agents, tool integration

1. Introduction

Building complex software development tools and integrating these tools with existing 3rd party tools is very challenging [20, 7, 17]. We have been developing a novel software architecture modelling and analysis tool, SoftArch, which presents a number of challenges in its construction. SoftArch needs to support an extensible set of architecture modelling abstractions, visual notations and editing tools. It also needs a user-controllable and extensible collection of model analysis agents to assist with validating an architectural model. Import of OOA specifications and export of OOD models and code fragments is needed, to make use of the tool organisationally feasible.

These requirements are a challenge to meet with conventional tool construction approaches, such as those provided by MetaEDIT+ [12], MOOT [16], KOGGE [3], JComposer [7], and MetaMOOSE [4]. This is because such approaches either produce inflexible, difficult to integrate, configure and extend tools, or provide inappropriate abstractions for building tools like SoftArch.

We describe the implementation of SoftArch using the JComposer meta-CASE toolset and focus on various adaptations we had to make to JComposer's tool development approaches in order to successfully realise SoftArch. We developed an extensible meta-model with its own visual programming language, enabling developers to extend SoftArch's architecture modelling abstractions. Editing tools and notational symbols with a high degree of user-customisability give developers a degree of freedom when representing model abstractions. User controllable and extensible analysis agents were developed using event-driven components, along with a visual end user programming language. The Serendipity-II process management environment [5] provides this event-based end user programming language, plus process and work co-ordination agent support. JComposer itself provides OOA and D model import/export for SoftArch, along with code generation and reverse-engineering support. These tools are integrated using a component-based software architecture. In addition, we have prototyped OOA and D model interchange between SoftArch and Argo/UML [17] using UML models encoded in an XML-based data interchange format. A proposed approach to dynamic architecture visualisation using SoftArch is briefly discussed. We briefly compare and contrast the implementation of SoftArch with other approaches.

2. Overview of SoftArch

There has been a growing need for support for software architecture modelling and analysis tools as systems grow more complex and require more complex architectures [1, 10, 13, 19]. We developed the SoftArch environment to address this need [10]. SoftArch supports the modelling and analysis of large, complex system architectures using primarily multiple views of visual representations of architectural abstractions. SoftArch uses a concept of successive refinements of architecture abstractions, from high-level component characterisations to detailed architectural implementation strategies.

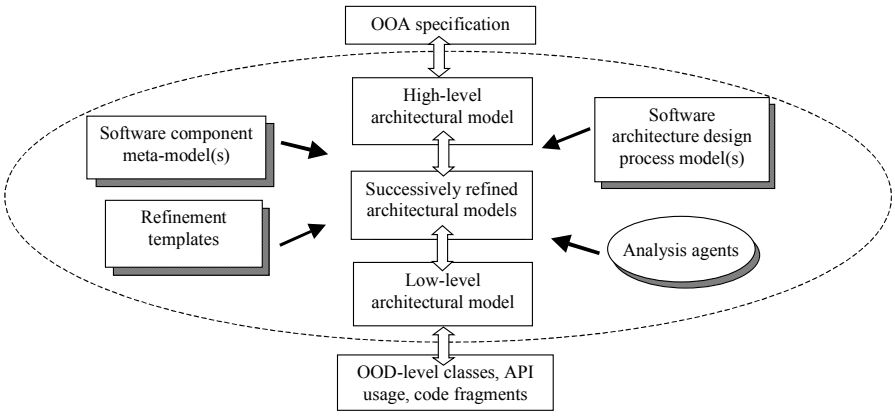


Figure 1. Overview of the SoftArch modelling and analysis approach.

Figure 1 illustrates this concept. An OOA specification (codified functional and non-functional requirements) is imported into SoftArch, typically from a CASE tool. Architects then build an initial high-level architecture for the system that will satisfy these specifications. This high-level model captures the essence of the organisation of the system's software components. It includes information about the non-functional properties of parts of the system, and links architectural components to parts of the OOA specification they are derived from. Architects then refine this high-level model to add more detail, making various architectural design decisions and trade-offs, and ensure the refined architectural models meet constraints imposed by the high-level model. Eventually architects develop OOD-level classes which will be used to realise the architecture, and export these to CASE tools and/or programming environments for further refinement and implementation.

Figure 2 shows an example of SoftArch being used to model the architecture of an e-commerce application (a collaborative travel itinerary planner [8]). The travel planner system is made up of a set of client applications/applets (shown in view (1) at the top). These communicate via the internet to a set of servers, in this example comprising a chat server, itinerary data manager and RDBMS. View (2) shows a more detailed view of the itinerary management part of this system. This includes the itinerary editor client and its connection to the itinerary management server, a client map visualisation, and a map visualisation agent, which updates the map to show a travel path when the itinerary editor client is updated by the user. Architecture components can be refined by creating a subview containing their refinements, by enclosing their refinements (like for "server apps" in view (1)), or using explicit refinement links. OOA and D-level classes and services can also be modelling in SoftArch, and refined to/from appropriate architecture abstractions.

View (3) shows an analysis agent reporting dialogue. A collection of user-controllable analysis agents monitor the state of the architecture model under development. They report inconsistencies, problems or suggested improvements to the user non-obtrusively via this dialogue, are run on-demand by the developer, or act as "constraints" that validate modelling operations as they are performed. SoftArch OOA level abstractions can be sourced from a CASE tool, and OOD-level classes exported to a CASE tool or programming environment (by generating class stubs). Reverse engineering of existing applications is also supported, with OOD-level abstractions able to be imported from a CASE tool and grouped by reverse-refinement into higher-level architectural abstractions.

SoftArch poses various challenges for the tool developer:

- Architectural abstractions include components, associations and component annotations, each which may have a variety of properties [10]. The modelling abstractions available needs to be extensible by the user of SoftArch, to allow them to capture information about the architectural entities they deal with in useful ways, and to add additional components, component properties, etc. as required.
- The modelling notation and editing tools need to be flexible and preferably extensible, supporting model abstraction enhancement and tailorability of the tool. Users should be able to reconfigure the tool to display architecture abstractions as they prefer.
- Templates, or reusable architectural model fragments, are required to assist developers in reusing common architectural styles and patterns. Thus SoftArch must support abstraction of views to templates, instantiation of templates, and ideally support keeping templates and derived model components consistent when either changes.
- Analysis tools that constrain how a model is built and/or check model validity on demand must be user-controllable and extensible. When doing exploratory modelling, modelling alternatives or changing a model dramatically, we have found architects prefer to relax constraints. They then successively re-activate checks as they need them.

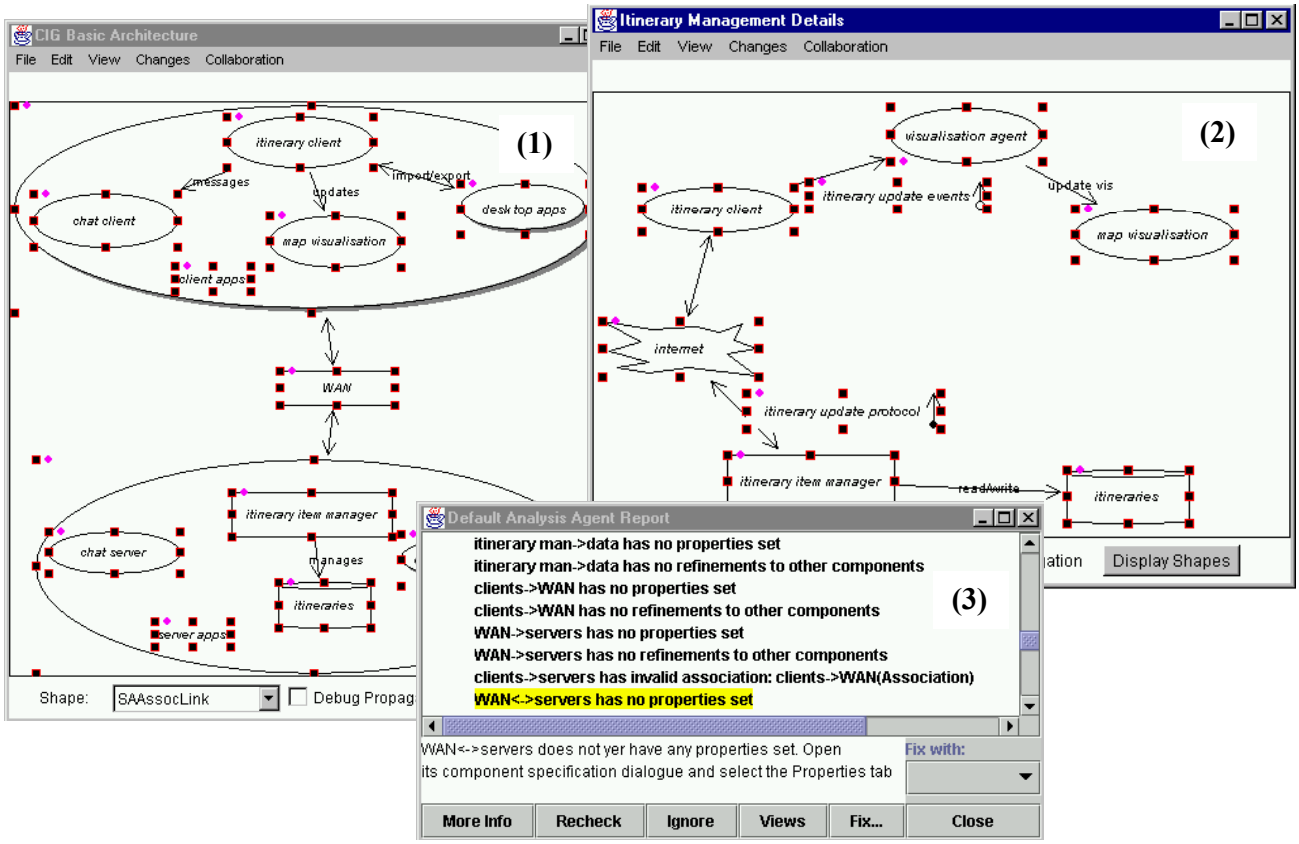


Figure 2. Examples of architecture modelling and analysis in SoftArch.

- The architecture development process should be definable and process management tool support provided to developers. This should not just guide development but also support automated analysis tool activation/deactivation, and configure available modelling abstractions appropriate to the development process stage being worked on.
- Import/export support between CASE tools and SoftArch should leverage existing support within CASE tools where possible. For example, using a CASE tool or programming environment API, using XML-based encoding, or using source code files.

3. SoftArch Architecture and Implementation

The basic architecture of SoftArch is illustrated in Figure 3. SoftArch maintains a collection of meta-model entities, specifying available architectural abstractions and basic syntactic and semantic constraints. A collection of reusable refinement templates supports reuse of common architectural styles and patterns. A collection of analysis agents monitor the changing architectural model and inform the developer of problems. An architecture model holds the current system architectural model (repository, multiple views, refinement links etc.).

We integrated SoftArch with the Serendipity-II process management environment. Serendipity-II provides architecture development process models, work co-ordination agents based on these processes, and user defined analysis agents used to check the validity of SoftArch models. SoftArch was also integrated with the JComposer component engineerint tool. JComposer provides OOA-level class components for SoftArch and SoftArch generates OOD-level class components in JComposer. SoftArch also uses JComposer's code generation facilities to generate Java classes based on OOD-level architectural abstractions and middleware and database component properties described in SoftArch. Generated Java classes can be modified in tools like JBuilder and JDK, and changes reverse-engineered back in JComposer and then into SoftArch. We have prototyped simple XML-based import/export tools, which exchange OOA and D models with Argo/UML.

We implemented SoftArch with the JViews multi-view, multi-user software tool framework, using the JComposer meta-CASE and component engineering toolset [7]. Our JComposer tool also provides a component engineering environment for JViews.

We encountered a number of challenges when using JViews and JComposer to engineer SoftArch. JComposer does not directly support extensible meta-models for CASE tools, however, and its notation tailoring tool enables users to inappropriately modify notation-implementing editors and icons.

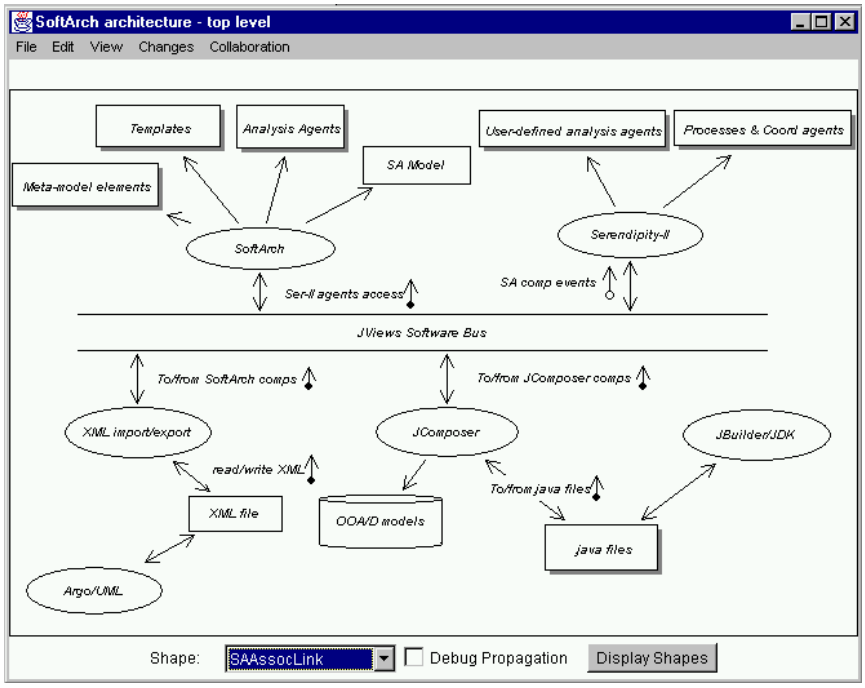


Figure 3. Basic architecture of SoftArch.

Flexible and extensible analysis tools can be built, but no direct abstractions are provided by JViews, and some Java programming is required to achieve these. Tool integration is supported directly via component interface-based mechanisms, but indirectly via components implementing 3rd party tool communication protocols and data exchange format parsing and generation.

The following sections examine the construction of various SoftArch facilities in further detail, focusing on the approaches we used to satisfy some of these more challenging requirements of the tool. As JViews and JComposer did not directly support many of these capabilities, we discuss how we overcame these short-comings. We then discuss the various lessons we learned from developing SoftArch, and summarise some directions for future software tool construction approaches we have been exploring because of this work.

4. Architecture Modelling

5.1. Meta-model Support

SoftArch uses a basic model of architecture components, inter-component associations and component and association annotation to describe architectural models [10]. Each of these architectural entities has a set of properties associated with it. Property values can be simple numbers or strings, or a collection of value ranges. JComposer, like most meta-CASE tools, assumes a tool developer would have a fixed set of tool repository component and relationship types e.g. process stages, in/out ports, filters and actions in Serendipity-II, and components, association, generalisation, aspects etc. in JComposer itself [5, 7]. Thus with SoftArch there might be a fixed set of different architecture component, association and annotation types, each with a fixed set of properties, which could each be modelling as appropriate JViews repository component specialisations.

However, in order to support user-extension of SoftArch's software architecture modelling capabilities, we had to develop a meta-model for SoftArch in JComposer, as well as the component/association/annotation architecture model repository representation. Figure 4 (a) illustrates the basic components of this meta-model. SoftArch components, associations and annotations must each have a type, with the meta-model allowing the specification of valid component associations and annotations. Each different type has a set of properties, which have property type and value constraints. For example, component types include "SA Entity", "OOA class", "Process", "Server", "Client Process", "RDBMS" etc. Association types include "dependency", "data usage", "event subscribe/notify", "message passing" etc. Annotations include "cached data", "data exchanged", "events exchanged", "replicated data", "process synchronisation", etc.

Component (and association and annotation) types also specify valid refinements allowed. For example, the most general "SA Entity" component can be refined to any other kind of architecture component when modelling. The "Client Process" type cannot, however, be refined to "Server Process" or "RDBMS" components, as such a refinement does not make any sense.

Unlike most CASE tools, SoftArch does not inherently enforce constraints like valid associations/annotations, valid refinements or valid properties/property values for components. A set of

analysis agents does this, and selected agents can be turned on and off to allow architects greater or lesser flexibility to model and change architectures (see Section 6). We found this facility to be very useful when architects dramatically change an architecture, or are doing alternative or exploratory modelling of parts of an architecture. Relaxing some constraints makes it easier for architects to morph or revise parts of the model through partially inconsistent states, than if meta-model typing constraints are always rigidly enforced.

We allow users of SoftArch to open JViews projects, which contain partial meta-model specifications. Meta-model components in different projects build upon one another to construct a complete set of component and other types available when modelling architectures with SoftArch. Users can extend the meta-model using a simple visual specification tool, illustrated in Figure 4 (b). Using multiple meta-model projects allows architects to package domain-specific meta-models e.g. "basic abstractions", "real-time systems", "e-commerce systems" etc., each with specialised architecture modelling abstractions.

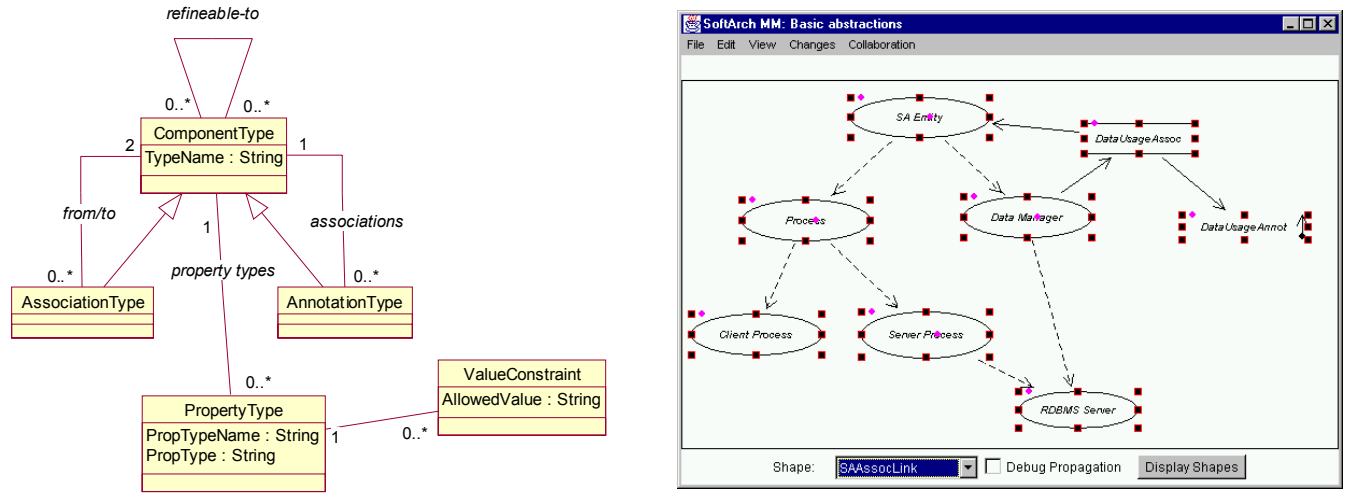


Figure 4. (a) SoftArch meta-model; (b) visually viewing and programming the meta-model.

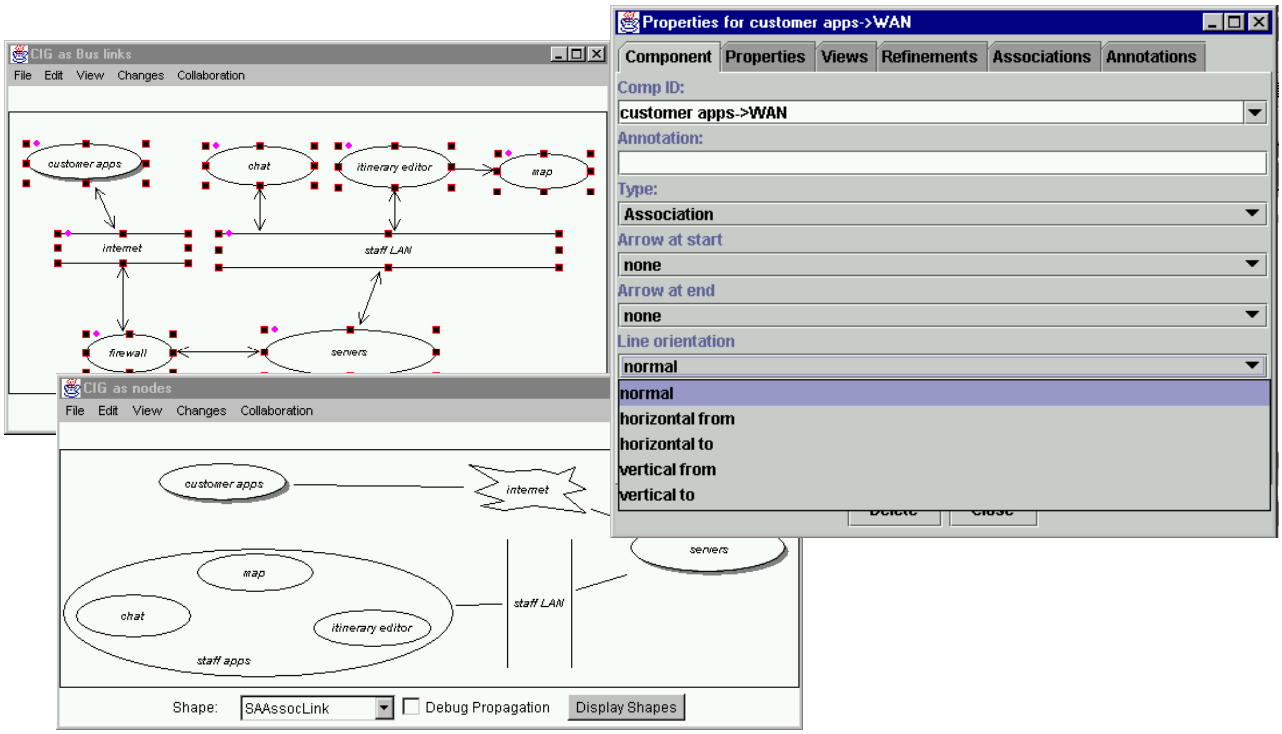


Figure 5. Examples of SoftArch Notation Usage.

5.2. Flexible Modelling Notation

JComposer provides a notation icon editor, BuildByWire, which can be used by tool users to reconfigure their icon appearance in certain ways [7]. With SoftArch, we decided to take an alternative approach and provide users with a range of icon appearances that they could tailor as they required via the same dialogue used to specify and view architecture component properties. For example, Figure 5 shows two examples of modelling the same information in SoftArch, the top view using bus-style associations between client and server components and the bottom node-style connectors and enclosure of clients running on the same host. The dialogue shown provides configuration capabilities allowing users to tailor the appearance of component, association and annotation icons as they require. Automated tailoring can be achieved using Serendipity-II task automation agents (see Section 6).

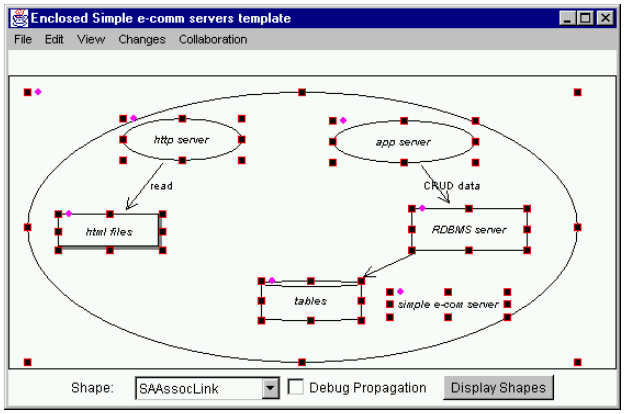
We adopted the customisable icon appearance approach over having end users use BuildByWire directly as it is much easier and quicker for them to tailor icons, and they do not need to learn to use the meta-CASE tool. They also can not make errors and cause SoftArch to fail, which is possible using BuildByWire directly. Users can, however, use the BuildByWire meta-CASE tool to extend the possible icon appearances if no pre-defined ones suit their needs.

5.3. Refinement Templates

In order to support reuse of common architectural styles and patterns, we developed reusable refinement templates for SoftArch. A view in SoftArch which specifies the refinement of one (or more) architectural components into more detailed architectural model components can be copied and packaged for reuse. For example, Figure 6 (a) shows a packaged refinement template commonly used in simple e-commerce applications. The high-level component "simple e-com server" encloses (and thus is refined to) several parts: an http server with html and other files, an application server, and an RDBMS server with tables. Figure 6 (b) shows how the user of SoftArch has reused this refinement template when developing part of the travel itinerary system's architecture. SoftArch allows users to reuse refinement templates by creating subviews for a specified component or by automatically copying the template components into their model (as in this example).

JViews does not explicitly support the concept of templates. When developing Serendipity-II's process templates we built a complex mechanism for copying and instantiating template process models [5].

When developing SoftArch refinement templates we instead extended the versioning and import/export mechanisms JViews supports. A template is created by exporting a view to a file then importing it and using JViews' component identifier (ID) mapping mechanism to create a template. When instantiating a template, we export the template to a file then import it, using the same ID mapping mechanism to create new components with unique IDs in the software architecture model. Refinement links are created automatically by SoftArch for subviews, and are created automatically for imported enclosed components and explicit refinement links.



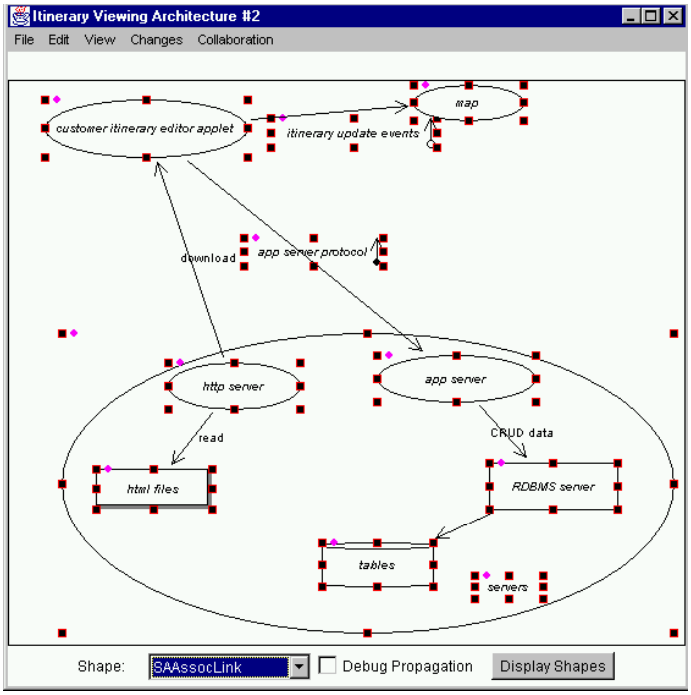


Figure 6. (a) Example of SoftArch template; (b) reused template.

This approach proved to be a much simpler solution than that used for Serendipity-II, but provides almost identical template support. JViews' version merging abstractions [7] can even be used to reconcile changes made to the template or components copied from the template into the architecture model.

5. Process and Analysis Support

6.1. Process Management

We wanted to provide SoftArch users with integrated process management support to allow them to use enacted process models to both guide and track their work. It would also automate tedious tasks like enabling/disabling analysis agents and configuring allowable component types and notation appearance during different stages of architecture model development. Rather than building process support into SoftArch, as done in Argo/UML [17], using CAME tools like MetaEdit+ with very limited automation support [12], or forcing developers to configure the tool themselves, as in Rational Rose [15], we reused the Serendipity-II process management environment.

Figure 7 shows a simple architecture development process in Serendipity-II, along with a task automation agent which enables and disables groups of analysis agents when a particular process stage is enacted or finished. Serendipity-II detects changes made to SoftArch models and records these against process stages, allowing developers to track work associated with different process tasks/subtasks. The task automation agent illustrated here detects process activation/deactivation (the left-hand square icons, or "filters), then uses two actions (shaded ovals) to enable and disable named SoftArch analysis agents (right-hand side rectangles). The actions send events to the SoftArch analysis agent manager to enable or disable the named SoftArch analysis agents. The filters and actions used here are reused from a library of such event-driven components. Others can be implemented using JComposer and Java and added to this library as required.

This integration is achieved by Serendipity-II using JViews' component event propagation mechanism to listen to SoftArch component events and to record these. The task automation agents, like the one shown here, send events to SoftArch which configure analysis agents, configure display of notational symbols and configure available meta-model abstractions. This produces what seems to the developer to be a more or less seamlessly integrated process management and task automation tool support for SoftArch.

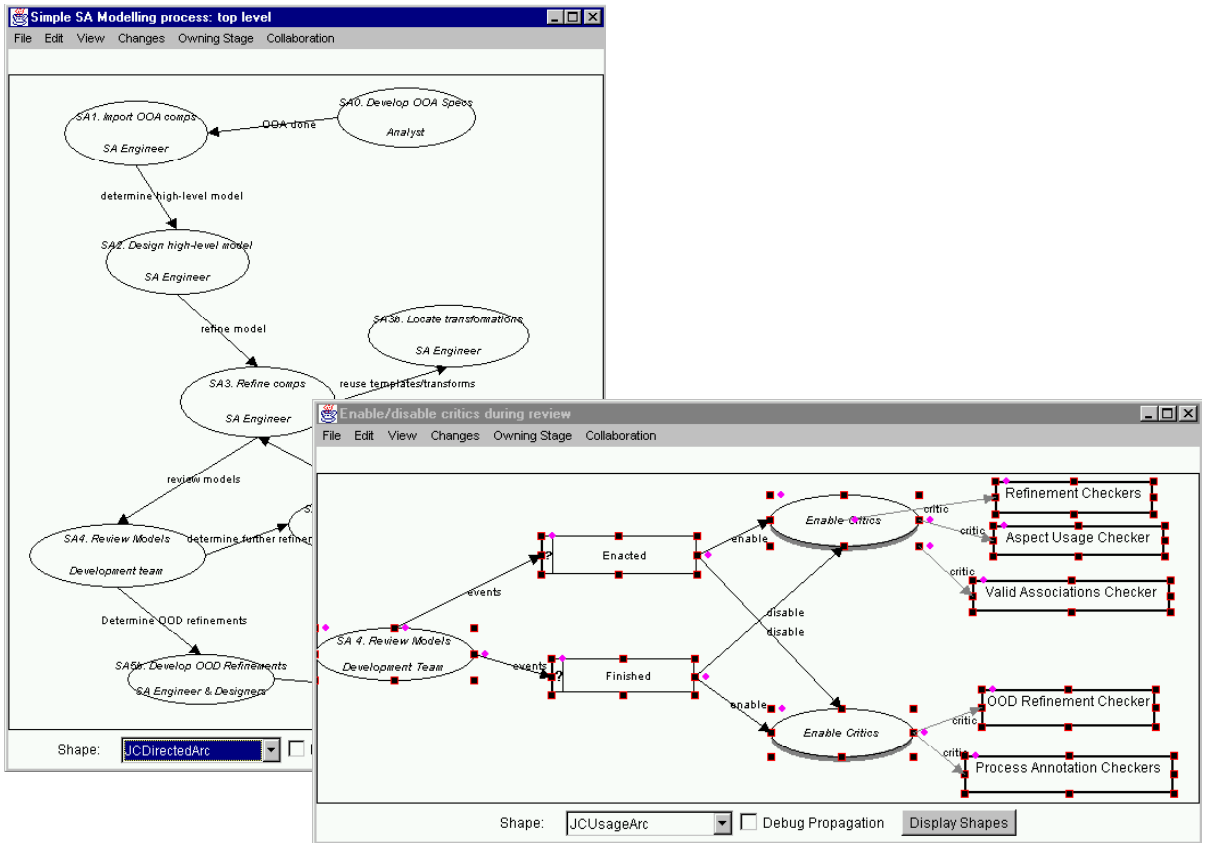


Figure 7. (a) Simple software process; (b) simple analysis co-ordination agent.

5.2. Design Constraints, Critics and Analysis Agents

SoftArch's meta-models have a set of analysis agents (implemented by event-driven JViews components) which monitor the state of the architecture model being developed. Agents may be fired immediate an invalid action is made e.g. incorrect association type specified between two architecture components, and the editing operation reversed and an error dialogue shown. Alternatively, they can monitor changes and unobtrusively add messages to an analysis report dialogue (like the one shown in Figure 2), or can be run on-demand by developers and their error messages displayed as a group. Users can control the way an analysis agent behaves using a control panel dialogue e.g. change an agent from running as a constraint to a critic, enable or disable agents etc. As show in Figure 7, Serendipity-II visually-specified task automation agents can also be used to control analysis agents.

Users can also extend the set of analysis agents being applied to a SoftArch model by using Serendipity-II's task automation agent specification tool. Figure 8 shows a user-defined analysis agent that checks to see if a component has associations (either from it to other components, or to it from other components). The top "guard" filters are fired when a component has been changed, and then following filters determine if the component has associations to/from it. If neither, an action (bottom oval icon) generates an error event which the analysis agent manager displays in an error dialogue (if this agent is run as a constraint) or displays in an analysis agent report.

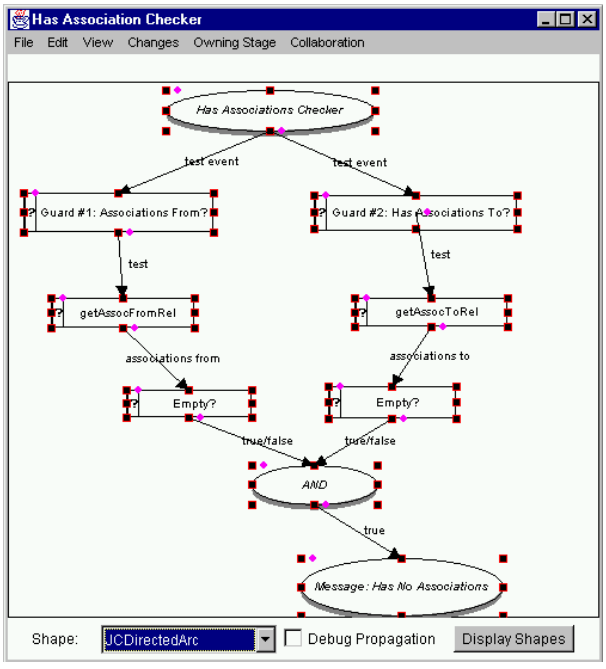


Figure 8. Simple visual analysis agent specification.

6. Tool Integration

7.1. OOA/D Import & Export

Many tools exist which provide object-oriented analysis and design capabilities. Our own JComposer is one such example, but others include CASE tools like Rational Rose [15] and Argo/UML [17]. We originally planned SoftArch as an extension to JComposer, but decided it would be more useful as a stand-alone tool, that could ultimately be used in conjunction with other, 3rd party CASE tools.

SoftArch requires constraints from an OOA model, particularly non-functional constraints like performance parameters, robustness requirements, data integrity and security needs and so on. These constrain the software architecture model properties that needs to be developed in order to realise the specification. These also influence the particular architecture-related design decisions and trade-offs software architects need to make. Similarly, a SoftArch architectural model is little use on its own, but needs to be exported to a CASE tool and/or programming environment for further refinement and implementation. Some code generation can even be done based on a SoftArch model description e.g. appropriate middleware and data management code generated. When reverse engineering an application, an OOD model will need to be imported into SoftArch and a higher-level system architecture model derived from it. Ultimately an OOA specification may be exported from SoftArch to a CASE tool. Thus SoftArch must support OOA and D model exchange with other tools, and ideally some code generation support.

We initially used a JComposer component model as the source for SoftArch OOA-level specification information. JComposer allows not only functional requirements to be captured, but has the additional benefit of requirements and design-level component "aspects", which are used to capture various non-functional requirements [9]. We developed a component that supports basic component and aspect import into SoftArch from a JComposer model, using JViews' inter-component communication facilities to link SoftArch and JComposer.

Rather than add OOD and code generation support to SoftArch itself, we leveraged existing support for these in JComposer. SoftArch uses JComposer's component API to create OOD-level components (classes) in JComposer, and instructs JComposer to generate code for these to produce .java files. JComposer supports a concept of code fragments, which SoftArch uses to generate some basic Java component configuration, communications and data access code for generated classes. Figure 9 illustrates the interaction of JComposer and SoftArch to achieve OOA import and OOD/P export for SoftArch.

JComposer was reasonably straightforward to integrate with SoftArch as JComposer provides a JViews-implemented, component-based API. Other CASE tools and programming environments do not generally provide

such open, flexible integration mechanisms. Generated .java class source code files can be used in tools like JDK and JBuilder, and changes reverse engineered back into JComposer and then into SoftArch. We have prototyped a data interchange mechanism to enable SoftArch to exchange OOA and D models with Argo/UML using an XML-based encoding of UML models. This is a less tightly integrated mechanism than that used by SoftArch and JComposer, but allows other tools using the XML exchange format for UML models to be integrated with SoftArch in the future.

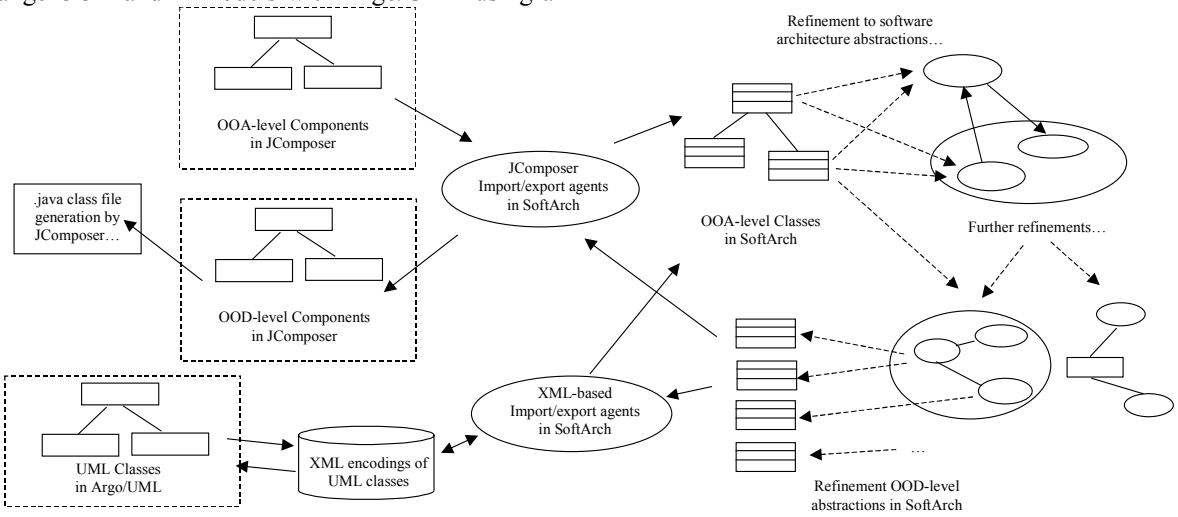


Figure 9. Import/export approaches in SoftArch.

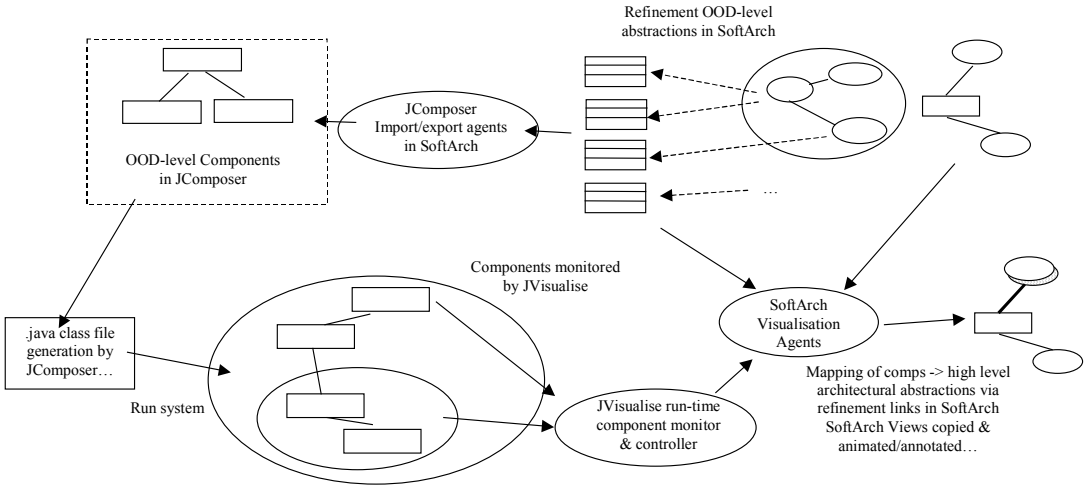


Figure 10. Planned dynamic architecture visualisation using SoftArch views.

7.2. Runtime Architecture Visualisation

So far we have discussed static architecture modelling, analysis and OOD/code generation support with SoftArch. Ultimately we would like to extend SoftArch's support for architecture modelling to include dynamic architecture visualisation and configuration i.e. run-time visualisation and manipulation of software architecture abstractions using SoftArch-style views. We are beginning work to achieve this by making use of our JVisualise component monitoring and configuration tool [7]. JVisualise allows running JViews-based systems to be viewed using JComposer-style visual languages. Users can also manipulate visualised components – changing their properties, adding or removing inter-component relationships, and creating new component instances. We intend to enhance JVisualise to enable any JavaBean-based application to be thus monitored and controlled.

Figure 10 illustrates how SoftArch will be used to visualise and configure running software architectures. JVisualise will request running components send it messages when they generate events, and will create proxies to enable it to intercept operation invocations on components. SoftArch will instruct JVisualise to send it these low-level component monitoring events, which will be mapped onto SoftArch OOD components using the JComposer-generated Java class names. SoftArch will then allow users to view information about running components using higher-level SoftArch views, as OOD-

level components will have refinement relationships to higher-level architecture components in these views. For example, when components implementing a server are created and the server initialised, SoftArch can show a single server component has started in a high-level SoftArch view. Similarly, when the server component receives a message from a client, SoftArch can annotate a high-level association link to indicate this. The user may add a client component to this dynamic visualisation view and connect it to the server. SoftArch can instruct JVisualise to create appropriate components which implement the client and initialise them.

JComposer-generated OOD models and code may be extended if necessary to include additional monitoring components and wrappers to intercept data and communication messages. JVisualise would use these to provide improved event and message monitoring and control support.

7. Discussion

A wide variety of tools and approaches exist with which to build a system like SoftArch. General-purpose programming languages and frameworks, such as Java and JFC, Borland Delphi, Smalltalk, or similar, can be used to implement such a tool "from scratch". However, many tool facilities required by SoftArch, including multiple views with consistency management, multi-user support, version control, persistency and distribution, and so on, are time-consuming to build using such approaches. In addition, building tools with extensible meta-models, visual languages and tool integration mechanisms with these low-level abstractions is extremely difficult.

General purpose drawing editor frameworks, such as Unidraw [21] and Hotdraw [2], could be used to provide editing support, and middleware architectures like CORBA [14], DCOM [18] and Xanth [11] used to support distribution and transparent persistency. Again, these technologies assist tool developers but still lack appropriately focused software tool building abstractions. An existing CASE tool, such as JComposer [7], MOOSE [4] or Argo/UML [17] could be extended to add SoftArch-style support. However, such an approach would make an already very complex tool more monolithic, the existing CASE tool infrastructure may not support some desired characteristics of SoftArch, and the resultant tool may not be usable with other 3rd party tools.

A variety of meta-CASE and CAME tools exist which might be usefully employed. Examples include KOGGE [3], MetaEDIT+ [12], MetaMOOSE [4], MOOT [16], and JComposer [7]. Tools like MetaEDIT+ and KOGGE provide a range of abstractions and tools enabling quick development of conventional CASE tools. Unfortunately they do not support well the need for users of SoftArch to extend architecture model abstractions and notations, do not provide adequate model analysis tool building support. MOOT and MetaMOOSE provide better support for extensible meta-models for software tools, and reasonably tailorable notations. However, they do not support template reuse well, and their analysis tool and tool integration capabilities are limited.

We found our JComposer tool to be of relatively limited usefulness in developing SoftArch. JComposer and its underlying framework, JViews, do not directly support the concept of an extensible tool meta-model, user-configurable icons for visual languages, patterns and templates, model analysis and process co-ordination, and flexible tool integration support. Process co-ordination and tool integration are provided by additional plug-in components (for example, the Serendipity-II process management tool for processes, and various components for database, remote server and XML data encoding use). This support could be improved to make build environments like SoftArch easier.

Allowing users to dynamically extend the meta-model of their enviroments, the visual languages they model with, the analysis tools and incorporate integration mechanisms with third-party tools are all very difficult in general. Our approach with SoftArch has been to build a JViews meta-model that has its own visual programming language, and have SoftArch use this model to validate architecture models. This proved challenging to realise, as JViews components designed for building software tools weren't built with a meta-model in mind, but rather a fixed, JComposer-generated model. Re-architecting both JViews and JComposer is required to provide suitable abstractions that make it easier to build such facilities. Similarly, while we developed the BuildByWire visual tool for iconic specification, this was not intended for use by tool users directly, but for tool developers. We need to modify the architecture of this to better-support end user configuration of iconic appearance, while retaining tool editing semantics.

We have built some reusable components in JViews which can be deployed for use in other environments to support analysis agent specification. We have also developed some basic agents in Serendipity-II that can be deployed by end users to extend the constraint and analysis checking of their tools while in use. However, these require further development to become easier to use by both tool developers and users. Similarly, our tool integration components built for SoftArch could be usefully generalised to make building file and XML-based tool integration easier. We are extending JViews' support for patterns and templates, and also extending JViews and JComposer to provide higher-level dynamic monitoring to better support visualisation of running SoftArch-modelled systems.

Alternative approaches to building SoftArch might have used a meta-CASE tool which allows end users to extend a meta-model and/or visual notation. However, most meta-CASE tools, like JViews, assume tool developers specify such meta-level constructs, rather than tool users. Another approach would be to use tools designed for end user computing, somewhat like Serendipity-II's process modelling and agent specification

tools. In fact, we originally explored building most of SoftArch using Serendipity-II in this fashion. Unfortunately the abstractions supported by such an approach for SoftArch-style notations, architecture models and analysis are very difficult to express in such end user computing tools, and the efficiency and extensibility of the resulting solution likely to be poor.

8. Summary

We have described the construction of the SoftArch software architecture modelling and analysis tool. SoftArch requires a number of facilities that are challenging to build using conventional tool development approaches. We achieved the aim of an extensible set of modelling abstractions and notations by using a user-extensible meta-model and set of user-customisable icons. Reusable refinement templates are supported by SoftArch, leveraging component import/export and version merging capabilities of our tool implementation framework. Process support, including work co-ordination and user-defined analysis agents, are supported by integrating SoftArch with the Serendipity-II process management environment. OOA/D import/export and code generation and reverse engineering support are provided by integrating SoftArch with the JComposer component engineering/meta-CASE environment and the Argo/UML CASE tool.

We are investigating extending our JComposer meta-CASE toolset to better support meta-models for software development tools, and to provide abstractions for template and pattern reuse. In addition, we are investigating other process management tool integration approaches, such as the workflow management coalition's process interchange format. We are also investigating other interchange formats for CASE tools and programming environments, allowing more OOA specification information, especially non-functional requirements codification, to be exchanged, along with improved OOD and code generation facilities. We are beginning to develop an exploratory dynamic architecture visualisation and configuration facility, using SoftArch and the JVisualise component monitoring tool.

References

1. Bass, L., Clements, P. and Kazman, R. Software *Architecture in Practice*, Addison-Wesley, 1998.
2. Beck, K. and Johnson, R. Patterns generate architectures. *Proceedings ECOOP'94*, Bologna, Italy, 1994.
3. Ebert, J. and Suttenbach, R. and Uhe, I. Meta-CASE in practice: A Case for KOGGE, *Proceedings of CaiSE*97*, Barcelona, Spain, June 10-12 1997, LNCS 1250, Springer-Verlage, pp. 203-216.
4. Furguson, R.I., Parrington, N.F., Dunne, P., Archibald, J.M. and Thompson, J.B. MetaMOOSE – an Object-oriented Framework for the Construction of CASE Tools, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 19-32.
5. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, Vol. 2, No. 5, September/October 1998, IEEE CS Press.
6. Grundy, J.C. and Hosking, J.G. Directions in modelling large-scale software architectures, In *Proceedings of the 2nd Australasian Workshop on Software Architectures*, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
7. Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Constructing component-based software engineering environments: issues and experiences, *Journal of Information and Software Technology: Special Issue on Constructing Software Engineering Tools*, Vol. 42, No. 2, January 2000, pp. 117-128.
8. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
9. Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In *Proceedings of the 4th IEEE Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, IEEE CS Press, pp. 84-91.
10. Grundy, J.C. *Software Architecture Modelling, Analysis and Implementation with SoftArch*, Technical Report, Department of Computer Science, University of Auckland, December 1999.
11. Kaiser, G.E. and Dossick, S. Workgroup middleware for distributed projects, *Proceedings of IEEE WETICE'98*, Stanford, June 17-19 1998, IEEE CS Press, pp. 63-68.
12. Kelly, S., Lyytinen, K., and Rossi, M., "Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment," In *Proceedings of CAiSE'96,* Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.
13. Leo, J. OO Enterprise Architecture approach using UML, In *Proceedings of the 2nd Australasian Workshop on Software Architectures*, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
14. Mowbray, T.J., Ruh, W.A. *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley, 1997.
15. Quatrani, T. *Visual Modelling With Rational Rose and UML*, Addison-Wesley, 1998.
16. Phillips, C.E. , Adams, S., Page, D. and Mehandjiska, D., Designing the client user interface for a methodology independent OO CASE tool, *Proceedings of TOOLS Pacific'98*, Melbourne, Nov 24-26, IEEE CS Press.
17. Robbins, J.E. and Redmiles, D.F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 61-70.
18. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
19. Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
20. Thomas, I. and Nejmeh, B. Definitions of tool integration for environments, *IEEE Software*, vol. 9, no. 3, March 1992, 29-35.
21. Vlissides, J.M. and Linton, M.A. Unidraw: a framework for building domain-specific graphical editors, *ACM Transactions on Information Systems*, vol. 8, no. 3, July 1990, 237-268.

STEP-based CASE Tools cooperation

Alain Plantec [1] and Vincent Ribaud [2]

[1] *SYSECA, 34 quai de la Douane, 29285 Brest Cedex, France,*
alain.plantec@syseca.thomson-csf.com
[2] *EA2215-LIBr, Faculté des Sciences, BP 809, 29285 Brest Cedex, France,*
ribaud@univ-brest.fr

Abstract

Computer-Aided Software Engineering (CASE) tools need to cooperate and this can be accomplished by exchanging or sharing meta-data stored in a repository.

STEP is an ISO 10303 standard developed to facilitate product information sharing by specifying sufficient semantic content for data and their usage. STEP is providing a dedicated technology, mainly an object oriented modeling language *EXPRESS* and a standardized data access interface *SDAI*.

Meta-modeling the repository in EXPRESS allows a facilitated cooperation. Both exchange and sharing are provided by the SDAI generated from the EXPRESS meta-schema. Some experiments are related and an industrial project is depicted. Designer/2000 modeling is jointly used with dedicated Visual Basic code generators. Consistency is needed between these two tools families. This is achieved with a simple tool, but the use of the experimental method proposed is still difficult. Impedance mismatch between relational and object database paradigms may be the origin of the difficulties.

Keywords : CASE tools interoperability, CASE tools implementation, STEP standard, SDAI, EXPRESS

Introduction

CASE tools assist system development in managing system documentation. Documentation is structured with the help of various models, elaborated throughout the system development cycle. Information on the different models are the data (in fact meta-data) processed by the CASE tools. Cooperation of CASE tools rely on common meta-data access. This kind of cooperation is described as a data integration in [12].

CDIF (*CASE Data Interchange Format*) [3] and IRDS (*Information Resource Dictionary System*) [6] are two examples of proposals intended to facilitate the cooperation of CASE tools and the exchange of models between the vendor's tools.

In early 90's, CDIF and IRDS are the major representatives of the two approaches used to (meta-)data integration : exchange of meta-data files or sharing through a common repository. These approaches are still valid today, although the technology slightly differs (e.g. use of marked-up language such as XMI or dedicated API).

One major component of a CASE tool is the repository. A repository holds the system documentation in a central place online. Various tools pick information in the repository, process them and store the results in the repository. The structure of data in the repository is often referred as the *meta-model*. The repository itself is usually implemented using either a relational or an object-oriented database management system.

STEP is an ISO 10303 standard developed to facilitate product information sharing by specifying sufficient semantic content for data and their usage. Parts of ISO 10303 are intended to standardize conceptual structures of information which are either generic or within a subject area (e.g. mechanics). Standardized parts are expressed with a dedicated technology, mainly an object-oriented modeling language called EXPRESS and a standard data access interface called SDAI.

As mentioned in the STEP box, the SDAI is a functional interface for EXPRESS-modeled database and is independent of any particular system and language. The SDAI allows data sharing as well as data exchange. The key point is that a SDAI is automatically generated from the EXPRESS schema of the database (as long as an SDAI generator has been made for the target database management system).

STEP description and implementation methods

The EXPRESS language [1] is an object-oriented modelling language. The application data are described in schemata. A schema has the type definitions and the object descriptions of the application called *Entities.* An entity is made up of attributes and constraint descriptions.
The constraints expressed in an entity definition can be of four kinds: **(1)** the *unique* constraint allows entity attributes to be constrained to be unique either solely or jointly, **(2)** the *derive* clause is used to represent computed attributes, **(3)** the *where* clause of an entity constraints each instance of an entity individually and **(4)** the *inverse* clause is used to specify the inverse cardinality constraints. Entities may inherit attributes and constraints from their supertypes.

The STEP physical file format defines an exchange structure using a clear text encoding of product data for which a conceptual model is specified in the EXPRESS language. The mapping from the EXPRESS language to the syntax of the exchange structure is specified in [2].

The Standard Data Access Interface (SDAI) [3] defines an access protocol for *EXPRESS*-modelled databases and is defined independently from any particular system and language. The representation of this functional interface in a particular programming language is referred to as a language binding in the standard. As an example, ISO 10303-23 is the STEP part describing the C++ SDAI binding [4].
The five main goals of the SDAI are: **(1)** to access and manipulate data which are described using the EXPRESS language, **(2)** to allow access to multiple data repositories by a single application at the same time, **(3)** to allow commit and rollback on a set of SDAI operations, **(4)** to allow access to the EXPRESS definition of all data elements that can be manipulated by an application process, and **(5)** to allow the validation of the constraints defined in EXPRESS.
An SDAI can be implemented as an interpretor of EXPRESS schemata or as a specialized data interface. The interpretor implementation is referred to in the standard [3] as the SDAI late binding. An SDAI late binding is generic in nature. The specialized implementation is referred to in the standard as the SDAI early binding.

References

[1] ISO 10303-11. *Part 11: EXPRESS Language Reference Manual*, 1994.

[2] ISO 10303-21. *Part 21: Clear Text Encoding of the Exchange Structure*, 1994.

[3] ISO DIS 10303-22. *Part 22: Standard Data Access Interface*, 1994.

[4] ISO CD 10303-23. *Part 23: C++ Programming Language Binding to the SDAI Specification*, 1995.

This paper argues that given a CASE tool, data interoperability can be accomplished through an SDAI generated from the EXPRESS schema resulting from the meta-model used in the CASE tool. Benefits of this method include data exchange as well as data sharing, allowing system developers to use best suited CASE tools to their projects, even if they belong to different CASE toolsets. However, complex repository causes a complex meta-modeling and the resulting SDAI can be difficult to use.

The paper is organized as follows: an example of different data integration is described in section 1. Section 2 shows how different CASE tool were needed and used in a commercial system. Then we finish with perspectives and a conclusion.

1 Examples of data integration

1.1 UML

Within the context of a research project, colleagues were faced to use jointly two kinds of CASE tools: a UML tool and a SDL tool. The cooperation should be the following: an UML tool will be used to design class diagrams and collaboration diagrams. SDL code will be generated from both diagrams and then imported into the SDL tool.

Within another research project, a colleague wished to use UML to design class diagrams and then generate a SmallTalk-80 implementation. Unfortunately, he didn't find any UML tool able to generate SmallTalk-80 code.

We started two different projects of two persons within the context of final-year course-work (bachelor students). We chose Argo/UML from Jason Elliot Robbins [10] for its open-implementation and its conformity to the UML Meta-model 1.1 [1]. Moreover, Argo/UML allows the two types of data integration mentioned above : a set of Java classes providing an API (*Application Programming Interface)* to the meta-data as well as a file exchange format (*.xmi*).

Meta-programming with an API For the cooperation between Argo/UML and SDT [11], a SDL tool, meta-programming with ARGO API was chosen. Argo/UML does not use a database management system to store information about diagrams. Hence in order to share meta-data with the class and collaboration diagrammers, students [4] incorporated a SDL generator in Argo/UML. This generator was written in Java.

Part of the time devoted to the project has been used to understand the UML meta-model (available only in a

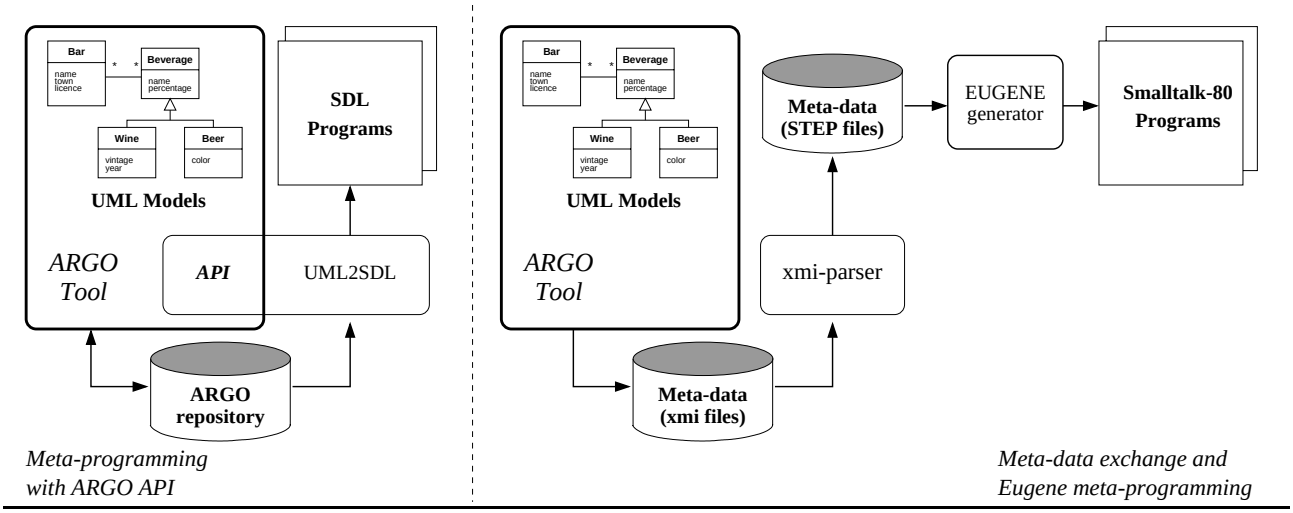


Figure 1. Meta-programming with ARGO API *versus* meta-data Exchange and Eugene meta-programming

graphical UML form) and to learn the use of the API (formed by a total of 120 classes) and the way it matches the meta-model. Then the students were able to use the API to write their SDL generator.

Meta-data exchange In the second project, meta-data exchange between Argo/UML and the generator was the solution we kept. The generator was built with *Eugene*, our STEP-based application generators builder [9]. *Eugene* is used within the context of research projects at Brest University and also in industrial projects in Syseca, a software company.

Like the first project, part of the time was devoted to UML meta-model understanding. Building an application generator with Eugene requires an EXPRESS description of the meta-model of the generator inputs (here a *.xmi* file) and students did it [5]. Then a meta-program was written in order to generate Smalltalk-80 code from meta-data.

Discussion We cannot compare the time devoted to real development in each project. The SDL generator was written without any meta-environment whereas the SmallTalk-80 generators uses that type of environmment. But there were two successive phases in both projects, i.e. learning the system (API or meta-model) and programming. Two points should be noted:

- Learning an API is an experimental task, and no learning method can be provided. Consistency in the naming of elements and operations in the API helps to make learning and use more efficient.

  The use of Eugene implies writing of a schema of the meta-model. The learning phase is in fact a meta-modeling phase. This activity helps the students in the learning of UML meta-model.

- Programming an API depends on the API itself. Little experience can be re-invested in another API.

  Meta-programming is based on the meta-modeling phase, and another project will require another meta-modeling activity. So some meta-modeling experience will grow from a project to another.

1.2 STEP use

Cooperating with a CASE tool is made easier if the CASE tool provides an access to meta-data (API, meta-data files or others formated outputs). Experience gained from the above projects enables us to provide a method (supported by a tool, an SDAI generator) to write a CASE tool intended to cooperate with an existing CASE tool (see fig. 2):

Meta-modeling The structure of the existing (source) CASE tool repository is modeled with EXPRESS schemata.

SDAI generation An SDAI for the management system running the new CASE tool (called the *target system* below) is generated. This requires naturally an SDAI generator suited to the target system, but such an SDAI generator is re-used for each CASE tool available within this target system.

The SDAI is useful for each source CASE-tools: meta-data produced by a source CASE-tool are imported into the new CASE tool. For such a task, a specific program (i.e. a program parser or a meta-data converter) is implemented. An SDAI can be generated within the source system and used for this implementation.

CASE tool development The development of the new CASE tool is based on the SDAI, which provides a standard access to the meta-data exported from the

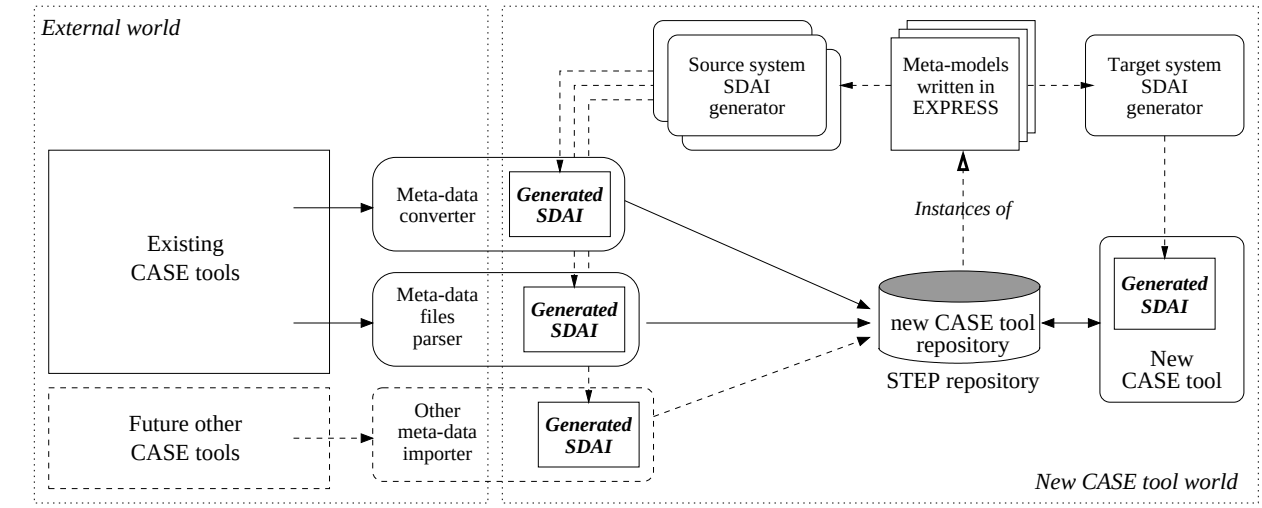


Figure 2. Using generated SDAI to interoperate with a given CASE tool

existing CASE tool and managed by a STEP repository.

2 Working with different CASE tools

2.1 The context

At Syseca Brest, a small team (3-6 persons) has been developing new software within a global project named *ARIANE* : the management of the textile department of a supermarket chain. Technical choices made at the beginning of the project (1995) and still valid are Oracle7 (now Oracle8) for the database management system and VisualBasic and SQL for the client software. System analysis and design is done with the help of Designer/2000; the repository is continually updated and SQL DDL code (the database schema) used in the project is always obtained by the code generators of Designer/2000.

Since 1998, a part of the team's effort has been devoted to developing and maintaining a family of VisualBasic generators, called *GARI* (for Generator ARIane). Eugene is the environment used to build the generators. Input to these generators are either SQL select statements or EXPRESS schemata hand-made from Designer/2000 information.

2.2 Designer/2000

Oracle Designer/2000 is a suite of software toolsets for designing Windows-based client/server applications that interact with an Oracle database. Designer/2000 incorporates support for busines process modeling, system analysis, software design and code generation [8]. Designer/2000 provides a multi-user repository implemented using Oracle's RDBMS. The repository consists of tables that store information on the system we are analysing, designing and producing. A good introduction to Designer/2000 software toolsets and also a software development method using these tools can be found in [2]. Designer/2000 provides an Application Programming Interface (API) to the repository. The API is a set of database views and PL/SQL packages that allow safe access to the repository data (meta-data).

2.3 Visual Basic Generators

The GARI family is used throughout the projects. Some generators use EXPRESS schema as inputs and still produce VB code. These schemata need to be hand-written from the meta-data of the repository. They may include entities and their attributes or tables and their columns, all of which have individual properties useful for the generators. The re-writing in EXPRESS schema of the information still present in the Designer/2000 repository is a tedious task, prone to errors and requiring repeated efforts to maintain the mapping between Designer/2000 information and VB code generated.

So the problem lies on providing inputs to GARI generators with a guaranteed and automatic consistency with the Designer/2000 repository. This will provide a seamless integration of all CASE tools used in the project.

2.4 Possible solutions

2.4.1 Generating EXPRESS from repository data

A first solution will be making a translation tool able to produce EXPRESS schema from the repository meta-data. The seamless integration is obtained through three steps (see our current implementation depicted in figure 3): analysis and design using Designer/2000 tools, generation of the EXPRESS schema, generation of the VB code with the GARI family.

Pros and cons It took three weeks to make the above translation tool (called *Malam*) [7]. It works as expected and provides consistency. But this consistency is possible because there is no semantic loss between the information needed in the repository and the translation in EXPRESS.

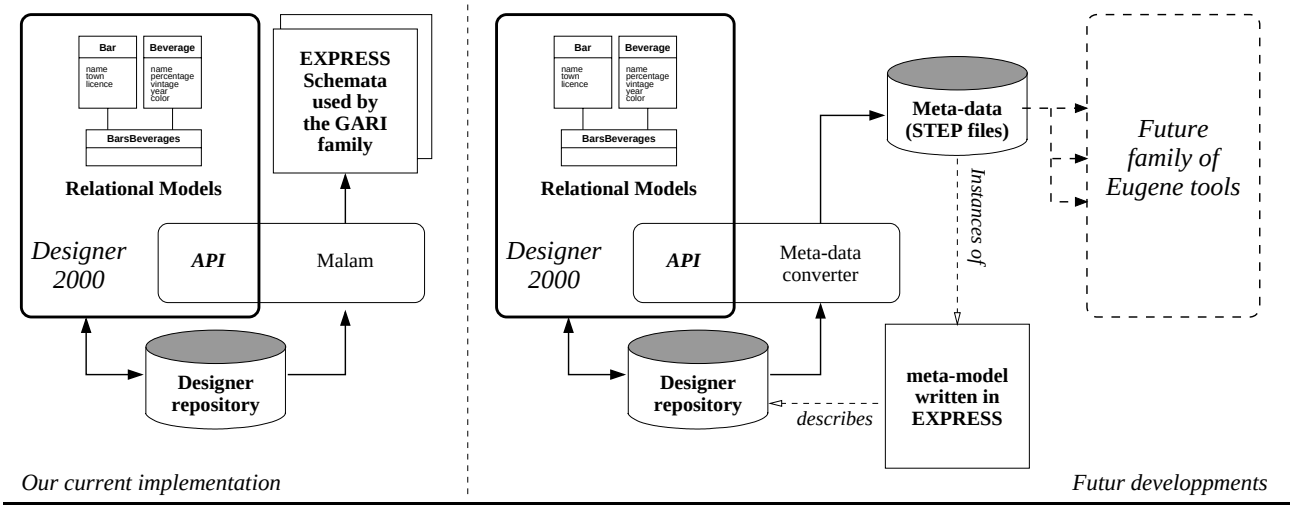


Figure 3. Cooperation between Designer/2000 and GARI with an intermediate translation tool

We are working essentially on table definition which are easy to translate.

2.4.2 Meta-modeling the repository structure in EXPRESS

As stated in conclusion of section 1, cooperating with Designer/2000 will be made easier by an SDAI operating on the repository, this SDAI being generated from a meta-model of the repository.

Since the repository is a standard SQL database, the translation tool above depicted in 2.4.1 can be used to produce automatically an EXPRESS schema of the repository structure. We did so but we now need to refine the schema. As a matter of fact, the repository consists of a relatively small number of tables that store the meta-data. These tables have complex (undocumented) relationships. There are, however, many views of these tables that represent repository objects, such as entities and attributes. These views are an important part of the API because they allow us to examine the definition of objetcs created through the toolsets [2]. Unfortunately, if translating automatically SQL DDL statements is straightforward, this is not true with SQL DML statement, specially if they are complex.

Pros and cons The generated schema contains more than 5000 EXPRESS statements. Generating an SDAI for this schema provides a complex API, usable in many situations. Until now, we haven't built new tools that will use this SDAI. Intuitively, we expect that using this SDAI will be so complicated that it requires company investments.

3 Perspectives

Perspectives depends on the quality and the readibility of the Designer/2000 repository meta-modeling. Commercial tools often change but our experience with Oracle CASE tools indicates that the repository (formerly named Case*Dictionary in previous version of Oracle CASE) is stable, at least for the analysis and design phase. So, we are pursuing our efforts in repository understanding and meta-modeling refinements. The data-flow between Designer/2000 and the futur family of tools is depicted in figure3.

The difference of paradigm between a relational database (the repository) and an object-oriented schemata causes some problems, which may not be solved automatically.

4 Conclusion

We need a cooperation between different CASE tools, especially if we wish to guarantee consistency. This requires access to the CASE tool repositories. STEP is an ISO standard (ISO-10303) for the computer-interpretable representation and exchange of product data. We successfully used STEP framework to produce SDAI automatically from the repository meta-modeling, and using this standard meta-data access more easily than the dedicated repository API. However, when the repository structure is complex, following this approach requires investments. In fact it depends on the quality of the meta-model. Hence in some situations, dedicated translation tools using the repository API are easier to develop.

References

[1] UML metamodel 1.1. Technical report, Object Management Group, 1997.

[2] Paul Dorsey and Peter Koletzke. *Designer/2000 Handbook*. McGraw-Hill, 1998.

[3] EIA. *CDIF - Framework for Modeling and Extensibility*, 1994.

[4] Divi Lainé et Armelle Prigent. Modélisation UML et SDL dans le développement des systèmes temps-réel. Technical report, Université de Bretagne Occidentale, 1999.

[5] Céline Courbalay et Jean-Marc Douarinou. Traducteur d'UML vers SmallTalk-80. Technical report, Université de Bretagne Occidentale, 1999.

[6] ISO/IEC 10027. *Information technology - Information Resource Dictionary System (IRDS) framework*, 1990.

[7] Mikael Le Moal. Intégration d'oracle designer dans smalltalk-80. Technical report, Université de Bretagne Occidentale, 1999.

[8] Oracle. *Oracle Designer/2000 A Guide to Repository Administration*, 1995.

[9] Alain Plantec. *Exploitation de la norme STEP pour la spécification et la mise en œuvre de générateurs de code*. PhD thesis, Université de Rennes I, 35065 Rennes cedex, France, 1999.

[10] Jason Elliot Robbins. Argo/UML. http://www.ics.uci.edu/pub/c2/uml/index.html.

[11] Telelogic. SDT. http://www.telelogic.com.

[12] A. I. Wasserman. Tool Integration in Software Engineering Environments. In *Lecture Notes in Computer Science, Software Engineering Environments*, pages 137–149. Springer-Verlag, 1989.

A Pretty-Printer for Every Occasion

Merijn de Jonge

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Merijn.de.Jonge@cwi.nl

Abstract

Tool builders dealing with many different languages, and language designers require sophisticated pretty-print techniques to minimize the time needed for constructing and adapting pretty-printers. We combined new and existing pretty-print techniques in a generic pretty-printer that satisfies modern pretty-print requirements. Its features include language independence, customization, and incremental pretty-printer generation.

Furthermore, we emphasize that the recent acceptance of XML as international standard for the representation of structured data demands flexible pretty-print techniques, and we demonstrate that our pretty-printer provides such technology.

Keywords: documentation, languages, XML, tool construction, software engineering

1. Introduction

Pretty-printing is concerned with formatting and presentation of computer languages. These languages include ordinary programming languages and languages defining data structures. XML [6], recently accepted as international standard for the representation of structured data, brings formatting issues (related to the transformation of XML documents to user-readable form) towards a broad community of tool builders.

These tool builders as well as language designers demand advanced pretty-print techniques to minimize the time required for developing new or adapting existing pretty-printers. For both it is essential to maximize language independence of pretty-printers and to be able to add support for new languages easily. Moreover, pretty-printers should minimize code duplication, be customizable, extensible, and easy to integrate.

Most pretty-print technology used in industry today does not meet these requirements. This lack of sophisticated technology makes development and maintenance costs of pretty-printers high. Despite the academic research in this field which has yielded advanced pretty-print techniques, we observe that these techniques have not come available for practical use yet.

In this paper we combine new and existing techniques to form a pretty-print system that satisfies modern pretty-printer requirements. It features language independence, extensibility, customization, pretty-printer generation, and it supports multiple output formats including plain text, HTML, and LaTeX. Furthermore, the pretty-printer can easily be integrated in existing systems and is freely available.

This article is organized as follows. Section 2 describes several aspects of pretty-printing by summarizing earlier work in this field. In Section 3 we describe the design and implementation of the generic pretty-printer GPP. Several case studies are discussed in Section 4. Section 5 explains how our pretty-printer can be used to format XML documents depending on their document type definition (DTD) and how it may function as alternative to the extensible style language (XSL). Contributions and future work are addressed in Section 6.

2. State of the art

Traditionally, mostly ad-hoc solutions have been used to cope with the problem of formatting computer languages. Not only were traditional pretty-printers bound to specific languages, they also contained hard-coded formatting rules which made them non-customizable.

The first general solution to the pretty-print problem was formulated by Oppen [20]. He described a *language independent* pretty-print algorithm operating on a sequence of logically continuous blocks of strings. The division of the input by delimiters (either block delimiters or white space) provides information about were line breaks are allowed.

Oppen also introduced *conditional formatting* to support different formattings when a block cannot fit on a single line. He distinguishes inconsistent breaking, which minimizes the number of newlines that are inserted in a block to make it fit within the page margins, and consistent breaking, which maximizes the number of newlines. Conditional formatting has been adopted in most modern pretty-printers.

In addition to Oppen, many language independent pretty-print *algorithms* are described in the literature. Traditional algorithms which are more or less similar to Oppen's include [23, 18, 24, 19, 30]. A consequence of conditional formatting is an exponential growth of the pos-

sible formattings. While the traditional algorithms only consider a small subset of these formattings in order to limit execution time, more advanced formatting algorithms are designed in the community of functional programming [12, 26, 14, 34]. These algorithms heavily depend on lazy evaluation to abstract over execution time. This allows the pretty-printers to select an optimal formatting in a lazy fashion from all possible ones.

Several formatting primitives have been suggested as alternative to the blanks and blocks of Oppen. Modern pretty-printers describe formatting in terms of *boxes* (as introduced by [15] and [18]). PPML[19] defines a formalism based on boxes to define the structuring of displays. It introduces different types of boxes for different formatting. Examples are the `h` box for horizontal formatting and `v` for vertical formatting. Based on PPML, [30] introduces the language BOX, mainly to solve some technical problems of PPML. Another similar approach to PPML is described by Boulton [5]. He describes a formalism to annotate a grammar with, among others, abstract syntax and formatting rules. The syntax for specifying formatting is based on PPML.

Oppen [20] observed that the process of pretty-printing can be divided in a language dependent *front-end* for the translation of a program text to some language independent formatting, and a language independent *back-end* which translates the language independent formatting to an output format. All current pretty-printers that we are aware of follow this structure.

The division of a pretty-printer in a front-end and back-end not only makes a back-end language independent, it also makes a front-end output format independent. Despite this fact, by far the most back-ends that are described in the literature concentrate on the translation from a language independent input term to plain text. Articles which address the translation to other output formats include [19, 30, 27].

A nice formatting is a question of style and personal taste [16]. Blaschek and Sametinger [4] emphasize that the ability to *customize* the generated output of a pretty-printer to one's favorite style can improve the readability and maintainability of programs significantly. Customizing existing pretty-printers mostly requires changing the code manually, or modifying the formatting rules as annotations of the grammar (which, as a result, also modifies the grammar). An ordinary user cannot be expected to perform such modifications. A more user-friendly approach of customizable pretty-printing is described by [4]. They introduce user-adaptable pretty-printing using personal profiles which provide individual formatting rules for general language constructs.

A front-end for a language can be constructed by hard-coding the formatting rules manually, or be generated from a grammar annotated with formatting rules. The first approach is most commonly used, for example in [13, 19]. The latter approach, suggested by Oppen (who emphasized the importance of separating pretty-print information from code), is used in [23, 24, 5].

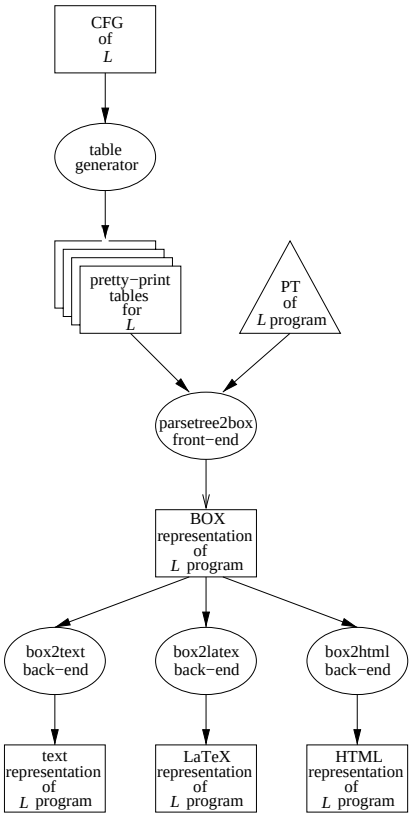


Figure 1. An overview of the generic pretty-printer GPP. It consists of a table generator, a front-end (`parsetree2box`), and three back-ends which produce plain text, HTML and LaTeX, respectively.

A front-end can also be generated from a grammar without annotated format rules by a pretty-printer *generator* that analyses the structure of a grammar to "guess" a suitable layout. Despite the usefulness of such generators in environments where a large number of evolving languages are used, little work has been carried out on this topic. The only pretty-printer generator that we are aware of is described in [30]. They describe a generator which produces dedicated, language specific front-ends. These front-ends contain formatting rules and the code to perform the formatting. The actual formatting can be customized by adapting or extending the generated code. Their approach yields highly customizable formatters but the formatters are language dependent and customization requires modifying the generated code (and thus requires understanding the generated code).

3. A pretty-printer for every occasion

Despite all research on the topic of pretty-printing, most pretty-printers that are used in practice are language specific, inflexible, and support only a very restricted number of output formats. Moreover, for many languages not even

a pretty-printer exists. Adding support for a new language or a new output format often means implementing a new pretty-printer from scratch. This is not only a time consuming task, but also introduces much code duplication which increases maintenance costs.

On the other hand, more advanced pretty-printers that have been developed as part of research projects are often incomplete (because they only address a limited number of pretty-print aspects), or are tightly coupled to a particular system [19, 30] which make them hard to use in general.

Summarizing, there is a great need for advanced pretty-print techniques in industry which are flexible, customizable, easy to use, and language independent. Despite the research in this field there are currently no such pretty-printers for practical use available.

In the remainder of this section we will describe the architecture and design of the generic pretty-printer GPP which satisfies modern pretty-print requirements. The pretty-printer is language independent and divided in front-ends and back-ends to make future extensions easy to incorporate. A box based intermediate format (called BOX), which supports comment preservation and which is prepared for incremental and conservative pretty-printing [25], is used to define the formatting of languages and to connect front-ends with back-ends. Furthermore, the pretty-printer uses new techniques to support customization of pretty-printers (based on re-usable, modular pretty-print tables), and incremental pretty-printer generation. We support multiple output formats including plain text, HTML, and LaTeX. Finally, the pretty-printer can be integrated easily in existing systems, or be used stand-alone and is freely available. Figure 1 gives a general overview of the architecture of GPP.

3.1. An open framework for pretty-printing

We followed the well-known approach of dividing a pretty-printer in a language dependent *front-end* and a language independent *back-end*. This allows for an open pretty-print system which can easily be extended to support new languages and output formats. A front-end for language L expresses the language specific layout of L in terms of a generic formatting language. A back-end producing output format O translates terms over this formatting language to O. A pretty-printer for L producing O as output can now be constructed by connecting the output of the L specific front-end to the input of the back-end for O. This architecture thus isolates language specific code in the front-end and output format dependent code in back-ends. Adding support for a new language only requires developing a new front-end for the language, likewise, to add support for a new output format, only a new back-end has to be developed.

We used the domain specific language BOX [30] to connect the output of front-ends to the input of back-ends (see Section 3.2 for a description of the BOX language). By using BOX to glue front-ends and back-ends, the framework allows any BOX producer to be connected to any BOX consumer. This flexibility allows a whole range of front-ends and back-ends of different complexity to be connected to the pretty-print framework. For example, multiple front-ends for a single language may exist simultaneously, providing different functionality or different quality. One of them might be optimized for speed, performing only basic formatting for instance, while another is designed to produce optimal results at the cost of decreased performance.

operator	*options*	*description*
H	hs	Formats its sub-boxes horizontally.
V	vs, is	Formats its sub-boxes vertically.
HV	hs, vs, is	Inconsistent line breaking. Respects line width by formatting its sub-boxes horizontally and vertically.
A	hs, vs	Formats its sub-boxes in a tabular.
ALT		Depending on the available width, formats its first or second sub-box.

Table 1. Positional BOX operators and supported space options (hs defines horizontal layout between boxes, vs defines vertical layout between boxes, and is defines left indentation).

3.2. The box markup language

BOX is a language independent markup language designed to describe the intended layout of text. Being a box-based language, it allows a formatting of text to be expressed as a composition of horizontal and vertical boxes. BOX is based on PPML[19] and contains similar operators to describe layout and conditional operators to define formatting depending on the available width. In addition to PPML, BOX supports tables, fonts, and formatting comments. In the remainder of this section we will give a brief overview of BOX (for a more complete description of the language we refer to [27]).

A term over the BOX language consists of a nested composition of boxes. The most elementary boxes are strings, more complex boxes can be constructed by composing boxes using *positional operators* and *non-positional operators*. The first (see Table 1 for a list of available positional operators) specify the relative positioning of boxes. The latter (see Table 2) specify the visual appearance of boxes (by defining color and font parameters), define labels, and format comments.

Examples of positional operators are the H and V operators, which format their sub-boxes horizontally and vertically, respectively:

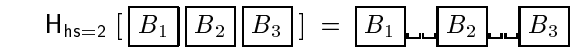
$$\mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \;=\; \boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}$$

$$\mathsf{V}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \;=\; \begin{array}{|c|}\hline B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline \end{array}$$

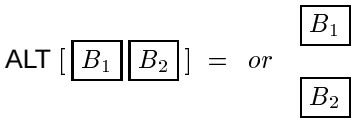
operator / *description*	
F	Operator to specify fonts and font attributes.
KW	Font operator to format keywords.
VAR	Font operator to format variables.
NUM	Font operator to format numbers.
MATH	Font operator to format mathematical symbols.
LBL	Operator used to define a label for a box.
REF	Operator to refer to a labeled box.
C	Operator to represent lines of comments.

Table 2. Non-positional BOX operators.

The exact formatting of positional box operators can be controlled using *space options*. For example, to control the amount of horizontal layout between boxes, the H operator supports the hs space option:



BOX as we use it slightly differs from its initial design as described in [30]. We simplified the language (mainly to improve comment handling) and made it more consistent. Furthermore, we introduced a generalization of the conditional HOV operator. This operator, which is available in some form or another in most formatting languages, formats its contents either completely horizontally or completely vertically depending on the available width (consistent line breaking). We introduced as generalization the ALT operator:



This operator chooses among two alternative formattings depending on the available width. It chooses for its first sub-box when sufficient space is available and for its second sub-box otherwise.

3.3. Pretty-print tables

We introduce the notion of interpreted formatting in which a front-end (see Section 3.4) formats its input by interpreting a set of language specific formatting rules. Formatting rules and code are separated by defining the formatting rules in pretty-print tables. Each formatting rule forms a mapping of the form $p_L \longrightarrow b$ (where p_L denotes a production of the grammar of the language L and b denotes the corresponding BOX expression) and specifies how the language construct p_L should be formatted.

Representing formatting rules in tables instead of having a single dedicated pretty-printer that contains all pretty-print rules for a language provides the following advantages. First, tables support a modular design of pretty-printers. As a consequence, a pretty-printer can follow the same modular structure as the corresponding modular grammar and re-use is promoted. Second, pretty-print tables promote incremental pretty-printer generation. When one or more modules of a modular grammar are modified, only the tables corresponding to the modified modules have to be re-generated. Third, tables allow easy personal customization by separating globally defined or generated formatting rules, and customized rules in different tables. Defining an ordering on tables determines which formatting rule should be applied when multiple rules exist for a single language construct. It allows a user to customize the pretty-printer by defining additional rules with higher precedence. Fourth, the separation of formatting rules in tables allows for a generic BOX producer which, when instantiated with language specific pretty-print tables, performs language specific formatting (see Section 3.4).

```
"package" Name ";" → PackagedDeclaration —
    H [KW["package"] H hs=0 [_1 ";"]],
"import" Name ";" → ImportDeclaration —
    H [KW["import"] H hs=0 [_1 ";"]],
"import" Name "." "*" ";" → ImportDeclaration —
    H [KW["import"] H hs=0 [_1 "." "*" ";"]]
```

Figure 2. A sample of a pretty-print table. The table contains mappings from grammar productions in SDF (on the left-hand side of '—') to corresponding BOX expressions (on the right-hand side of '—').

We use the syntax definition formalism SDF [11] to express language constructs in pretty-print tables. SDF in combination with generalized-LR parser generation [22] offers advanced language technology that handles the full class of context-free grammars. By using this technology in the pretty-printer we also obtain pretty-print support for this class of grammars. In addition to SDF, the general idea of pretty-print tables containing mappings from language constructs to BOX expressions can easily be implemented for other syntax definition formalisms (like BNF) or XML as well.

Figure 2 shows an example of a pretty-print table which defines a format for three language constructs of the programming language Java. The first entry in the table defines a formatting for PackagedDeclaration[1]. This language construct consists of the terminal symbols package and ';', and the non-terminal symbol Name. The formatting rule expresses that these three elements are layout horizontally, that package is formatted as keyword, and that no white space is inserted between the non-terminal Name and the semicolon. Observe the use of the numbered place holder ('_1') to denote the BOX expression corresponding to the formatted non-terminal symbol Name. The remaining formatting entries define similar formattings for the two import declaration constructs of Java.

[1] Please note that productions in SDF are reversed with respect to formalisms like BNF. On the right-hand side of the arrow is the non-terminal symbol that is produced by the symbols on the left-hand side of the arrow.

3.4. A generic box producer

We designed a generic, language independent front-end which applies formatting rules defined in an ordered sequence of pretty-print tables to a parse tree. Separating the language specific formatting rules in tables allows the generic front-end to be re-used unmodified to format any language. Constructing a pretty-printer for a new language only requires language specific formatting rules to be defined in tables.

The front-end operates on a universal format for the representation of parse trees (called AsFix [9]), which preserves layout and comments. Operating on parse trees in general has the advantage that lexical information for disambiguation is available. Therefore we do not have to deal with the insertion of brackets to disambiguate the generated output[2]. Because AsFix is a universal parse tree format, it can represent parse-trees for any language and therefore allows generic parse-tree operations to be defined in language independent tools. As a result, the transformation of a parse tree to BOX can be defined language independently in the single tool `parsetree2box` (see Figure 1). Using AsFix has the additional advantage that all layout is preserved in the tree which simplifies comment handling.

The front-end `parsetree2box` constructs a BOX term for a parse tree of a language by traversing the parse tree in depth first order and simultaneously constructing a BOX term according to the language specific formatting rules in the pretty-print tables. For each node in the tree that corresponds to a production of the language `parsetree2box` searches the tables for the corresponding BOX expression. When a format rule for a production does not exist, `parsetree2box` automatically generates a default rule (this approach makes pretty-print entries optional because simple formattings are constructed dynamically for missing entries). The BOX term thus obtained is then modified to include original comments, and is instantiated with BOX terms representing the formatted non-terminal symbols of the production. Original comments are restored by inserting `C` boxes (containing the textual representation of comments) in the BOX term, and by positioning these comment boxes using the `H` and `V` operators to preserve their original location.

3.5. Pretty-printer generation

Constructing a pretty-printer for a language by hand is a time consuming task. The ability to quickly and easily obtain pretty-printers becomes more and more important when the number of languages and dialects in use increases. For example, development of domain specific languages (DSLs), and language proto-typing requires the use of a large number of pretty-printers and demands enhanced technology for the construction of pretty-printers.

[2] We do not consider constructing valid parse trees (i.e., parse trees containing all lexical information for disambiguation) as part of pretty-printing. In case a tree is not constructed by a parser directly, disambiguation (like described in [30] and [21]) might be needed and has to be performed by third party tools.

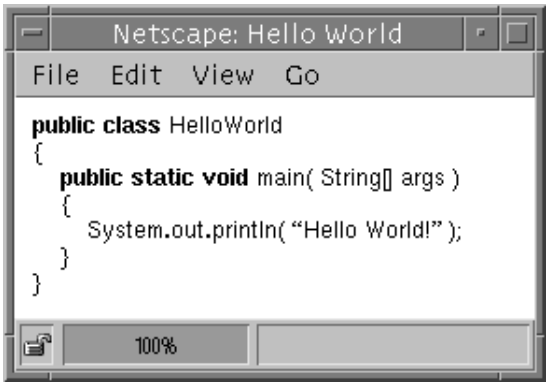


Figure 3. A screen dump showing the result of the HTML code of a Java code fragment as produced by `box2html`.

Pretty-printer generation, based on grammars without annotated format rules, is such technology. This technology supports the generation of a pretty-printer for a language by "guessing" a suitable layout based on grammar analysis and formatting heuristics. Obviously, the result of such generated pretty-printers will not satisfy completely in most cases and the ability to adapt generated pretty-printers strongly increases the usefulness of the generator and its generated formatters.

In addition to the pretty-printer generator described in [30], which produces dedicated, language specific front-ends, we introduce an alternative technique for the generation of pretty-printers which benefits from the table based pretty-print approach. Due to the separation of language specific formatting rules and generic code to perform a formatting, there is no need to generate any code. Only pretty-print tables have to be generated and the generic formatting engine `parsetree2box` can be re-used for each language to perform the actual formatting. This approach completely separates data (the pretty-print tables) and code (the generic formatting engine). The user can customize the formatting by overruling generated formatting rules in tables with higher precedence (see Section 3.3).

In our approach, a pretty-printer generator only consists of a table generator. We developed such a table generator which constructs a separate pretty-print table for each module of a modular SDF grammar. The generator currently only uses simple techniques to generate formatting rules for a language. Improving the generation process by using more advanced heuristics and grammar analysis is a current research topic. Another approach to improve the generated pretty-print tables would be to guide the generation process by means of user profiles (similar to [4]).

```
public class HelloWorld
{
  public static void main( String[] args )
  {
    System.out.println( "Hello World!" );
  }
}
```

Figure 4. The result of formatting a Java code fragment using the back-end `box2latex`.

3.6. Box consumers

A back-end transforms a language independent BOX term to an output format. The advantage of using GPP depends on the number of available output formats. GPP currently supports the output formats plain text, HTML, and LaTeX, which are produced by the back-ends `box2text`, `box2html`, and `box2latex`, respectively. PDF can also be generated but indirectly from generated LaTeX code.

From the three back-ends `box2text` is the most complicated because it has to perform all formatting itself. The translation to HTML and LaTeX is less complicated because the actual formatting is not performed by the back-end but by a WEB browser or LaTeX. The implementation of these back-ends therefore consists of a translation from a BOX term to native HTML or LaTeX code.

The translation to text consists of two phases. During the first phase the BOX term is normalized to contain only horizontal operators, vertical operators, and comments. During the second phase the simplified BOX term is translated to text and the final layout is calculated.

The formatting defined in a BOX term is expressed in HTML as a complex nested sequence of HTML tables. In contrast to BOX, HTML is designed to format a text logically (consisting of a title, a sequence of paragraphs etc.), not as a composition of horizontal and vertical boxes. Only the use of HTML tables (in which individual rows correspond to horizontal boxes and tables to vertical boxes) yielded a correct HTML representation of the formatting defined in a BOX term. Figure 3 shows a screen dump of a pretty-printed Java code fragment produced by `box2html`.

LaTeX code, representing the formatting defined in a BOX term, is obtained by translating the BOX term to corresponding BOX specific LaTeX environments. These environments provide the same formatting primitives as BOX in LaTeX. As an additional feature, `box2latex` allows one to define a translation from BOX strings to native LaTeX code. This feature is used to improve the final output, for instance by introducing mathematical symbols which were not available in the original source text (for example, it allows one to introduce the symbol 'ϕ' in the output where the word `phi` was used in the original source text). Figure 4 shows the result of processing a small Java code fragment by `box2latex`.

3.7. Implementation

For the implementation of the individual tools of GPP we combined modern parsing techniques with compiled algebraic specifications. The parsing techniques, based on SGLR (scannerless generalized-LR) parsing [32], allow us to easily define and adapt grammars and automatically generate parsers from them. The basic functionality of the individual tools is implemented as a number of executable specifications in the algebraic specification formalism ASF+SDF [11, 3, 31]. From these specifications we obtained C code by compiling the specification using the ASF+SDF compiler [28]. The generated C code is efficient and gives a promising performance of GPP despite of its interpreted approach based on pretty-print tables, and its implementation as algebraic specification.

The generated parsers and compiled specifications are glued together into a single component using Unix scripts. We use `make` in combination with dynamically generated Makefiles as performance improvement, to prevent doing redundant work.

In order to process files as produced by `box2latex` by `latex`, the style file `boxenv` is required which contains the implementation of the BOX specific environments. For a general usage of this style file and for an in-depth discussion of its implementation we refer to [7].

4. Case studies

4.1. Formatting real-world languages

We experimented with the pretty-printer and its generator and constructed pretty-printers for some real-world languages. These languages include the programming language Java and the extensible markup language XML. An application of the pretty-printer in industry is its use as formatter for Risla [2], a domain specific language for describing financial products.

For the Java pretty-printer we first constructed a grammar in SDF according to the Java Language Specification [10]. Then we generated pretty-print tables from this grammar. Finally, we customized the pretty-printer manually to meet our requirements. Figure 3 and Figure 4 show the result of formatting a small Java program. Figure 3 is obtained by using `box2html`, for Figure 4 we used `box2latex`.

The XML formatter is another application of GPP for real-world languages. Its development was very similar to the construction of the Java formatter. We first constructed a grammar from XML in SDF according to [6], then we generated and customized pretty-print tables. Thanks to the table based approach, we were able to re-use these tables for the pretty-printer of the language depicted in Figure 5 and 6. Similar to the grammar of this language, which combines the languages XML and BOX (see Section 5), we were able to also construct a corresponding pretty-printer for this language by combining (and re-using) the pretty-printers of XML and BOX.

4.2. Tool construction

The individual components of GPP provide basic language independent pretty-print facilities. These components can easily be used in combination with additional software to construct advanced special-purpose tools. We have combined these generic tools for instance, with language specific features to form two advanced formatting engines for the algebraic specification formalism ASF+SDF [3, 11, 31]. The tool `tolatex` generates a modular LaTeX document from an ASF+SDF specification by formatting each individual module incrementally, and combining them to form a single document with a table of contents and cross references between modules. Similarly, the tool `tohtml` generates hyper-linked HTML documents from a modular specification, featuring visualization of the import structure of the specification and hyper-links between modules.

Other examples of the use of the individual components for tool construction include the integration of GPP in the interactive ASF+SDF Meta-Environment [29], and its integration and distribution as part of XT [8], a distribution of tools for the construction of program transformation systems.

5. Formatting xml documents

The extensible markup language XML [6] is a universal format for the abstract representation of structured documents and data. Pretty-print techniques are used to transform XML documents to user-readable form. Formatting XML documents is being standardized in the extensible style language XSL [1]. The combination of XML and XSL separate content (XML) from format (XSL). Since the intended use of XML initially was limited to WEB documents, techniques for pretty-printing XML documents mostly concentrated on the transformation to HTML.

We expect that the need to represent XML documents in other formats than HTML will grow rapidly. Moreover, alternatives to XSL are desirable because the translation from XML to HTML using XSL is considered to be difficult [17]. Although XSL is powerful, its design might prove to be unnecessarily difficult for the common case and thus makes more simple pretty-print techniques sensible.

Our pretty-printer provides such techniques and combined with its ability to produces different output formats makes it suitable for formatting XML documents.

5.1. Using box to format xml documents

The Document Type Definition (DTD) of an XML document defines the structure of a document. The DTD of an XML document can thus be seen as language definition or grammar, and its contents as a term over that language.

A pretty-printer for a language can be constructed by defining mappings from language productions to BOX expressions. Similarly, a pretty-printer for a particular DTD can be constructed by defining mappings from DTD constructs to BOX. Once such pretty-print tables have been defined, well-formed XML documents over that DTD can be transformed to all output formats for which a back-end is available.

```
<!ELEMENT person (name, surname, age)> —
    V is=3 ["person" _1 _2 _3],
<!ELEMENT name (#PCDATA)> —
    H ["name: " _1],
<!ELEMENT surname (#PCDATA)> —
    H ["surname: " _1],
<!ELEMENT age (#PCDATA)> —
    H ["age: " _1]
```

Figure 5. A simple XML DTD annotated with BOX formatting rules

Example 5.1 In Figure 5 we define a simple DTD which structures personal data (name, surname, and age). The DTD is annotated with BOX formatting rules. These rules formulate that the contents of records should be formatted vertically, left indented, and preceded by the string "person".

Below the textual representation of a typical well-formed document over this DTD is displayed after formatting by `box2text`:

```
person
   name: Johny
   surname: Walker
   age: 5
```

Of course, the formatting can be improved, for instance by using tables to align field names and field values.

Example 5.1 demonstrates that the use of BOX as formatting language in combination with XML, and the use of the available back-ends allows XML documents to be formatted easily.

Currently, we do not support formatting rules to be defined as annotations of a DTD directly (as we did in Figure 5). Instead, we first generate an SDF grammar from a DTD, then we use the SDF grammar to generate a pretty-print table. This indirection allows us to experiment with XML by using existing pretty-print tools, minimizing the need for additional software.

5.2. An alternative style language

The obvious way to transform an XML document to HTML currently is by using XSL stylesheets. An XSL stylesheet specifies how particular documents should be presented in terms of some XML formatting vocabulary. An XSL stylesheet thus describes a structural transformation between the original document and the formatting vocabulary. HTML is used as formatting vocabulary when an XML document has to be transformed into a traditional WEB document.

<!**ELEMENT** *person* (*name*, *surname*, *age*)> —
“<html>” “<head>” “<title>” _1 _2 “</title>”
“</head>” “<body>” _1 _2 “ and ” _3 “</body>”
“</html>”,
<!**ELEMENT** *name* (#PCDATA)> —
“my name is ” _1,
<!**ELEMENT** *surname* (#PCDATA)> —
_1,
<!**ELEMENT** *age* (#PCDATA)> —
“I am ” _1 “years old”

Figure 6. Pretty-print tables used as language to define a simple transformation from XML to HTML.

In spite of its advantage of separating presentation and content, and its expressive power, we agree with [17] that XSL is difficult. First because the language uses the XML syntax which make XSL stylesheets difficult to read. Furthermore, the language is large as a result of the intention to make XSL stylesheets generally applicable. Finally, the combination of a formatting language and a transformation language makes XSL stylesheets complex and difficult to maintain because one has to deal with formatting and transformation issues (by means of tree traversals) simultaneously.

We think that these negative aspects make XSL stylesheets too difficult for many simple transformations. Separation of traversals and presentation, and a less complex language would ease describing simple presentations of XML documents.

With `parsetree2box` simple presentations of XML documents can be defined based on an implicit traversal of the parse-tree. Pretty-print tables are suitable to express a formatting in terms of a formatting vocabulary other than BOX. The combination of an implicit traversal and pretty-print tables as little language to express a transformation to HTML thus forms an alternative to XSL for simple formatting purposes.

Example 5.2 Example 5.1 demonstrated how formatting in terms of horizontal and vertical boxes can be defined for a DTD. Formatting a document according to these rules yields an unstructured representation of the document. Figure 6 shows how pretty-print tables can also be used to define a structured representation in terms of HTML.

The mappings in Figure 6 define for each production of the XML DTD the corresponding HTML code. Formatting a well-formed document using `box2text` according to these rules will yield:

```
<html>
  <head>
    <title>
      my name is Johny Walker
    </title>
  </head>
  <body>
    my name is Johny Walker
    and I'm 5 years old
  </body>
</html>
```

This HTML document can then be displayed by the user using an HTML browser.

Example 5.2 demonstrates how simple transformation rules of XML documents can be separated from code that defines traversals. This provides, in combination with the implicit tree traversals of `parsetree2box`, a simple formatting mechanism of XML documents and may serve as alternative to XSL.

For more complex transformations where implicit traversals are too restricted, we are planning to investigate on using languages designed primarily for transformations as alternative to XSL. An example of such a language is Stratego [33], which has more powerful transformation facilities and a better syntax. We expect that both will help to improve readability and maintainability.

6. Concluding Remarks

6.1. Contributions

In this paper we described the design, implementation, and use of the generic language independent pretty-printer GPP. The system can easily be extended in order to add support for more languages or more output formats. It can also easily be adapted to extend pretty-print support for existing languages. The system combines known techniques (like language independent pretty-printing, division of pretty-printers in front-ends and back-ends, and pretty-printer generation) with new techniques to provide advanced pretty-print support. Our contributions are: i) Formulation of formatting rules in pretty-print tables, which allows for a modular pretty-printer design and supports incremental pretty-printer generation. ii) Customization of pretty-printers by means of ordered pretty-print tables. iii) We developed a generic format engine (`parsetree2box`) which operates on a universal parse tree format and interprets language specific format rules contained in pretty-print tables. iv) We designed a table generator which generates pretty-print tables for a language by inspecting the corresponding grammar. v) We implemented three back-ends which make plain text, HTML, and LaTeX output available for all formatters. vi) The pretty-printer is designed as stand-alone system and can therefore easily be integrated in third-party systems. Moreover, the system is free and can be downloaded from `http://www.cwi.nl/~mdejonge/gpp/`.

Furthermore, we discovered that XML is a relative new application area of pretty-printing. We experimented with XML and we found two useful applications of our pretty-printer. First, the pretty-printer can be used to easily format an XML document depending on its DTD and to translate

it to several different output formats. Second, the pretty-printer can be used for simple term transformations as alternative to XSL. For complex transformations we suggest using more advanced transformation systems (like the programming language Stratego for instance) as alternative to XSL.

6.2. Future work

This pretty-print project was initiated as part of the development of a new ASF+SDF Meta-Environment and its integration as default formatter was the intended goal. The integration of the pretty-printer in this interactive programming environment is not finished yet but is planned to be completed soon.

The table generator is the one component of the pretty-print system that still needs additional research. This research includes experimenting with more advanced heuristics and grammar analysis to guess a suitable layout, and experimenting with user profiles to guide the generation process in order to respect user preferred formatting styles.

The recent experiments with XML proved the usefulness of the generic pretty-print approach that we followed. The rapid growing importance of XML and of formatting XML documents makes it an interesting application area for our pretty-printer and a natural extension of our research.

Acknowledgments The author wants to thank Arie van Deursen for all his suggestions and for the many pleasant discussions we had. The author also thanks Mark van den Brand and Paul Klint for commenting on earlier versions of this paper.

References

[1] S. Adler, A. Berglund, J. Caruso, S. Deach, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible Stylesheet Language (XSL) version 1.0. Technical Report WD-xsl-20000112, World Wide Web Consortium, 2000.

[2] B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[3] J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[4] G. Blaschek and J. Sametinger. User-adaptable Prettyprinting. *Software – Practice and Experience*, 19(7):687 – 702, 1989.

[5] R. J. Boulton. SYN: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical report, Computer laboratory, University of Cambridge, 1996.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.

[7] M. de Jonge. boxenv.sty: A LaTeX style file for formatting BOX expressions. Technical Report SEN-R9911, CWI, 1999. Available from `http://www.cwi.nl/~mdejonge/gpp/`.

[8] M. de Jonge, E. Visser, and J. Visser. XT: Transformation Tools. Available from `http://www.cs.uu.nl/~visser/xt/`.

[9] M. de Jonge, E. Visser, and J. Visser. Definition of the syntax definition formalism Sdf and its parse tree format AsFix. Technical report, CWI, 2000. In preparation.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsysems, 1996.

[11] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[12] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques*, number 925 in LNCS, pages 53 – 96. Springer Verlag, 1995.

[13] M. O. Jokinen. A Language-independent Prettyprinter. *Software—practice and experience*, 19(9):839–856, 1989.

[14] S. P. Jones. A Pretty Printer Library in Haskell, Version 3.0, 1997. Available from `http://www.dcs.gla.ac.uk/~simonpj/pretty.html`.

[15] D. E. Knuth. *TEX and METAFONT : new directions in typesetting*. Digital Press and the American Mathematical Society, 1979.

[16] G. T. Leavens. Prettyprinting Styles for Various Languages. *SIGPLAN Notices*, 19(2):75–79, 1984.

[17] P. M. Marden and E. V. Munson. Today's style sheet standards: The great vision blinded. *IEEE Computer*, 32(11):123–125, 1999.

[18] M. Mikelsons. Prettyprinting in an interactive programming environment. *SIGPLAN Notices*, 16(6):108–116, 1981.

[19] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

[20] D. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[21] N. Ramsey. Unparsing Expressiong With Prefix and Postfix Operators. *Software – Practice and Experience*, 28(12):1327–1356, 1998.

[22] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[23] G. A. Rose and J. Welsh. Formatted Programming Languages. *Software—practice and experience*, 11:651–669, 1981.

[24] L. F. Rubin. Syntax-Directed Pretty Printing – A First Step Towards a Syntax-Directed Editor. *IEEE Transactions on Software Engineering*, SE-9:119–127, 1983.

[25] M. Ruckert. Conservative Pretty-Printing. *SIGPLAN Notices*, 23(2):39–44, 1996.

[26] D. S. Swierstra, P. Azero, and J. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming*, number 1608 in LNCS, pages 150 – 206. Springer Verlag, Braga, Portugal, 1998.

[27] M. G. J. van den Brand and M. de Jonge. Pretty Printing within the ASF+SDF Meta-Environment: a Generic Approach. Technical Report SEN-R9904, CWI, 1999.

[28] M. G. J. van den Brand, P. Klint, and P. Olivier. Compilation and Memory Management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 198–213, 1999.

[29] M. G. J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF Meta-Environment. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer verlag, 1997.

[30] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1 – 41, 1996.

[31] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[32] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[33] E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30 – 44. Springer-Verlag, 1999.

[34] P. Wadler. A Prettier Printer, 1998. Available from `http://cm.bell-labs.com/cm/cs/who/wadler`.

Lua/P — A Repository Language For Flexible Software Engineering Environments

Stephan Herrmann
Technische Universität Berlin, FR 5–6, Franklinstr. 28/29, 10587 Berlin, Germany
phone: +49 30 314 73174 – email: stephan@cs.tu-berlin.de

Abstract

Ongoing development and combination of methods and tools for software development call for software engineering environments (SEE) with ever changing functionality. Also the integration of operative support for the software development process remains a major challenge. A good SEE design has to combine a high level of integration with great flexibility towards evolving methods and tools as well as adaptability towards different kinds of development projects. We have developed PIROL as a generic SEE demonstrating that an **executable meta model** may play a key role in combining integration and flexibility. We coin the notion of a **repository language** to denote a domain specific language for the domain of repository based meta models. We introduce Lua/P as PIROL's repository language with some non-standard properties and show how this language contributes to the desired properties of the environment.

Keywords: Repository, meta-modeling, domain-specific language, tool integration

1. Introduction

Although there is plenty of software tools, many development projects still lack a suitable, state-of-the-art software engineering environment (SEE). The reuse of SEE components, their composability and configurability still falls behind the demands of concrete development projects. In spite of all commonality, projects differ considerably in multiple dimensions. This is for a large part a matter of different workflows and can be modeled in notions of documents, states, persons, resources and schedules. Two factors add to these problems: (a) software development is a matter of very intense communication and (b) a very tight semantical integration of different (sub-)documents should be supported by the environment across tool borders.

It is of course possible to hand craft a specialized SEE for any project just using existing techniques. Only the effort needed is far too high. Thus the uniqueness of software development projects calls for techniques that allow to build a specific SEE from components in just a fraction of the time needed so far.

Many standards and techniques have emerged for solving single concerns of integration. Some focus on techniques for communication between tools (cf. CORBA[1], COM[2] etc.) others tend towards meta formats for data exchange (XML) or even standardize parts of the environment's meta model (XMI). Some approaches are very general techniques that add no special solutions for SEEs; others help under the precondition of a fixed set of notations which is not appropriate for many fields of software engineering where still new formalisms and combinations thereof are widely explored (cf. [3]).

In earlier work [4] we have presented the vision of using an executable object-oriented meta model as the central concept for a tightly integrated yet configurable SEE. Advantages include (a) a semantical enrichment of an otherwise purely syntactic data model, (b) tool independent implementation of process and framework integration (cf. [5]) and (c) adaptability of the meta model through subclassing and scripting. In this paper we elaborate on certain requirements towards this meta model. We show how data modeling can be extended with efficient implementation of very fine grained data structures and elaborate techniques for preserving consistency in the presence of multiple views on shared data. The driving force is always the desire to reconcile a high level of integration with clear modularity allowing flexible configuration of a concrete environment from existing parts.

We point out the importance of the language that is used for meta modeling. Such a language (the meta meta model) will be called a *repository language* throughout this paper. It is a domain specific language for the task of defining meta models as a basis for and as integrative glue of SEEs. The repository language has to blend in with the techniques used for persistence and communication. We will present current work on the repository language Lua/P that is part of the PIROL SEE.

The following section will give an overview of PIROL and the systems it is built upon. Sect. 3 shows how Lua/P encapsulates the underlying repository. Sect. 4 sketches the communicational context. Sect. 5 discusses issues of safeguarding consistency in the repository. Sect. 6 tackles the question of granularity with regard to data modeling and processing. Sect. 7 motivates a disciplined technique of class migration. Sect. 8 wraps up with some examples of applying Lua/P. Just a few implementation details are collected in Sect. 9.

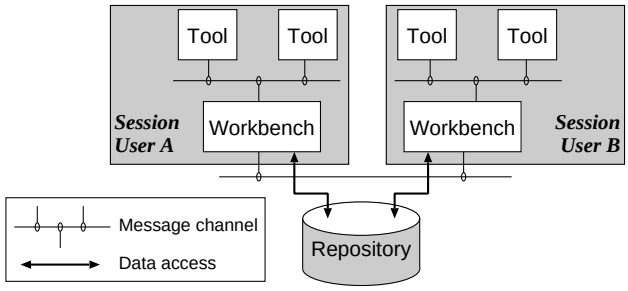


Fig. 1. Architecture of PIROL

2. A Generic SEE: PIROL

PIROL (Project Integrating Reference Object Library)[4] is an SEE designed for supporting all five dimensions of integration as defined in the ECMA reference model[5]. In this section we will give an architectural overview of the system.

PIROL is built according to a three–tier architecture as illustrated in Fig. 1:

1. Data storage is provided by a *repository* which is accessible only through a dedicated server process.
2. For each user a *workbench* defines his or her working context. The workbench caches accesses to the repository and provides private visibility of the user's current work. Communication between the workbench and the repository is based on the mechanisms provided by the repository. Also, messages may be sent to other workbenches.
3. *Tools* are connected to a user's workbench by means of a special messaging protocol. Tools only communicate with the workbench, never directly with the repository. Support is given for the implementation of new tools and also for the integration of existing ones.

The focus of this paper lies on the workbench and on Lua/P, PIROL's repository language which is implemented by an interpreter as part of the workbench. However, before presenting Lua/P we will briefly introduce two systems on which all this is based.

2.1. H–PCTE

PIROL uses a system called H–PCTE[6] as its repository (cf. Fig. 1). H–PCTE is an implementation of the ISO standard PCTE[7]. Its central component is the *object management system* (OMS), that defines a non–standard database system. PCTE closely adheres to extended entity relationship models. Thus the main elements are entities (objects), relationships (links) and attributes. The latter can be attached to either objects or links. The database is non–standard in that the primary access is not via tables and key attributes but by navigation along links.

H–PCTE is implemented (and continuously being improved and maintained) by the group "Praktische Informatik", at Siegen University[8]. A special focus of the design of H–PCTE was the performance issue in conjunction with fine grained data modeling. H–PCTE spells "high performance PCTE".

2.2. Lua

Lua[9] is developed at PUC-Rio, Brazil. It is titled an "extensible extension language". The notion extension language is to say that Lua is well-suited for integration into a given application in order to provide a high level facility for configurability, macro programming and the like.

Within our context the extensibility of Lua itself is of greater importance. Lua is adapted very easily to specific needs yielding a great variety of derivatives that could be called a *family of languages*. Lua is a (basically) interpreted language, which comes as a library that can be linked to applications. It is extensible by two mechanisms:

- C functions may be registered as Lua functions.
- The behavior of the interpreter can be modified by the Lua code itself. This is done by redefining standard constructs like reading or writing a field of an object.

At the time we started using Lua for PIROL this second mechanism was called "fallbacks" and operated globally for all kinds of objects. The PIROL development required to use fallbacks for very different purposes (depending on the kind of object). This contributed to an improvement of the mechanism, which is now called *tag methods* and can differentiate kinds of objects by a tag that is attached to each object.

By means of these extension mechanisms, we extended the host language Lua to encapsulate H–PCTE and add object orientation to Lua, as well as all those features that qualify it as a specialized repository language. Lua/P spells "Lua for PIROL".

3. Encapsulating the Repository

Lua/P encapsulates the repository in a way that combines the following properties:

1. Lua/P objects are persistent without the need of explicit read and write operations. Reading is lazily performed when dereferencing a link, writing occurs immediately upon every attribute assignment or object creation.
2. Lua/P is used to define all data types in the repository and adds behavior (methods) to repository classes.
3. Many functions from the large PCTE-API are encapsulated by core classes of the meta model written in Lua/P. There is no need to access PCTE directly.
4. Lua/P itself is an evolving language that gives specialized support for the most common concerns in SEEs. The following sections will focus on the concerns of consistency (Sect. 5) and fine grained meta modeling (Sect. 6).

Properties (1) and (2) mainly define Lua/P + H–PCTE as an OODBMS. Note however, that Lua/P is not meant to compete with standard OODBMS, but it combines

features of an object oriented database language with specialized features for programming SEEs.

We had to consider some subtleties in the type system of Lua/P as compared to PCTE: The type system of PCTE knows types of attributes (string, integer, boolean), objects and links, which are straight forwardly mapped to basic types, classes and object references in Lua/P. Lua/P, however, (a) encapsulates 1:n links by a List class and (b) adds lists of basic values and lists of tuples. These additions of course need to pay attention to efficiency, so some of these types are packed into one binary attribute in PCTE, or make use of link attributes, which otherwise have no representation in object oriented languages. Only tuples with more than one object component cannot benefit from such optimizations. They have to be represented by additional PCTE objects. Sect. 6 shows how structured data types can be added to this type system with minimal runtime overhead.

In addition to persistent objects, Lua/P also allows to declare classes or attributes as *transient*, which is important where the creation of temporary objects as repository objects would impose an unnecessary performance penalty. Clients, however, need not know about this distinction because persistent and transient objects are handled in a uniform way.

4. How Tools Access the Workbench

It has been shown how Lua/P encapsulates the underlying OMS. This section briefly shows the interface through which tools can operate on repository objects. This defines the context from which execution of Lua/P code is triggered and motivates why the Lua/P interpreter must support change propagation between tools.

The channel used for all communication between workbench and tools is a messaging facility with a multicast protocol. This facility is based on the package MSG from the FIELD environment[10] with significant modifications, that are not discussed here. Messaging is implemented by a server and a client library such that all clients can easily establish a socket connection to the server. The server is responsible for message dispatching. Workbench and tools are clients of the message server, which is in general transparent to both as they communicate with each other.

In PIROL six types of messages exist by which tools may request services from the workbench:

query: Read the value of one attribute (simple or complex).

query list: Query a detail about a list attribute. Four sub-functions exist:
`length()`, `item(index)`, `search(value)`, `filter(constraints)`

set: Assign a simple attribute value.

set list: Lists can only be modified by these sub-functions:
`append(value)`, `replace(index,value)`, `insert(index,value)`, `delete(index)`.

execute: Execute a Lua/P method, passing any number of arguments.

create: Create a new repository object. This may include a call to a Lua/P creation method.

Additionally the workbench broadcasts all attribute changes to the message channel. This is a very important feature of the Lua/P interpreter, because PIROL is designed for supporting multiple views, which obviously need to be kept consistent by means of some mechanism of change propagation. It is left to each tool to register a pattern for each object or attribute which is displayed by the tool. The tool then receives all relevant update messages and may update its display accordingly. List updates are sent as incremental changes (reflecting all those `append, replace ...` operations). For simple attributes the new value is passed with the update messages.

In PIROL tools can generally be implemented in any programming language. Currently most tools are implemented in Java.

A class library exists for *Java* that encapsulates the messaging library and provides proxy classes for all classes of the meta model core. Proxy classes typically have `get_`*xx* and `set_`*xx* methods for each attribute. Lists are encapsulated by a Java class, which automatically observes all changes of the corresponding list in the repository. It may register additional observers that propagate changes within the tool. Proxy classes have static methods wrapping the creation of a repository object. Method calls are directly delegated to the workbench. The main limitation of the Java client library is the lack of multiple inheritance in Java. So a *complete* mapping from Lua/P to Java is not possible.

A client library using *Lua* may exactly imitate the behavior of the workbench. No `get_`*xx* or `set_`*xx* methods are needed. Not even proxy classes are needed for this library, since they can be build on the fly from meta information available from the workbench. Currently however no tool is using this technique.

5. Consistency

PIROL emphasizes the role of multiple views within a repository based environment. This imposes obligations to safeguard the consistency of all views and their representations as objects in the repository. Consistency has to be ensured at least on two levels: when regarding tools as view–control components according to a model–view–control architecture, different views need to be kept consistent by means of change propagation as mentioned in the previous section (cf. also [11]). A more semantical understanding of consistency concerns invariants and semantical constraints describing interdependencies between different objects/attributes in the repository. In this section we will focus on the latter aspect of consistency, but we will also relate this aspect to techniques

Event	Arguments	Description
Simple attributes:		
assign	`object, value`	assignment of `value` to the resp. attribute of `object`.
get	`object`	retrieve the resp. attribute from `object`.
List attributes:		
adding	`list, index, value`	`value` is being added to `list` at position `index`.
removing	`list, index, value`	`value` is being removed from `list` at position `index`.
append	} all regular list functions	
remove		
...		

Fig. 2. Events for attribute guards

of change propagation.

5.1. Guarded Attributes

Lua/P has been presented as a language for data definition and manipulation. It is generally possible to set any attribute of any object to any value, as long as access permissions and type correctness are observed. This is very comfortable for most cases, but sometimes this weak encapsulation is not sufficient. For achieving a better encapsulation, Lua/P provides the mechanism of *guarded attributes* which allow to implement further constraints. Different applications of this technique are possible:

1. Consistency constraints of the meta model might require a change of one attribute to be propagated in terms of changing also some other attributes/ objects.
2. Other constraints might allow only specific changes, in which case assigning a wrong value should either throw an error or just do nothing.
3. Some attributes may represent something outside the Lua/P interpreter. Changing their value should produce a side–effect by calling some low-level interface.

Examples for the three kinds of constraints are:

1. Classes `CLASS` and `ROUTINE` from the PIROL meta model both have a flag `is_deferred`[1]. The constraint is: a class that has at least one deferred routine is itself deferred. Both flags are implemented as guarded attributes. When a routine is set to deferred, the corresponding class is also marked deferred, when a class is set to not deferred, it is ensured that also all routines are not deferred.
2. In class `WORKBENCH` an attribute `current_group` defines, on behalf of which group the current user is working. He or she may adopt another group simply by assigning that group to `current_group`. Given only this, every user could possibly gain the rights of *any group* by a simple assignment. However, implementing `current_group` as guarded attribute allows additional checking. Only groups, of which the user is a member are allowed. Violating this restriction generates a warning and the assignment is refused.
3. In fact the previous example already hints at a third application of guarded attributes. The attribute `current_group` is not only documentation but effectively defines the permissions of the current user. The effective access rights are always the combination of the user's personal rights and those of his or her current group. This is achieved by calling the PCTE function `pcte_adopt_group()` each time the `current_group` has successfully been assigned.

[1] Deferred stands for 'not fully implemented', 'abstract'. This is one of a set of notions which we borrowed from the nomenclature of Eiffel[12].

Definition of Guarded Attributes. Guarded attributes are declared and used just like others. Only a set of guard functions is added that is used for certain events. The syntax of a full guard definition is:

```
AttributeAccess Class.Attribute {
  Event = function(Arguments)...end,
  ...
}
```

This is for simple attributes. For list attributes the keyword `ListAccess` is used instead. Fig. 2 shows the events that can be (re)defined, Fig. 3 on the next page shows an example. Here only the *assign* event is overridden by a function that ensures the consistency of the flag `is_deferred` from our example (1) given above. `raw_assign` is the actual assignment function to be used within an *assign* function.

It should be mentioned that guarded attributes are mainly a matter of convenience. We could have required that all attributes be accessed only by means of explicit access functions (even within the same class!), which would, however, unnecessarily clutter the code and should actually only be used, where needed. That's why in Lua/P no reason exists to ever call a function like `get_attribute` or `set_attribute`. On the client side it is important that regular and guarded attributes are used in the same way. This guarantees that common tools like generic browsers are able to exploit guarded attributes in a meaningful way, without knowing about their guards.

5.2. Derived Attributes

The previous section showed how to ensure consistency in case of interdependencies between different objects/attributes. It is certainly a good idea to minimize the necessity of such consistency constraints by avoiding redundant data in the repository wherever possible.

```
Class ROUTINE {
    attributes={
        is_deferred : Boolean,                          -- Declaration
        ...
    },...
}
...
AttributeAccess ROUTINE.is_deferred {
    assign =                                                    -- Event
        function (routine, flag)                    -- Guard function
            if flag == routine.is_deferred then
                return
            end
            raw_assign(routine, "is_deferred", flag)
            if flag then
                local class = routine:get_class()
                class.is_deferred = flag
            end
        end
}
...
local routine = ...
routine.is_deferred = true      -- Application triggering the guard
```

Fig. 3. Guard for attribute `ROUTINE.is_deferred`

```
Class ROUTINE {
    inherit FEATURE,
    attributes={
        signature : Derived(String),                    -- Declaration
        arguments : List(ENTITY),
        ...
    }
}
...
function ROUTINE:derive_signature ()  -- Derivation Function
    local args = self.arguments:foldl ( "(",
        function (arg, pre)
            if not pre == "(" then pre = pre..", " end
            return pre..arg.signature       -- read another Derived
        end)                                                Attribute
    args = args..")"
    if self.type then
        return self.name..args..": "..self.type.name
    else
        return self.name..args
    end
end
```

Fig. 4. Derived attribute `ROUTINE.signature`

PIROLs fine grained meta model is a major step towards absence of redundancy. Another means is the mechanism of derived attributes.

As an example of derived attributes, consider the signatures of routines. Names, arguments and result types of routines are kept persistently either as direct attribute (name) or using separate repository objects of types `ENTITY` (arguments) and `TYPE` (result type). It should on the other hand still be possible to query the signature of a routine (encoded as a human readable string) with just one query. Pre-assuming a redundant *attribute* `signature` should be avoided for the sake of consistency, a *function* `ROUTINE:get_signature()` would be a good starting point, but it has a great disadvantage as compared to attributes. Only attributes allow to register observers that inform a client about all changes of the corresponding value. Function results may become invalid without further notice.

Derived attributes now combine the best of both worlds. They are free of redundancy because they are not persistent but computed when needed. Syntactically a derived attribute is an attribute and most importantly the workbench broadcasts all changes of the values of derived attributes for which a tool holds an observer. Fig. 4 shows how a derived attribute is declared using the type constructor `Derived` and how a derivation function is attached as `derive_xx()`.[2]

6. Fine Grained Meta Models

Fine grained data modeling is a powerful means for a tight integration of tools that are to share as much information as possible. Of course an object oriented meta model could very well be used to decompose a document all the way down to single identifiers and symbols. This technique is however not usable for SEEs. A prominent approach to fine grained data modeling for SEEs has been standardized as extension of PCTE[13]. Unfortunately no tool vendor ever really implemented this standard due to tremendous performance problems that should be expected. Database technology in fact imposes quite strict limits on the number of objects that can be accessed efficiently when eg. loading a document.

Quite a different lesson can be learned from the area of compiler construction and related tools. Such tools rely on a set of types that represent all constructs of a (programming) language in a tree or DAG structure, called *abstract syntax*. The definition of these types and many transformations are much more compact and perhaps more elegant when using a functional programming language rather than an object oriented one. For this reason a previous instantiation of PIROL [3] that was targeted at processing formal specifications based on their abstract syntax used the programming language Pizza[14]. We made good experiences with Pizza's combination of object oriented and functional techniques. In this setting the bottleneck was the serialization of Pizza objects. Serialization, which was used to write units of the abstract syntax as binary blocks into the repository, again imposed performance problems on the system.

In response to this experience, Lua/P was extended by some new features: The types needed for an abstract syntax or similar structures can be defined as *term grammars*. Terms as values of these types can be handled and stored efficiently by the workbench. Allowing term types for attribute declarations yields a smooth integration of medium grained objects and very fine grained terms. Finally a touch of functional programming in Lua/P allows concise implementations of algorithms over terms.

6.1. Term Grammars

Terms are tree structures whose leaves are simple values or terminal symbols. Simple values are strings, integers, boolean or subtypes thereof. Lua/P provides four

[2] ".." is the Lua operator for string concatenation. For the function `foldl()` cf. Sect. 6.2.

```
   Grammar {"EXPRESSION";
(1)    expr    = one_of{value, binexp, unexp, ifexp},
(2)    binexp  = {{e1=expr}, {operator=binop}, {e2=expr}},
(3)    unexp   = {{operator=unop}, expr},
(4)    ifexp   = {{condition=expr}, {then_exp=expr},
                                    {else_exp=expr}, '?'},
(5)    binop   = subtype_of{STRING},
(6)    unop    = one_of_const{{uplus='+'}, {umin='-'}},
(7)    value   = one_of{INT, BOOL, varappl},
(8)    varappl = subtype_of{STRING},
(9)    vallist = {value, '*'},
  }
```

Fig. 5. Grammar `EXPRESSION`.

kinds of type rules (the LHS of each rule is a type):

subtype_of	The LHS type can be used wherever the RHS type is required. It has the same structure.
one_of	The LHS type has alternatives that are listed here. The alternatives still have to be defined.
one_of_const	Similar to the above but the alternatives are terminal symbols given by their representation.
production	The LHS type is a tuple of the types listed at the RHS. Production rules have no keyword.

Fig. 5 gives an example grammar defining a simple expression language. The names `e1, operator` and `e2` as defined in rule (2) are selectors for the components of an `expr` term. The second component in rule (3) is not named, so the type name `expr` is also used as selector. The `'?'` in rule (4) specifies that the last component (`else_exp`) is optional. The `value` in rule (9) may occur zero to many times (denoted by `'*'`). Elements of such a list can only be accessed by their numerical index. Finally the whole grammar is given a name in order to make it a selectable name space.
Each type defined in a grammar can be used for attribute declarations as in

```
Class SIMPLE_FUNCTION {
  inherit ROUTINE;
  attributes={
    value : EXPRESSION.expr
  }
}
```

When storing such values to the repository a compact yet type safe binary encoding is used to pack complex terms into a single attribute of PCTE. The effect is that for Lua/P each (partial) term is a well defined entity and terms are structures built according to strict type rules. The system's performance however does not suffer from such very fine grained data modeling, because no additional PCTE objects are created.

6.2. A Touch of Functional Programming

Being able to define term types by grammars as shown above we left pure object oriented techniques. The new types can only be exploited effectively if we also add the appropriate functions for *handling* terms. Fortunately, Lua already provides the basic mechanism for functional programming. In Lua a function is a value that can be assigned to variables and can be passed as function argument or result. Additionally *function closures* are supported which by so-called upvalues provide a clean solution to the problem of variable scoping as it occurs in nested functions.[15]

```
function expr2string (t)
   return t_switch(t,
(1)    t_case('@value',
         function (val)
           return val
         end),
(2)    t_case({'expr', '@binop', 'expr'},
         function (e1, op, e2)
           return('('..expr2string(e1)..op..expr2string(e2)..')')
         end),
(3)    t_case('unexp', {'@unop', 'expr'},
         function (op, expr)
           return('('..op..expr2string(expr)..')')
         end),
(4)    t_case('vallist',
         function (list)
           return '{'..
                  list:foldl('',
                             function (v, col)
                               if col ~= '' then col=col..', ' end
                               return col..v.repr
                             end)..
                  '}'
         end),
       t_otherwise(
         function () return '?' end)
   )
end
```

Fig. 6. Using pattern matching for a simple pretty printer.

Next to pure higher orderedness a special merit of many functional languages is their support for *pattern matching*. In Lua/P this is done by a function `t_switch` which matches a given term against a list of type patterns. Patterns are given by `t_case` branches. In the simple case, each pattern specifies a type and a function that should be executed, if the term is conform to that type. The function is called with the term as only argument. In addition to the top-level type a pattern may also give a list of types to which the subterms must conform. If such a pattern is matched, the subterms are passed as distinct arguments to the function. When the string *representation* of a term is desired this conversion can be automated by prepending the `@` operator to the type pattern. Finally a `t_otherwise` branch may provide a default function, that is used if no pattern is matched.

See Fig. 6 for a simple pretty-printing function for expressions as defined in Fig. 5. Note, that in object-oriented programming the standard technique for this problem would be to apply the visitor pattern, introducing far more overhead than the more functional approach.

The first branch matches subtypes of `value`, the next branch matches any term consisting of an expression,

a binary operator and another expression. Branch (3) combines matching of top-level type (`unexp`) and structure (unary operator and expression). All operators and values are passed by their representation (use of `@`). Expressions are passed as terms. Branch (4) again is a simple match by type. It shows an application of the `foldl` function, which is borrowed from ML[16]. We introduced `foldl` to Lua/P as one of the most general higher order functions, that iterates over a list, collecting the results through a second argument (`col`). In this example the effect resembles a smarter mapconcat function: the representation of all list elements are concatenated using `', '` as a separator except for the first element.

7. Upgrading

In most object oriented languages the type of an object is an unchangeable property. In PIROL it has reasons of methodology that it should be possible to create an object with an unspecific type which later-on, as more information is gathered, is converted according to a more precise type. This may even iterate along several steps, such that the object's type stepwise becomes more and more specific. Because references to such an object must remain valid, it is important that

- the object identity is not changed and
- the object remains conform to the typing of all incoming references.

It is the first demand, that leads to *upgrading* as a special language feature rather than a mere copy method. The second demand restricts upgrading to a type change from a superclass to one of its subclasses. (For type conversions, or "migration", cf. eg. [17]).

Regarding the consistency of attribute values, upgrading is very similar to object creation. At creation time a creation method puts the object into a sound state with respect to its attribute values. When upgrading an object new attributes may be added, which are initialized by an upgrade method. Lua/P also introduces a hook called `upgrade_pre` which may be used to decide if upgrading should be allowed or disallowed. This way a class may put forward constraints on the state of objects that shall be upgraded.

At the user interface level upgrading is available through an operation "`insert as ...`". A selected unspecific object may eg. be inserted into a class diagram *as class* or *as attribute*.

8. Application of Lua/P

This section gives examples for parts of an environment that may be implemented in Lua/P and benefit from execution within the workbench.

8.1. Process Modeling

Many projects have taken efforts on formalizing the dynamics and constraints of software development processes. Using Lua/P for this task is elegant because all relevant entities are equally at hand and can be handled in a uniform manner. A process model in Lua/P may at the same time refer to a `SUBSYSTEM` that is being developed including all contained `CLASS`es as well as the `PERSON` who is responsibly for delivering these items in a certain `STATE` using certain (time–, hardware– ...) `RESOURCE`s etc.

As a simple example we present a state machine that manages the progress of development items along a chain of document states like *busy*, *proposed* (for publishing), *published* etc. Each artifact carries a reference to an object of class `STATE`. `STATE`s are connected to *next states* by `TRANSITION`s. Transitions in turn have a `GUARD` and an `ACTION`. In the most simple implementation `GUARD` objects specify which persons/groups are allowed to perform the corresponding transition and an `ACTION` object specifies how the access permissions of an object shall be changed.

This state machine can be adapted on two levels. First, the concrete set of states and transitions is a graph of objects that can be initialized by a Lua/P script once for each project. Second, the algorithms of guards and actions can easily be redefined by subclasses of `GUARD` and `ACTION`.

8.2. Common Services

Class `WORKBENCH` is central for the PIROL meta model. For each user an object of this class defines the working context that is available to all tools. This object contains eg.

- a list of tools that are installed and configured,
- a reference to the current project, group (cf. Sect. 5.1) etc.
- a simple clipboard.

Based on this information, services are implemented like selecting all tools that are available for editing a given object, providing lists of recipients to whom messages can be sent (within the group, project, company ...). Other services that are implemented in Lua/P are version control and access control.

Uniform Context Menus. All these services are really helpful only if they are easily performed at the user interface. An example of a flexible integration of services into the interface of all tools is the mechanism of *workbench provided context menus*. In addition to any local context menu as it may be needed for operating a tool, a submenu `Workbench ▷` is always provided. Concerning the selected object, this menu is dynamically put together by the workbench. The menu definition is kept as a set of transient objects within the workbench. The tool only reads the menu definition and displays the menu. Upon selection of one menu option the corresponding command (function closure) is executed within the workbench. This way no tool needs to know about any common service in particular, but all services provided by the workbench can be exported easily to all

tools. Inserting a new service (like the state machine shown above) or changing the configuration of a service (like adding a new transition to the state machine) has immediate effect on the context menu within all tools.

9. Implementation Issues

So far we have presented the current state of Lua/P. We did not develop this language from scratch nor in one single step. Lua/P evolved from Lua in several incremental cycles. We would like to give some hints on how Lua made this development fairly easy.

Lua [9], [15] is in fact an interpreter *framework* in the sense, that it is an application with many *hot spots*, through which a programmer may add and modify the application's behavior. In Lua these hot spots are those tag methods that have been introduced in Sect. 2.2. By means of tag methods, a simple assignment like `obj1.att2=obj3.att4` can be redirected to invoke one tag method for retrieving the value of attribute `att4` from a repository object `obj3` and another tag method for storing this value in attribute `att2` of another repository object `obj1`. Both tag methods make use of calls to functions from the PCTE API that for this purpose have been lifted from C to Lua. As seen from the Lua interpreter, repository objects are opaque handles, whose semantics are defined solely by tag methods. A similar technique is used for term values (cf. Sect. 6) which are efficiently implemented in C (which allows easy interfacing also with other programming languages).

All access functions, that are stored with individual Lua/P classes, are elegantly stored in function tables using Lua *closures*, that contain all necessary context information besides the function proper.

Another example for tag methods is the invocation of creation methods: the Lua/P statement `obj=CLASS1:make(arg)` denotes the creation of an object of `CLASS1` using the creation method `make` for initialization. This is implemented in Lua as follows: retrieving field `make` from `CLASS1` yields a method-*descriptor*. When trying to execute this descriptor as a function, a tag method (for event `function`) is called, which will find out, that no target object exists but one must be created prior to invoking the actual method `make`.

The use of associative arrays (cf. [9]) allows to write down all additions to the language in a usable, descriptive syntax without ever touching the implementation of the Lua parser. Only tiny changes have been made to the code examples in this paper, in order to give it a more standard appearance. Such modifications can easily be handled by a simple pre–processor.

Note, that aside from Lua/P no other language is required: no data definition language, because this is just one role of Lua/P, and no scripting language for whichever purpose, because Lua/P can also be used for scripts that automate any repeated tasks.

10. Conclusion

Several languages and systems exist, that are partially related to the work presented in this paper. PCTE's DDL is a pure data definition languages. ODMG-ODL[18] and CORBA-IDL[1] define object interfaces in terms of attributes and methods. As early as 1987, Garlan [19] defines different languages for defining different kinds of views. His *basic views* define the structure and behavior of objects in the database, *dynamic views* can be compared to derived attributes, with the restriction, that their results must be sets of objects. Pizza[14] is a programming language that combines object oriented and functional techniques. Within the UniForM workbench[20] concurrent Haskell is used for encapsulating repository objects and tools and implementing tool communication. Derived attributes are supported by notations ranging from Object–Z up to UML. OPM[21] is a specialized DBMS that implements derived attributes.

None of these systems defines a repository language as comprehensive as Lua/P. We should clarify that PIROL is not a production environment competing with industrial tools. Until now, some well known concerns like fine-tuned transaction management remain incomplete in PIROL, but by means of PIROL and Lua/P we are able to demonstrate, which abstractions and mechanisms may be covered by a repository language in order to enhance modularity and integration of an SEE.

We have shown that Lua/P is well-suited as a data definition language and at the same time lifts the repository types to a comfortable object oriented programming language. Fine-tuned type mappings and the introduction of transient attributes and objects provide for optimizations as they minimize the number of repository objects. Using term grammars for the definition of very fine grained data further improves the performance of the system, because terms can be packed into a single attribute.

Next to efficiency, Lua/P provides two mechanisms for preserving data consistency. Derived attributes help to avoid redundancy. Guarded attributes may either restrict attribute changes or operationally enforce consistency by propagating changes to other attributes/objects. Additionally, guarded attributes may lift properties of the underlying repository to Lua/P.

Aside from data modeling, Lua/P can be used for implementing the dynamics of repository objects using methods in the common object oriented sense. Addition of functional techniques enhances Lua/P's capability for transformations over complex structures like abstract syntax.

Finally, Lua/P closes the gap between tools and the repository. All persistent objects are accessible via a specialized messaging facility, which helps to keep visualizations up-to-date by broadcasting all relevant changes.

The process model and all common services that are implemented in Lua/P are independent from (but still

contribute to) any integrated tool. Modularity of the resulting SEE allows for configuration and adaptation at different levels: (1) Selection of methods and notations has impact on the product related part of the meta model and on the selection of tools. (2) Also adaptation of the process model or common services requires a few specific additions to the meta model. (3) Tailoring an environment that has been constructed by the above steps for a concrete project is mainly a task of writing simple Lua/P scripts.

Of course the crucial part in putting together an SEE from diverse components remains implementation, adaptation and integration of new and existing tools. We have made good experiences with tools providing different levels of openness. This ranges from tools specifically written for PIROL up-to monolithic tools with only insufficient interfaces[3]. We have good results especially with a graphical editor that could be adapted at the source code level. A clearly delimited adapter layer suffices for a very close integration.

Most importantly an object oriented meta model written in Lua/P greatly fosters the separation of concerns of different tools and the workbench. Thus flexibility of the SEE PIROL is essentially founded on the specific design of its repository language Lua/P.

10.1. Current and Future Work

We are currently working on specialized language constructs for defining connectors that help to integrate existing tools by automating a mapping between two mismatching meta models [22]. This approach also overcomes a major drawback of *basic views* according to [19]: in some cases it is necessary to multiply instantiate a certain view with regard to a common base structure.

Allowing wildcards in attribute names for guards (cf. Sect. 5.1) will allow to use guards as advice in the style of AOP[23], [24].

It will be easy to lift PCTE's distinction between association (usually: existence links) and object composition (composition links) to Lua/P. This powerful feature is unknown to common object oriented programming language.

References

[1] "The Common Object Request Broker: Architecture and Specification, revision 2.1," TC Document formal/97.9.1, OMG, 1997.

[2] S. Willliams and C. Kindel, "The component object model: A technical overview," Tech. Rep., Microsoft Corporation, available from `http://www.microsoft.com`, 1994.

[3] R. Buessow, W. Grieskamp, W. Heicking, and S. Herrmann, "An open environment for the integration of heterogeneous modelling techniques and tools," in *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*. October 1998, number 1641 in LNCS, Springer.

[4] B. Groth, S. Herrmann, S. Jähnichen, and W. Koch, "Project Integrating Reference Object Library (PIROL): An object–oriented multiple–view SEE," in *Proc. of SEE'95*, Noordwijkerhout, Holland, April 1995, ACM–Press, Malcolm S. Verrall.

[5] "Reference Model for Frameworks of Software Engineering Environments – ECMA TR/55 3rd edition," Tech. Rep., European Computer Manufacturers Association (ECMA), June 1993.

[6] Udo Kelter, "H–PCTE — a high–performance object management system for system development environments," in *Proc. COMPSAC '92*, Chicago, Illinois, Sept. 1992, pp. 45–50.

[7] "ISO/IEC 13719-1: Portable Common Tool Environment (PCTE)," Abstract specification, International Organization for Standardization (ISO), 1995.

[8] U. Kelter, "Einführung in H-PCTE," Skriptum, Fachgruppe Praktische Informatik, FB Elektrotechnik und Informatik, Uni Siegen, 1998.

[9] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "Lua—an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[10] Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software*, vol. 7, no. 4, pp. 57–66, July 1990.

[11] D. Platz and U. Kelter, "Konsistenzerhaltung von Fensterinhalten in Software–Entwicklungsumgebungen," *Informatik Forschung und Entwicklung*, vol. 12, no. 4, pp. 196–205, 1997.

[12] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall International, New York, 1992.

[13] "Amendment 1 to ISO/IEC 13719-1: Fine-grain object extensions," Tech. Rep., International Organization for Standardization (ISO), 1995.

[14] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice," in *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[15] L. H. de Figueiredo R. Ierusalimschy and W. Celes, *Reference manual of the programming language Lua 3.2*, `http://www.tecgraf.puc-rio.br/lua/manual`.

[16] L. C. Paulson, *ML for the working programmer*, Cambridge University Press, 2nd edition edition, 1996.

[17] J. Su, "Dynamic constraints and object migration," in *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, Eds. 1991, pp. 233–242, Morgan Kaufmann.

[18] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, Eds., *The Object Data Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[19] David Garlan, *Views for Tools in Integrated Environments*, Ph.D. thesis, Carnegie Mellon University, May 1987.

[20] C. Lüth, E. W. Karlsen, Kolyang, S. Westmeier, and B. Wolff, "HOL-Z in the UniForM-workench – a case study in tool integration for Z," in *Proceedings of the 11th International Meeting of Z Users (ZUM'98)*, J. P. Bowen, A. Fett, and M. G. Hinchey, Eds., Berlin, 1998, number 1493 in LNCS, pp. 116–134, Springer.

[21] I.A. Chen, A.S. Kosky, V.M. Markowitz, and E. Szeto, "Constructing and maintaining scientific database views," in *Proc. of the 9th Conference on Scientific and Statistical Database Management*, August 1997.

[22] S. Herrmann and M. Mezini, "Dynamic view connectors for separating concerns in software engineering environments," in *Procs. of MDSOC workshop at ICSE 2000*.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin, "Aspect Oriented Programming," in *Proceedings of ECOOP '97*, 1997, number 1241 in LNCS, pp. 220–243.

[24] *AspectJ Language Specification*, available from `http://aspectj.org`, Aug 1999.

Applying Workflow Technology to the Construction of Software Engineering Tools

Anthony Barnes
School of Computer and Information Science
University of South Australia
Mawson Lakes, SA 5095, Australia
email: 9705532s@camtech.net.au

Jonathan Gray
School of Information Technology and Computer Science,
University of Wollongong
NSW 2522, Australia
Tel +61(0)2 4221 3157
email: jpgray@computer.org

ABSTRACT
This paper describes work on a project concerned with practical approaches to software engineering tool construction. Specifically, this paper reports how workflow technology, and an emerging standard for workflow products, can be used to provide a low cost constructional approach to software engineering tool developers. Using an example Software Engineering Environment (PSEE), built on top of a commercial RDBMS, we demonstrate a mapping from the PSEE onto the WfMC Process Definition Interchange Process Model. The resultant workflow process definition can be imported into a WfMC conformant workflow management system, thereby enabling the enactment of the software process model by different workflow management systems.

Keywords
Workflow, process management, tool construction.

1 INTRODUCTION

A *software engineering tool* is a software product that provides some automated support for the software engineering process [1]. This includes: support for development activities such as specification, design, implementation, testing, and maintenance; support for process modeling and management; and metaCASE technology, used for the generation of custom tools to support particular activities or processes. There are many different kinds of software engineering tool variously known as CASE (Computer Aided Software Engineering), CAME (Computer Aided Method Engineering), IPSE (Integrated Project Support Environment), SEE (Software Engineering Environment), and CSCW (Computer Supported Cooperative Work). The development of these tools is in itself a significant and challenging software engineering task. Although these tools differ in purpose and scale, developers of these tools often face similar constructional issues, such as: selection of host computing platform and implementation language, conformance with standards and reference models, choice of repository, integration and interoperability mechanisms, and user interface style. The focus of our work lies not in the application of these different kinds of tool, but in the *engineering of the tools themselves*.

The provision of appropriate software engineering tools plays an important role in the promotion and adoption of sound software engineering practices. One of these sound practices is the identification of, and adherence to, suitable software development processes. For all organisations involved in software development and for all kinds of software, the identification and implementation of a suitable development process is a recognised approach to improving the quality of the software produced. The use of an appropriate software process management tool can assist the implementation, maintenance, and improvement of an organisation's software development process. There are many standard development methods and processes such as JSD, SSADM, OMT, UML/Unified Process, and there are plenty of commercially available tools to support them. However, many organisations do not use these standard methods and would prefer to use their own particular process or a modified version of the standard processes. These organisations could benefit from the provision of flexible and customised process management tools tailored to the specific requirements of their particular development processes.

Workflow technology is defined as: "The automation of a business process, in whole or part, during which

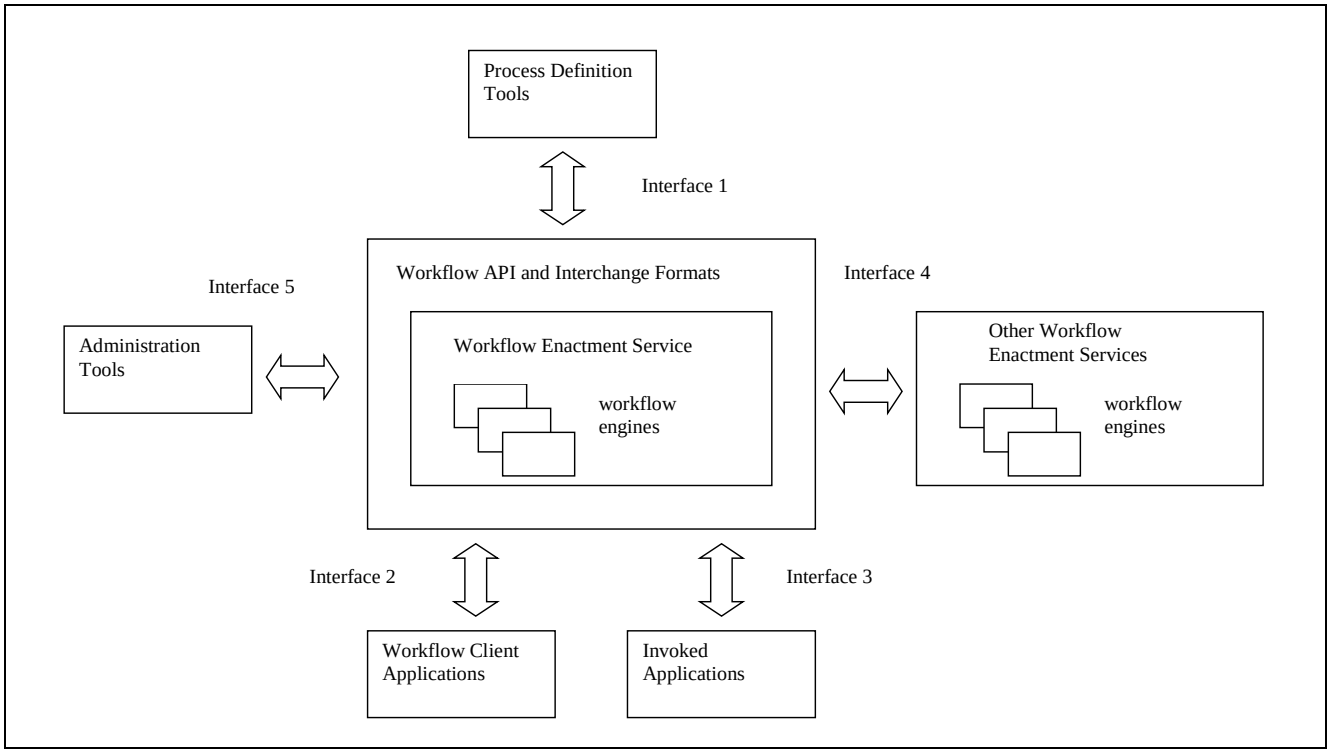


Figure 1 - Workflow Reference Model

documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." page 8 of [2]. Software tools to support workflow technology are known as Workflow Management Systems, and many commercial workflow products are available. There is a growing worldwide interest in workflow technologies, and quite a demand for workflow products [3]. The recent upsurge in electronic commerce has provided additional impetus to research and development of workflow products [4] [5]. The market for workflow products is vastly greater than the market for specialised software engineering tools such as a process centred software engineering tool. The central idea of our project is the treatment of software process management as a specialisation of a more general kind of process management known as workflow. Our approach is to use this more general purpose technology, that is workflow products, as a constructional technology for building the specialised software engineering tools. The cost of developing such specialised tools is generally quite high, and due to the small market for these tools, tool vendors need to charge very high prices to recover their costs. By using off-the-shelf workflow technology, we *aim to reduce the cost and time of developing* these specialised process management tools, thus providing a cheap and rapid approach to the delivery of custom process management tools to those organisations that could benefit from them. Furthermore, by conforming to workflow standards [6], we hope to produce tools with improved integration and interoperability facilities.

Our methodology in project is described as follows. We have taken an existing implementation of a software process management tool, referred to as the PSEE, and seen if another, functionally equivalent, version of this particular tool could be quickly and simply implemented using workflow technology. If this technique is feasible with the PSEE, then we should be able to use the same technique to produce a range of custom process management tools for managing different software development processes. The first step in this technique is to write a high level, implementation independent, description of the PSEE tool in a standardised workflow description language. This language is supplied by the WfMC Process Definition Interchange Process Model [6]. The resultant workflow process definition can then be imported into a WfMC conformant workflow management system for enactment of the software process model. Thus, the nucleus of a software process management tool has been automatically generated from high level descriptions

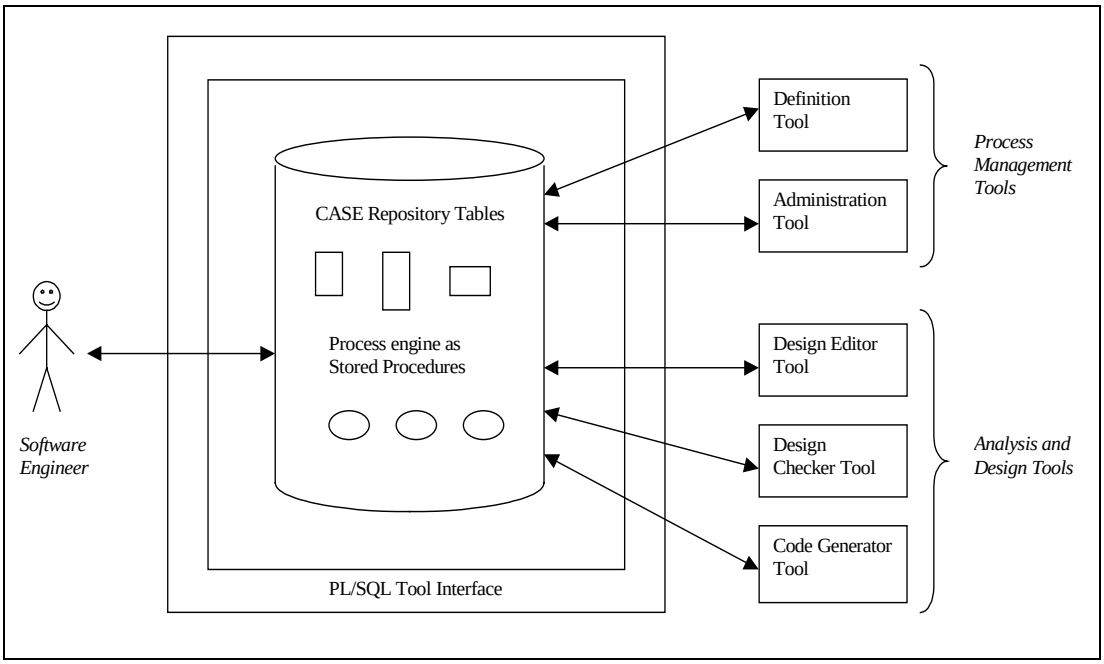


Figure 2 – PSEE for Process Management

with a minimum amount of work. To generate other process management tools, with different functionality to the PSEE, only the high level descriptions, written in the WfMC's standard language, need to be modified.

In the remaining sections of this paper we provide: more detailed background and motivation information about our work; a brief summary of key workflow and WfMC concepts; a description of the PSEE and an example of a software process model translated into a WfMC process definition. We discuss some of the results and issues addressed so far in our investigations, and we identify areas of further investigation.

2 BACKGROUND AND MOTIVATION

The current project arises from the authors' previous work in the area of CASE tool construction technologies and techniques [7,8,9,10]. A number of approaches to tool development have been explored and several technologies, including RDBMS's, stored procedural SQL, CDIF, Java, and formal specification languages, have been evaluated for their benefits to tool developers. This work has been motivated by the desire to discover cost and time effective development approaches for method engineers who wish to construct their own specialised custom software engineering tools.

One of the lessons learned from these investigations was the usefulness of readily available subsytems, such as an RDBMS, for implementing both tool repositories and process logic. Although a commercial RDBMS may not provide the same performance as a special purpose database for a CASE repository, the widespread availability, standardisation, and relative ease of access and programming of an RDBMS can be a significant advantage. Similar benefits of an RDBMS for low-cost implementations of workflow engines have been recently reported [11]. This report motivated the authors to examine workflow technology, in particular, the standardisation activities of the Workflow Management Coalition (WfMC) [12]. If RDBMS products can be useful constructional technologies for software engineering tool developers, and RDBMS products can provide low-cost implementation of workflow engines, what benefits might software engineering tool developers find in workflow technology?

3 WORKFLOW AND THE WFMC

Workflow is the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules [2]. A *workflow management system* defines, creates, and manages the execution of workflows through the use of software, running on one or more workflow engines, which are able to interpret the process definition, interact with workflow

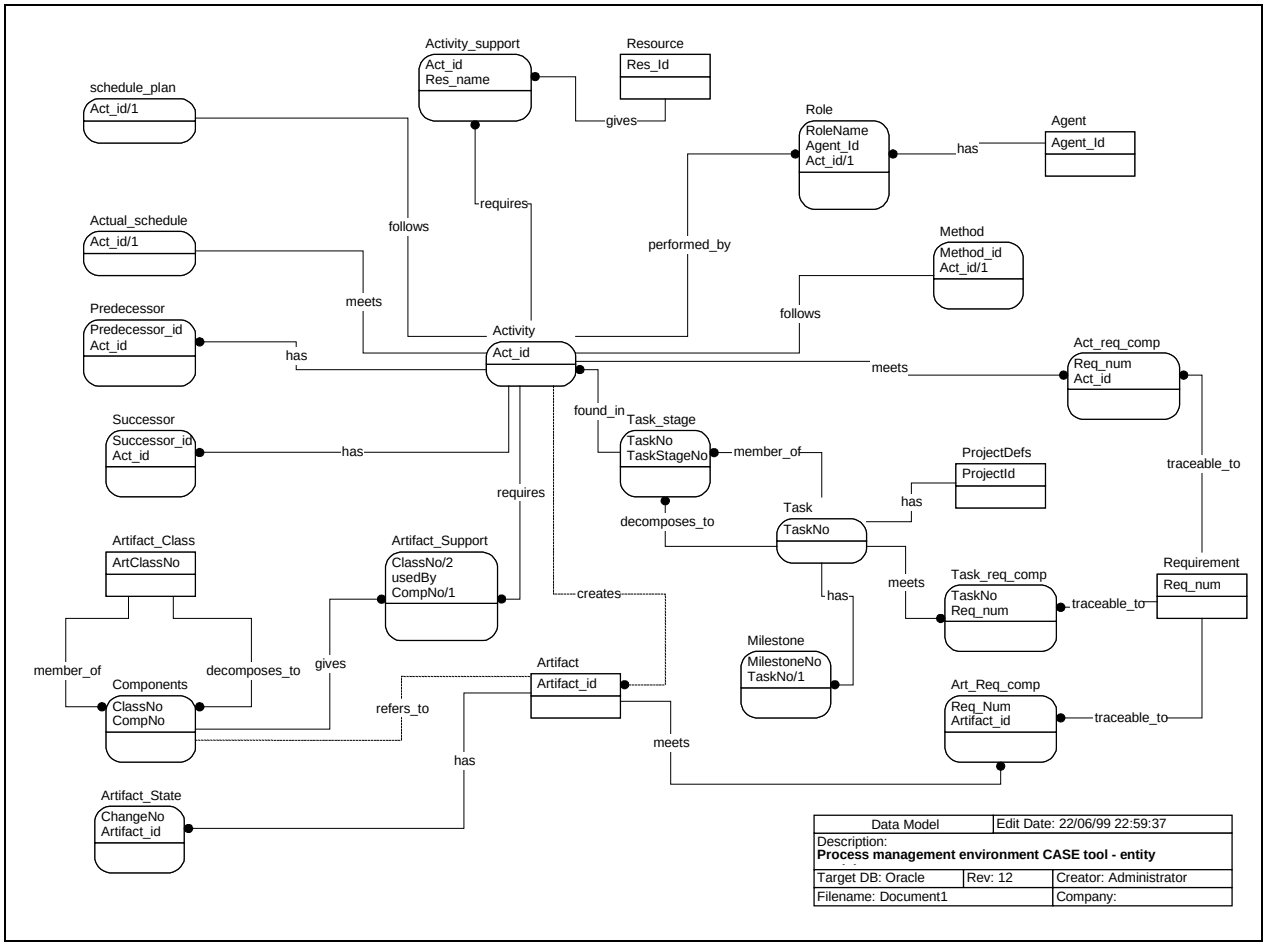


Figure 3 – PSEE Meta-schema

participants and, where required, invoke appropriate IT in the PSEE meta-schema to the entities in the Workflow tools and applications [2]. Workflow technology was first applied to document-oriented business processes such as approval processes in financial domains like banking. The

workflow coordinates the flow of documents between people, enforcing business rules for routing and deliveries. Participants access documents and update them according to their role in the process [13]. Workflow technology has subsequently been applied to areas such as Enterprise Resource Planning (ERP), Enterprise Application Integration (EAI), business component programming, and electronic commerce.

The Workflow Management Coalition (WfMC) is a group of companies who have joined together to establish a common “Reference Model” for workflow mangement systems [14]. The workflow reference model identifies a number of major components and a set of interfaces known as the WAPI - Workflow API’s and Interchange formats. These interfaces regulate the interactions between the workflow control software and other system components including process definition tools, administration tools, client applications, invoked applications, and other workflow engines (Figure 1).

In our investigations, we treat the software development process as a specialised kind of business process, in which documents, information, and tasks are passed from one participant to another according to a set of rules known as the development method. If the software process is considered as a specialised kind of workflow, then the software process management tool used to model and manage this process can be considered as a specialised kind of workflow management system. Using the WAPI

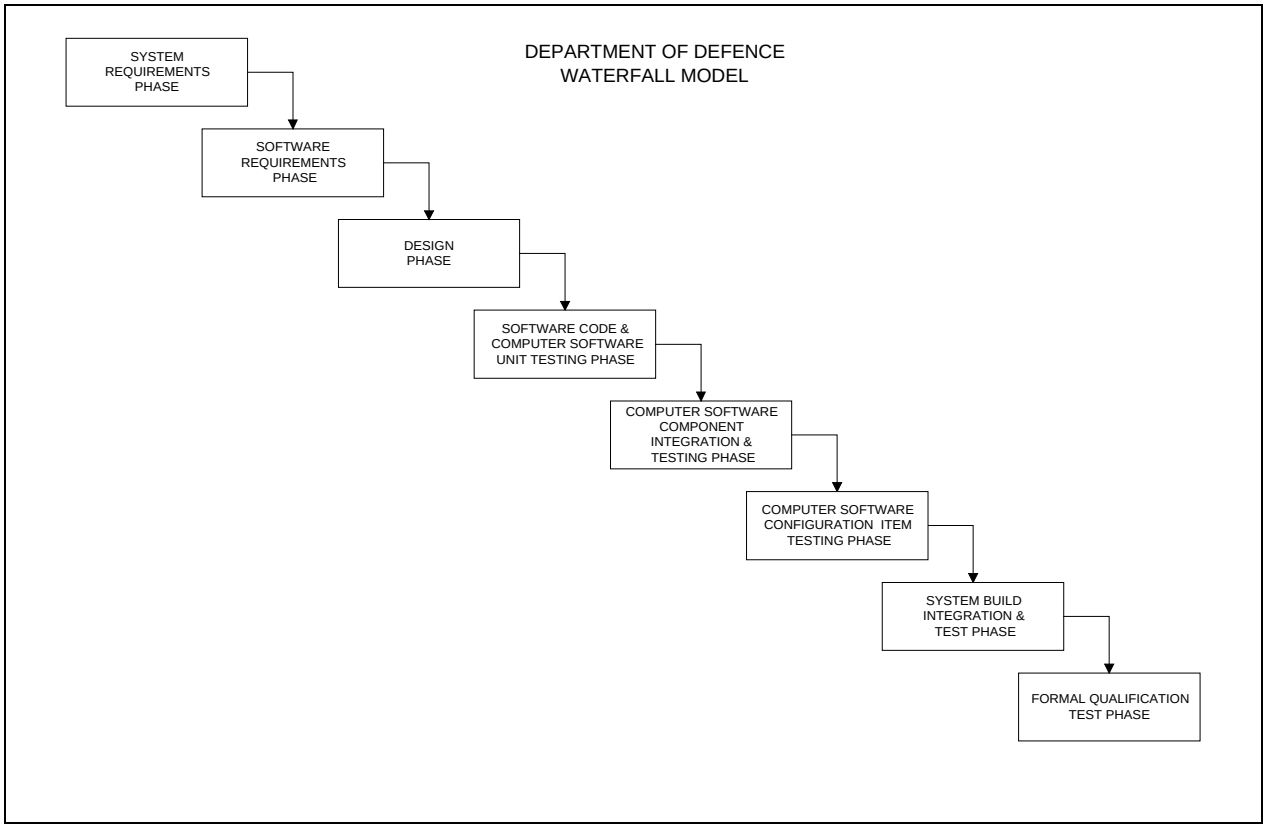


Figure 4 – DoD Waterfall Software Process Model

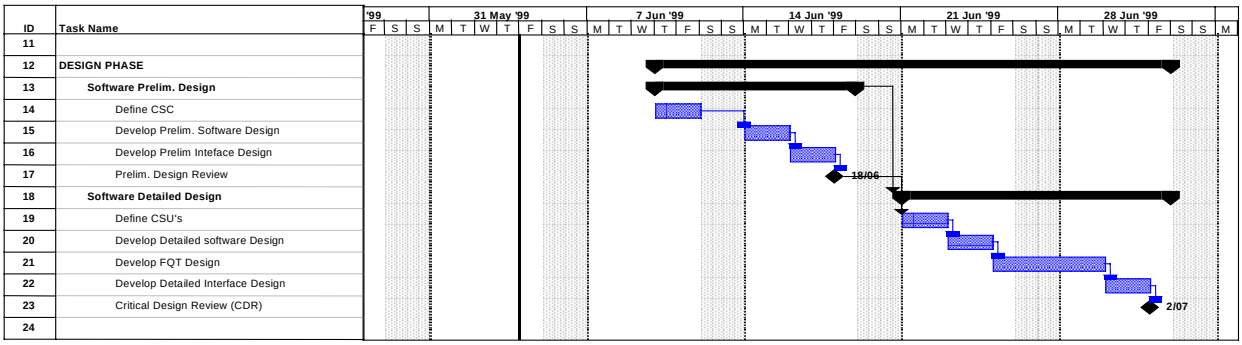


Figure 5 – Design Phase activities in an example project

interface 1 for Process Definition Interchange, a software development process can be defined as a workflow process in the WPDL grammar. Software engineers are represented as the workflow participants in this process definition, and their analysis and design tools are the workflow invoked applications. This process definition can then be interpreted by a workflow engine that is WfMC conformant, in order to model and manage the software process.

4 EXAMPLE PSEE TOOL AND PROCESS

To test the feasibility of the proposed approach, we selected a particular process management tool, referred to

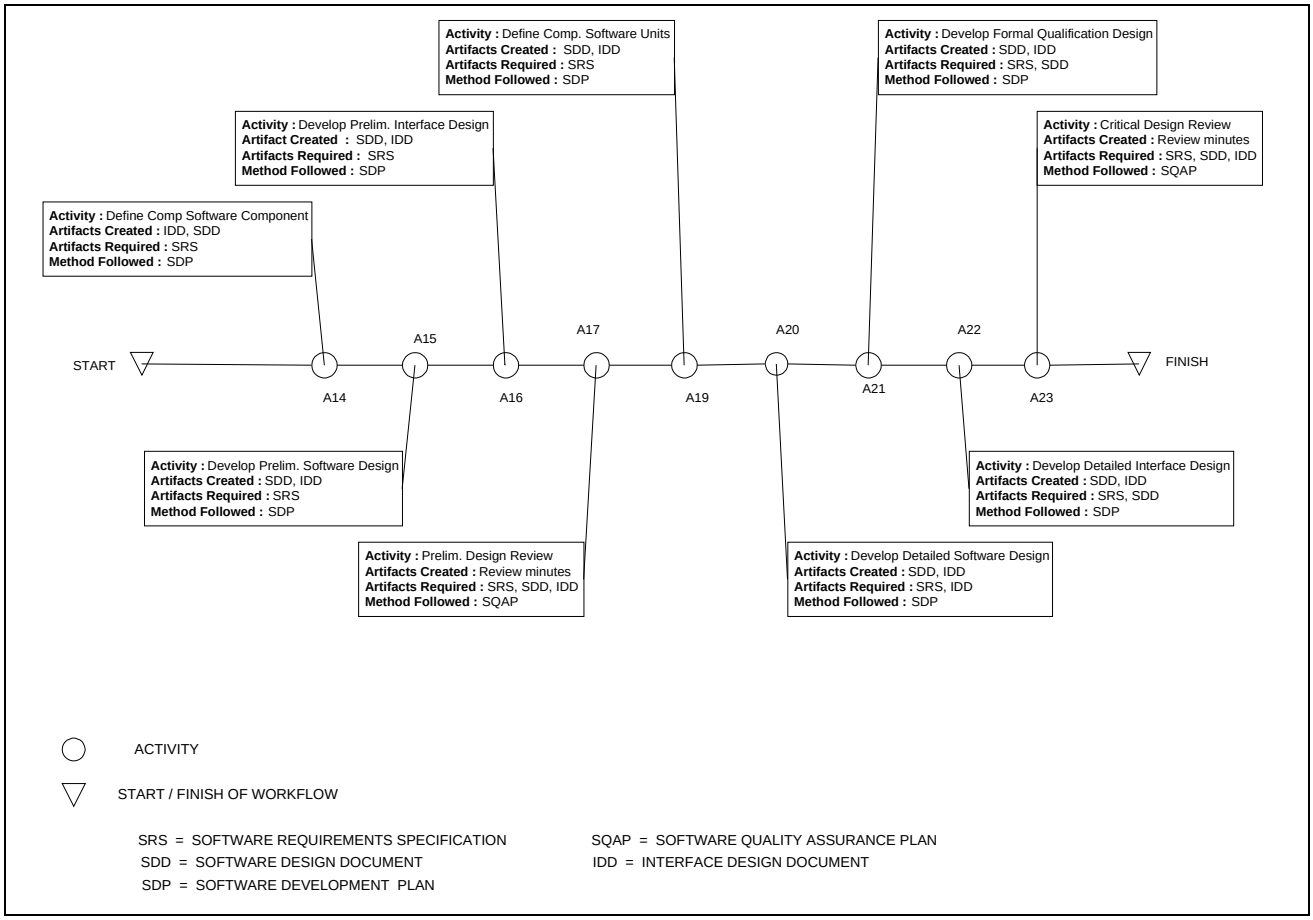


Figure 6 – Design Phase workflow for the example project

as the PSEE, along with a specific software development process that could be modeled and enacted by this tool. The PSEE implementation described here is a research vehicle previously developed at the University of South Australia. Although it does not have all the functionality

of a fully featured commercial process management tool, it has sufficient modeling capability for this exercise. The gross architectural features of the PSEE are shown in Figure 2. Essentially, we have a commercial RDBMS that is used to implement both a CASE repository and a simple software process enactment engine. The repository is a collection of relational tables defined by the schema shown in Figure 3. This schema, or meta-model, can store definitions of various process models. These models can be executed by the process engine, which is implemented as stored procedural SQL code within the DBMS. The executing model can invoke tools, and communicate simple notification messages to/from participating software engineers. Currently, the PSEE meta-model permits the definition of process models that are variants of the waterfall type of process model. Given this limitation, it is still possible to define quite rich and complex models of this general type incorporating features such as activity decomposition into subprocesses, iteration, roles and agents, artifact (document) usage and classification.

To illustrate how the PSEE meta-model represents a software process we will take a simple example of a waterfall process model shown in Figure 4. This is in fact the DoD Waterfall Model defined in [15]. We will examine the Design Phase of this process in more detail, and show how a particular software project, conformant to this model, is represented in the PSEE repository. Figures 5 and 6 show example data from the Design Phase of a conformant project with its associated activities, artifacts, and methods. In the PSEE repository, this data is stored in

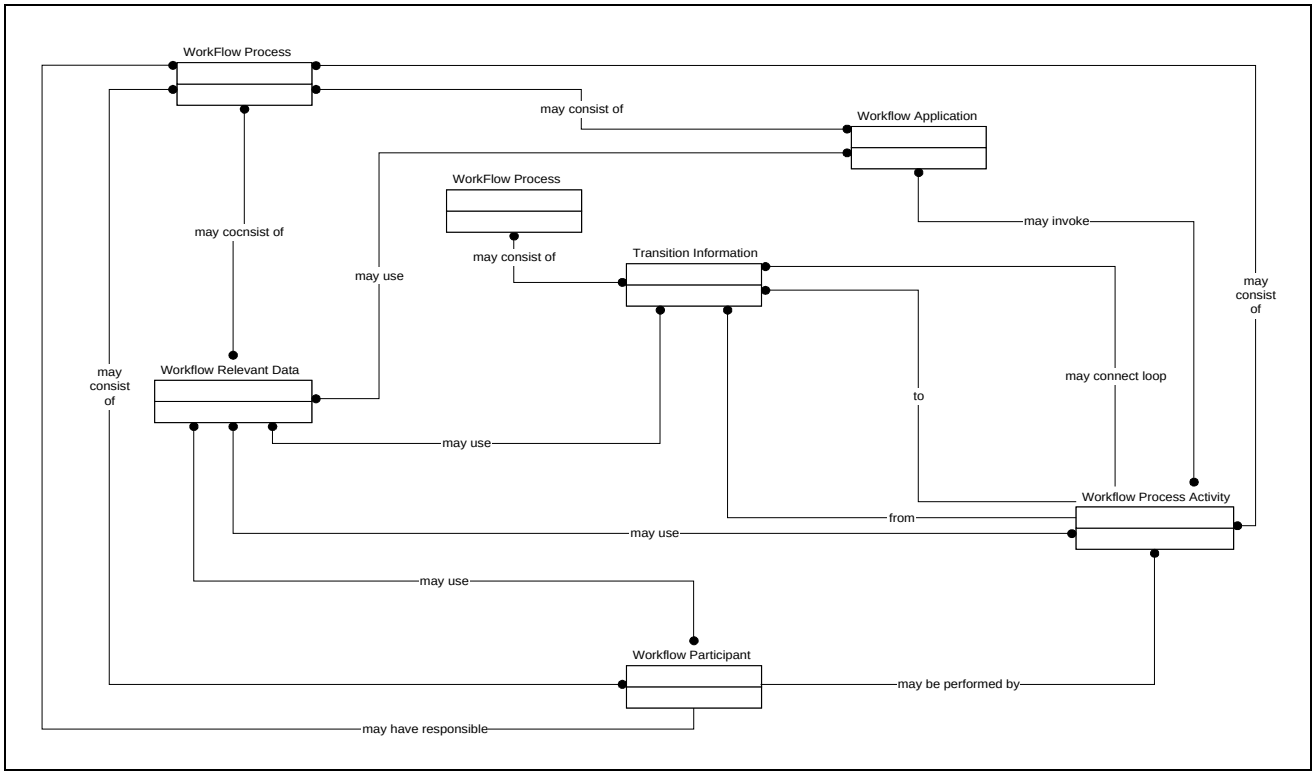


Figure 7 – Workflow Process Definition Meta-model

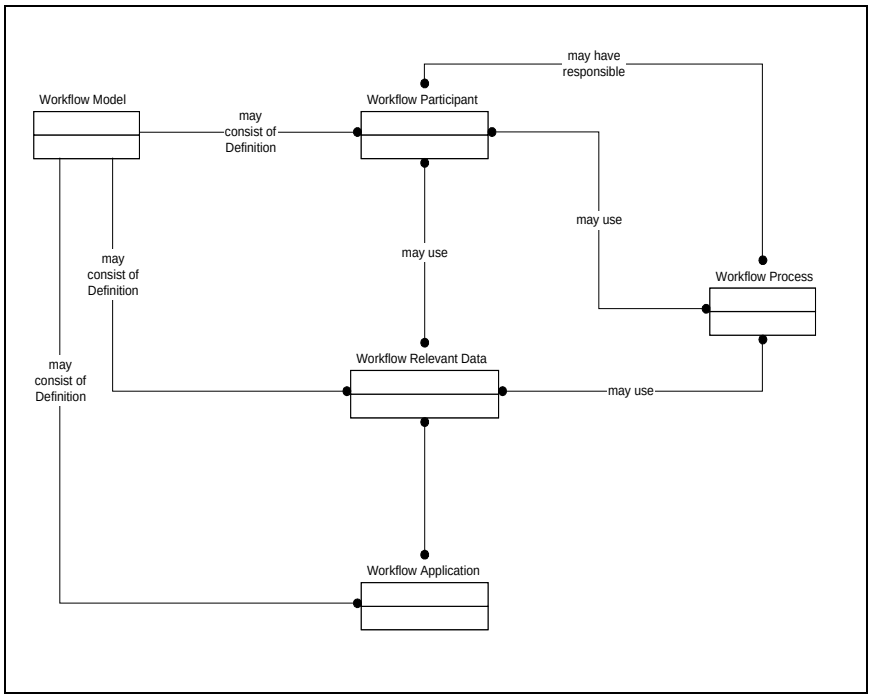


Figure 8 – Workflow Model Definition Meta-model

	Workflow Process Entity	Workflow Model Entity	Workflow Participant Entity	Workflow Process Activity Entity	Workflow Relevant Data Entity	Transition Information	Workflow Application
Workflow Model Definition Meta-Model	Activity_Support Activity Schedule_plan Actual_plan Predecessor Successor Artifact Artifact_State Milestone Artifact_Support Task Task_Stage Art_req_comp Act_req_comp Method Role	ProjectDefs	Agent		Requirement Task_req_comp Artifact_Class Components		Resource
Workflow Process Definition Meta-Model	Task Task_Stage Milestone		Role	Activity Method Schedule_plan Actual_plan	Act_req_comp Artifact_support Artifact_State Art_req_comp Artifact	Predecessor Successor	Activity_Support

Figure 9 – Mapping the PSEE to WfMC Meta-models

relational tables, and fragments of these table contents are provided in the appendix.

The WfMC reference model provides a basic process definition meta-model for workflow process definition at Interface 1 (see Figure 1). This meta-model identifies a basic set of object types for the interchange of simple process definitions. Further object types may be added by vendor specific extensions. There is also a textual grammar for the interchange of these process definitions known as the Workflow Process Definition Language (WPDL) [6]. Figure 7 shows the entities and relationships present in the Workflow Process Definition Meta-Model. Each of these entities has many mandatory and optional attributes for defining workflow processes of varying degrees of complexity. This meta-model includes some entities, such as Participant, Application, and Workflow Relevant Data, whose scope may be wider than a single process definition. The meta-model assumes the use of a common process definition repository to store these entity definitions. To allow these common entities to be referenced across process definitions, and to support the efficient transfer of data to/from the repository, a Workflow Model entity and additional relationships are introduced to this minimal meta-model (Figure 8).

The mapping of the PSEE meta-model to the WfMC Interface 1 is done in two stages. First, we map the entities Process Definition and Model Definition meta-models (Figure 9). The PSEE meta-model is being treated as a specialisation of the more general workflow meta-models. We therefore find cases of several specialised PSEE entities mapping onto one, more general, workflow entity. In the second stage, we translate the entities and attributes of the PSEE meta-schema into the WPDL grammar for import into a WfMC conformant workflow management system. This translation has been performed for the example software project described above. Fragments of this WPDL workflow description are provided in the appendix.

5 DISCUSSION AND SUMMARY

The WPDL workflow description of the example project has been checked through a parser supplied by the WfMC. The next stage in our investigations is to obtain one or more workflow management systems that are capable of importing this description written in the WPDL grammar. We can then complete the construction of a simple process management tool based on this technology.

Clearly, the construction of a working process management tool requires more than simply the definition and enactment of the process model. For the tool to be useful it must interact with the participants, that is, the software developers involved in the process, and it must invoke and communicate with the software engineering tools used by the participants. Other aspects, such as auditing, performance monitoring, and interoperability with other process management/workflow environments need to be considered. Our WPDL workflow description represents a simple waterfall software process model. Not all process models are this simple. We have not yet attempted to represent many of the more interesting features of process description such as iteration. Demonstrating a syntactic mapping between a software process model and a workflow process definition, in itself, does not demonstrate

that the behaviour of the enacted processes will be equivalent. Once we have constructed the workflow based software process management system, we can compare the behaviour, and determine what these differences are. We can also explore other issues of the tools' operation such invocation and notification mechanisms, and examine the possibilities of interoperation with other CASE and workflow environments.

Until more comprehensive process management tools have been completed, it will not be possible to determine if this really is a low-cost option. We are assuming that the market for workflow products, which is a more general application domain than software process management, is much larger than the market for specialised software engineering tools. Such economies of scale will hopefully reduce the purchase cost of a workflow product below the cost a software process management tool. Typically, these products cost in the range of $2000-$3000 per user. We also have to add to this, the development cost of defining the software process model, translating it into WPDL, importing the WPDL to a workflow management system, and configuring the system for support of software process management.

At this stage, it is difficult to tell if the WfMC reference model will be a widely adopted or long-lived standard. It is not uncommon in software engineering, and the IT industry generally, for standards to emerge and fail to be adopted. The WfMC standards are continuing to develop, and the WPDL may be superceded by an XML based process description language [13]. However, even if the particular features of the WfMC reference model do not prevail, the general characteristics of workflow products and process models are likely to persist for some time, and our approach to tool construction may still be promising. In this paper, we have reported upon our initial experiences with workflow technology and the WfMC reference model. Over the following months, we anticipate completion of more workflow-based software tools, and further reports on these activities will be produced.

REFERENCES

[1] I. Sommerville. *Software Engineering*. Addison-Wesley, (1995).

[2] Workflow Management Coalition *Terminology and Glossary.* WfMC Document TC-1011, issue 3, Feb 1999. http://www.aiim.org/wfmc/standards/docs.html.

[3] Workflow Management Coalition. *Workflow and the Internet: Catalysts for Radical Change*. A WfMC White Paper, 11 June 1998. http://www.aiim.org/wfmc/standards/docs.html.

[4] M. Anderson. *Interoperability – enabling E-commerce.* A WfMC White Paper, 1 April 1999. http://www.aiim.org/wfmc/standards/docs.html.

[6] Workflow Management Coalition. *Interface 1: Process Definition Interchange Process Model.* WfMC Document TC-1016-P, 12 November 1998. http://www.aiim.org/wfmc/standards/docs.html.

[7] J. P. Gray. CASE tool construction for a parallel software development methodology. *Information and Software Technology*, 39(4): 235-252, (1997).

[8] J. P. Gray and B. Ryan. Integrating approaches to the construction of software engineering environments. In *Proc. 8th Conf. on Software Engineering Environments (SEE'97)*, pages 53-65, Cottbus, Germany, 8-9 April 1997, IEEE Computer Society Press (1997).

[9] J. P. Gray and B. Ryan. Applying the CDIF standard in the construction of CASE design tools. In *Proc. Australian Software Engineering Conference (ASWEC97)*, Sydney, Australia, 28 Sept - 3 Oct 1997, IEEE Computer Society Press (1997).

[10] J. P. Gray and B. Ryan. TGE Approach for Constructing Software Engineering Tools. In *Proc. Australian Software Engineering Conference (ASWEC98)*, Adelaide, Australia, 9-13 November 1998, IEEE Computer Society Press (1998).

[11] R. Tagg and W. Lelatanavit. Using an Active DBMS to Implement a Workflow Engine. In *Proc. Int. Database Engineering and Applications Symposium (IDEAS98)*, Cardiff, UK, 8-10 July 1998, IEEE Computer Society Press (1998).

[12] Workflow Management Coalition (WfMC) web site at http://www.aiim.org/wfmc/

[14] Workflow Management Coalition. *The Workflow Reference Model.* WfMC Document TC-1003, 19 January 1995. http://www.aiim.org/wfmc/standards/docs.html.

[15] *Total quality management for software*. edited by G.Gordon Scheulmeyer, James I. McManus. Van Nostrand Reinhold (1993) ISBN 0-442-00794-9

[5] A. Sheth, W van der Aalst, I. B. Arpinar. Processes Driving the Networked Economy. *IEEE Concurrency*, 7(3), (July-Sept 1999) 18-31

[13] M-T Schmidt. The Evolution of Workflow Stanards. . *IEEE Concurrency*, 7(3), (July-Sept 1999) 44-52

APPENDIX

WORKFLOW PROCESS DESCRIPTION OF THE EXAMPLE SOFTWARE PROCESS IN WPDL GRAMMAR

This Appendix contains only a small sample of the process definition. The complete process definition is much larger and it can be obtained from the authors upon request. The corresponding relational table definitions and data from the PSEE repository can also be obtained from the authors.

```
// <Model>

MODEL    'GENERIC_WATER_FALL_MODEL'

   WPDL_VERSION "7.0 Beta"
   VENDOR      "VENDOR:PRODUCT:RELEASE"
   CREATED     1999-05-27
   NAME        "GENERIC WATER FALL"
   DESCRIPTION "WPDL-NOTATION OF GENERIC WATERFALL MODEL"
   AUTHOR      "ADB"
   STATUS      "UNDER_REVISION"
   EXTENDED_ATTRIBUTE  'ProjectId'  STRING  "PRJ001"

// <Workflow Participant List>

   PARTICIPANT         'BATNY002'
      NAME         "Norman Bates"
      TYPE         HUMAN
      EXTENDED_ATTRIBUTE   'EmailAddress' STRING "batny002@venus"
   END_PARTICIPANT

   PARTICIPANT         'LAFPY001'
      NAME         "Patrick LaFleur"
      TYPE         HUMAN
      EXTENDED_ATTRIBUTE   'EmailAddress' STRING "lafpy001@jupiter"
   END_PARTICIPANT

// <Workflow Application List>

   APPLICATION         'RES001'
      NAME         "EMACS"
      DESCRIPTION      "Text Editor"
      TOOLNAME         "emacs.exe"
      EXTENDED_ATTRIBUTE   'Res_Id' STRING  "RES001"
   END_APPLICATION

// <Workflow Process Relevant Data List>

   DATA        'Artifact_Class'
      NAME     "Artifact_Class"
      DESCRIPTION "Describes artifact classes used in SW Life Cycle"
      TYPE     ARRAY
            OF RECORD
               STRING 'ClassNo'
               STRING 'Name'
               STRING 'Type'
               STRING 'InUse'
            END
      LENGTH      [0...14]
      DEFAULT_VALUE    ( ("A" "PROJECT_ARTIFACTS" "STRUCTURE" "Y")
              ("B" "FUNCTIONAL_BL" "STRUCTURE" "Y")
              ("C" "SYS_REQ_SPEC" "STRUCTURE" "Y")
              ("D" "COMP_SYS_CONFIG_ITEM" "STRUCTURE" "Y")
              ("E" "DEVELOPMENTAL_BL" "STRUCTURE" "Y")
              ("F" "DESIGN_DOCUMENTS" "STRUCTURE" "Y")
              ("G" "CODE_UNDER_DEV" "STRUCTURE" "Y")
              ("H" "CODE_UNDER_TEST" "STRUCTURE" "Y")
              ("I" "COMP_SW_CONFIG_ITEM" "STRUCTURE" "Y")
              ("J" "COMP_SW_COMPONENT" "STRUCTURE" "Y")
              ("K" "COMP_SW_UNIT" "STRUCTURE" "Y")
   END_DATA
```

```
// <insert entity Task_req_comp data here as a complex data type>
// <Insert entity Requirement data here as a complex data type>
// <Workflow Process Definition>

WORKFLOW    'WATER_FALL_LIFE_CYCLE'

   CREATED  1999-05-27
   NAME     "WATER FALL LIFE CYCLE"
   DURATION_UNIT  D
   DURATION 94

// <Activity List>
   ACTIVITY    'SYSTEM_REQUIREMENTS_PHASE'
      NAME     "SYSTEM REQUIREMENTS PHASE"
      DESCRIPTION "DEVELOPING SYSTEM REQUIREMENTS"
      IMPLEMENTATION   WORKFLOW 'SYSTEM_REQUIREMENTS'
   END_ACTIVITY

   ACTIVITY 'SOFTWARE_REQUIREMENTS_PHASE'
      NAME  "SOFTWARE REQUIREMENTS PHASE"
      DESCRIPTION "DEVELOPING SYSTEM SOFTWARE REQUIREMENTS"
      IMPLEMENTATION   WORKFLOW 'SOFTWARE_REQUIREMENTS'
   END_ACTIVITY

// <Transition Information List>
   TRANSITION  T_1
      DESCRIPTION "Finish-Start Transition"
      FROM  'SYSTEM_REQUIREMENTS_PHASE'
      TO 'SOFTWARE_REQUIREMENTS_PHASE'
   END_TRANSITION

   TRANSITION  T_2
      DESCRIPTION "Finish-Start Transition"
      FROM  'SOFTWARE_REQUIREMENTS_PHASE'
      TO 'SOFTWARE_DESIGN_PHASE'
   END_TRANSITION

END_WORKFLOW

// <INSERT WORKFLOW 'SYSTEM_REQUIREMENTS' HERE>
// <INSERT WORKFLOW 'SOFTWARE_REQUIREMENTS' HERE>
WORKFLOW 'SOFTWARE_DESIGN'
   CREATED  1999-05-27
   NAME     "SOFTWARE DESIGN"
   DURATION_UNIT  D
   DURATION 17

// <Activity List>
   ACTIVITY 'SOFTWARE_PRELIMINARY_DESIGN'
      NAME  "SOFTWARE PRELIMINARY DESIGN"
      DESCRIPTION "PRELIMINARY SOFTWARE DESIGN"
      IMPLEMENTATION   WORKFLOW 'PRELIMINARY_DESIGN'
   END_ACTIVITY

   ACTIVITY 'SOFTWARE_DETAILED_DESIGN'
      NAME  "SOFTWARE DETAILED DESIGN"
      DESCRIPTION "DETAILED SOFTWARE DESIGN"
      IMPLEMENTATION   WORKFLOW 'DETAILED_DESIGN'
   END_ACTIVITY

// <Transition Information List>
   TRANSITION  T_1
      DESCRIPTION "Finish-Start Transition"
      FROM  'SOFTWARE_PRELIMINARY_DESIGN'
      TO 'SOFTWARE_DETAILED_DESIGN'
   END_TRANSITION

END_WORKFLOW

END_MODEL
```

SESSION 3

MODELING, TRANSFORMATIONS, AND GENERATION TECHNIQUES

An Approach for Generating Object-Oriented Interfaces for Relational Databases

Uwe Hohenstein

Siemens AG , ZT SE 2, D-81730 München (GERMANY)
Phone +49 89 636 44011, Fax +49 89 636 45450, E-mail: Uwe.Hohenstein@mchp.siemens.de

Abstract

Several tools and class libraries aim at supporting the access of relational databases from object-oriented applications by providing some kind of persistence layer. Nevertheless, a user's exertion of influence is sometimes too low, as the layer is mostly a black box. That is, the layer possesses a certain functionality, or it does not. Either the performance is sufficient, or is not. One has to live with the layer as it is, there are no possibilities to improve performance, to include new features such as advanced querying or transactions, or the other way round, to remove needless functionality.

This paper presents an alternative tooling, putting emphasis on highly customisable database layers. An adequate object-oriented access interface can be designed according to the application's needs. In order to keep the effort for implementation low, layers are generated out of an object-oriented design tool. It is shown how a UML model describing the persistent data can be used, and how the design tool can be enhanced to allow for generation. Experiences will stress on the advantages over commercial persistence layers: Flexibility is higher as the functionality can be freely designed. And transparency is given such that the structure of tables and the accesses to the database can be modified, e.g., for tuning the performance.

Keywords: Persistence layer, object-oriented database interface, Rational Rose, generator.

1. Introduction

Nowadays, it is commonly accepted that using object-oriented technologies in the software development process possesses advantages with regard to extensibility, flexibility, and reusability, thereby enhancing the productivity of programming. The object-oriented paradigm, through the notions of inheritance and encapsulation, reduces the difficulty of developing and evolving complex software systems.

Nearly all applications require a persistent storage of data in a database system (DBS). From the point of programming, it is desirable to store and retrieve objects of the application in an easy manner. Object-oriented DBSs (ODBSs) pick up this point bringing object-orientation into database technology. They enhance object-oriented languages to support database capabilities like persistence, transactions, and queries in a homogeneous manner so that the programmer gets the illusion of just one language. Owing to being a new database technology, ODBSs have got only a niche in the marketplace – in spite of their advantages in many application areas. Enterprises are just advanced to gain confidence in relational DBSs since robustness and reliability are gradually accepted. Storing data in relational databases, lots of applications have been developed recently. Since any DBS requires a large amount of administration, nobody will switch to a new database system without need. Hence, even new applications will use a system that is already available in a department or business unit. Replacing a relational DBS with an ODBS, as an alternative solution, is often impossible due to the so-called legacy problem [1]: There is a lot of data stored in relational databases. This data is a necessary input to many decision making processes, and thus an enormous investment of a company. Migrating the data in an object-oriented DBS is difficult and risky, frequently leading to an unacceptable lack of operation. Consequently, relational DBSs are still the state-of-the-art for many companies, even if the object-oriented paradigm is used in programming.

In fact, there is no principle problem to make relational data accessible from object-oriented programming languages. Relational database applications can be written by using SQL statements embedded in a general purpose programming language. Even if an embedding in C++, SmallTalk and Java is sometimes not supported, a detour via "Embedded SQL" for C can be taken. But database vendors are more and more enabling a direct embedding for C++ and Java, e.g., SQLJ.

Anyway, this approach suffers from the need to switch between two different languages and to interface them with extra programming effort. Both languages, the programming language for implementing application logic, and SQL for accessing tables in the database, are strictly separated. This is the so-called *impedance mismatch.* The languages follow different paradigms. While the programming language is procedural, SQL is descriptive, specifying what data should be retrieved instead of how this is to be done. Data exchange between

both languages requires special concepts such as host variables and cursors. Cursors allow receiving the query result one tuple at a time, and the values of each tuple are put into host variables, which have only scalar types. The disadvantages are obvious: The application receives tuples of atomic values and must convert them to objects. Hence, the *semantic gap* is coming to light: The application maintains complexly structured objects, while the relational DBS manages simple records in flat tables, because of different type systems. Objects have to be split up in order to fit into tables, and the other way round tuples have to be joined to build objects. This makes the handling cumbersome and application programs difficult to write and hard to read.

We accommodate ourselves to the significance of accessing relational DBSs from the programming language C++. We propose a layered system [2] that provides a flexible and homogeneous coupling of both worlds, solving the problems of impedance mismatch and semantic gap in an elegant way. The impedance mismatch is avoided by staying completely in C++. Database features are encapsulated in predefined classes and methods. Hence, our proposal hides the specific coupling mechanisms of relational systems. C++ programs is given the ability to invoke database functionality in a convenient and comfortable way.

In fact, there are some commercial C++ class libraries and tools such as RogueWave's DBTools.h++ that attempt to ease the access of relational databases for C++ applications. Section 2 will discuss some disadvantages of those tools. Essentially, commercial tools provide persistence layers that do often not possess enough flexibility to support specific demands. The other way round, there are few possibilities to reduce the functionality to a degree really needed by the application. Moreover, the layers are black boxes. This is particularly critical if a lack of performance requires database tuning since this has to be done in the implementation part (which is hidden by the tool). A layer implemented by one's own provides better adaptability and flexibility, but requires more effort for implementation.

In order to reduce the effort for developing application-specific persistence layers for a relational database, we suggest generating the implementation of the layer. Section 3 will first present a simple persistence layer with an object-oriented C++ interface, before the process of generating the implementation is illustrated in Section 4. To this end, we take benefit from object-oriented modelling tools.

Section 5 reports on experiences we made with the generative principle in a concrete project. The main advantage of our approach is that flexibility and adaptability are completely under control. The persistence layer is customisable with regard to application's requirements on database functionality. Performance can be improved that way.

Section 6 finally outlines some future work we are planning to do.

2. Persistence tools

Using embedded SQL is the classical way to build relational database applications. But such a programming of object-oriented database applications is very complex due to the impedance mismatch and the semantic gap. Fortunately, there are several products on the market that promise to make this task easier. However, there are still some deficits we want to discuss.

There are products such as RogueWave's class library DBTools.h++, the Java database interfaces JDBC ("Java Database Connectivity") and SQLJ, or Microsoft's palette of interfaces, e.g., ODBC ("Open Database Connectivity"), ADO ("Active Data Objects"), RDO ("Remote Data Objects"), DAO ("Data Access Objects") and OLE-DB, partially in combination with the MFC ("Microsoft Foundation Classes"). All these interfaces behave similarly: They encapsulate the concepts of relational database technology, e.g., transactions, tables, columns, queries etc. by means of corresponding C++ classes `TRANSACTION`, `TABLE`, `COLUMN`, `QUERY`, or similar. In fact, the handling of relational databases gets a C++-like appearance making programming easier, but the gain with regard to object-orientation is only little. Applications still have to manage tables and to use SQL for manipulating data. Even if the syntax may be different, the concepts of SQL must be known when querying data. Since a lot of application developers have not much experience with relational database design and SQL, those tools will not solve the big problems. But without doubt this is an important step to achieve independence of concrete database systems; using the interface eases a later switch to another database system.

Other tools aim at providing more comfort by managing C++ objects in relational databases. Those products start with an object model [1] modelling the data to be stored in the database in an object-oriented manner. The object model is then automatically mapped onto relational tables using some strategies. Moreover, a database layer is generated that breaks down objects into tuples accordingly. The layer provides an interface that allows handling objects of the object model. It is possible to store, retrieve and to delete objects in the database, independently of how they are represented in tables. Programming database applications thus becomes easier. Particularly, the programmer is no longer responsible for designing tables since the tool does this job implicitly. Well-known tools are Persistence, POLAR (IBL Ingenieurbüro), ObjectFactory (RogueWave), JavaBlend and TOPLink (The Object People). The high-end of those tools more or less emulate object-oriented DBSs on top of relational ones. But they can certainly not compete with the performance of real ODBSs, which are optimised to handle complex structures and inheritance

[1] The term "object model" here means the result of modelling data, i.e., a schema in database speak. It should not be understood as the modelling language, e.g., UML (Unified Modeling Language) or ER (Entity-Relationship).

already on the physical storage level. For example, the typical traversal from object to object is an operation that is performed very quickly in ODBSs. Doing this with an object-oriented layer will take much more time as either several accesses to the database are necessary, or costly joins must be performed on tables. Subtyping is for free in ODBSs, but in contrast requires several operations on the relational database. It is not only the number of accesses, but also the internal overhead required to emulate ODBS features, e.g., a caching mechanism to synchronise several copies of database objects in memory. All these things cause some overhead that leads to a loss of performance in any case, especially if object-oriented accesses are used extensively. The comfort given by object-oriented accesses is traded for worse performance. We made some studies that reveal a loss of 40%!

A lot of applications can presumably live with the loss of performance, particularly, if the profit from faster application development is taken into account. But if performance is critical, applications will run into trouble because tuning the database layer comes up against limiting factors. For example, it is often not possible to change the structure of tables afterwards, because the tools operate in a top-down manner: If the tables are not adequate due to performance reasons, one has to modify the object model in such a way that the tool will produce the desired, more efficient tables. This is very cumbersome because the internal strategies must be understood and turned up. Thus, the choice of the object model depends on achieving certain tables. And even more worse, any change of tables requires a new object model, which affects the application heavily! By the way, this is also a problem for accessing existing "legacy" tables. Here again, the design of the object model is essentially shaped by the tables. Similarly, there is only little flexibility to modify the (relational) database accesses because the persistence layer is a black box. Unfortunately, changing the table structure and reorganising the accesses possess the best potential for improving the performance. The full potential for tuning is only available if the code generated by the tool is understandable, and changeable – but it is mostly not.

Finally, we want to mention some special data re-engineering approaches [3], which aim at accessing existing tables. [4, 5] discuss a generative, specification-based approach. It is based upon a specification that describes the re-engineering of tables explicitly, i.e., how tables can be modelled as object classes. Hence this is a bottom-up approach starting with tables. There are very flexible mechanisms to remodel relational data in an object model. A specification is input to a generator that produces a C++ interface conforming to the ODMG2.0 standard [6] for ODBSs. In other words: The approach provides an adjustable, object-oriented access to relational database systems, completely simulating an ODBS on top. Similarly, the prototype COMan [7] is working. Both tools are only available as research prototypes, beyond having maturity for the market.

It is important to note that the newest database technology of object-relational systems such as Informix Universal Server with the Universal Data Option and Oracle8 also do not provide a satisfying solution. Besides embedding SQL in C++, they integrate object-orientation in the DBS kernel. This means that tables may now be complexly structured. A user can define new data types including operations (in principle classes) that can be used as attribute domains. Nevertheless, it is not possible to store C++ objects directly as database objects; there is still a mismatch to C++, as there are two separate worlds with different concepts for relationships and inheritance.

In spite of the criticism, it should be mentioned that the tools are useful as they make programming relational database applications easier. [8] estimates the effort for developing a persistence layer at 30% of the planned resources! Nevertheless, there are special situations that require an enhanced influence on the object-oriented layer, e.g., due to performance reasons. Then it is inevitable to establish an own persistence layer between object-oriented application and relational database.

3. Persistence layer

We now introduce a simple persistence layer for relational databases. Later on, we will show how to generate the implementation of the layer. The layer should satisfy the following requirements:

- The interface provided by the layer supports database functionality in an easy manner.
- Especially, the relational database technology (tables and SQL) is transparent.
- The principle is in some way schematic and intuitive. This makes it possible to use the interface by just knowing the object model.

An object model, i.e., an object-oriented design of the structure of an application, is a good starting point. The object model comprises the model of persistent data. More precisely, we should use the term "database model" to denote the subset of *persistent* data in the object model. The database model may omit single attributes of a class or complete classes. It is useful to let the database model be a subset of the object model. But sometimes there are good reasons to keep them apart in two separate models. Then there is a higher independence between application and persistence, but a need for some mediation that makes the usage more complex: The database model is used to manage data from the database, which then must be converted in such a way that it fits to the object model.

Our approach relies on a database model being a subset of the object model. The database functionality is then related to objects of the object model: Real objects are stored, retrieved and deleted. Knowing the object model is enough to use the database interface.

Such a persistence approach can be designed in several ways. [9] gives a good overview of object/relational access layers and discusses several implementation aspects (which affect in turn the design of the interface). When designing such a layer, there are

different forces: On the one hand, the layer should satisfy all the needs from a functional point of view. On the other hand, we should keep the layer as simple as possible in order to enable generation. Particularly, the part to be generated should be small.

We here pursue a *shadow class* concept which is very practicable in most situations: Each persistent class `A` obtains a superclass `DBM_A`; DBM stands for database management. The DBM shadow class provides all the methods needed for handling objects in the database: There are methods to create, find, store, and delete objects in the database. Associations between classes are reflected by methods that allow for traversing from an object to related ones. Special types of associations such as aggregations/compositions in UML [10] may affect the semantics of methods. For instance, it is useful to treat aggregations/compositions as "complex objects": Fetching an object that has aggregations will also get the associated parts, and similarly for removals. Hence, complex objects are managed as a whole.

Let us discuss the principle of the persistence interface in more detail by means of an example. We assume the following object model:

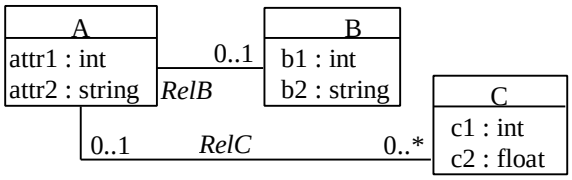


Figure 1: Sample object model

The shadow class approach enhances the object model by several DBM superclasses as shown in Figure 2:

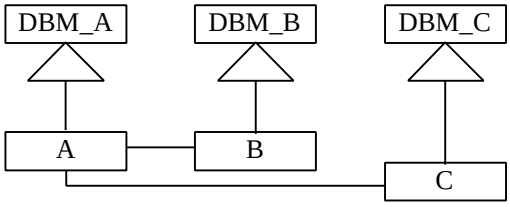


Figure 2: Shadow class approach

The signature of the `DBM_A` class declares methods to handle objects of class `A`:

```
class DBM_A {
public:
   int create();
   int store();
   int remove();
   static A* find(<key>);
   static Set<A*>* find(<query>);
   // for to-1 association RelB:
   B* getRelB();
   int setRelB(B*);
   // for to-n association RelC:
   Set<B*>* getRelC;
   int addRelC(C*);
   int removeRelC(C*);
protected:
   virtual int getAttr1()    = 0;
   virtual string getAttr2() = 0;
}
```

`DBM_B` and `DBM_C` possess analogous signatures. The method `create` is used to create new objects in the database, while `store` overwrites existing objects. We assume that each class possess an identifying property, i.e., a key attribute or a set of attributes. This key is used to determine whether an object is new. If an object that already exists in the database is created, an error is issued. The return value detects the success of a method.

The `find`-method is `static` as the invocation is independent of an object. `<key>` should denote the key attribute(s) of the class; hence it is guaranteed that at most one object is returned. More precisely, it is a pointer to the object. `<query>` should denote some kind of general associative query. In the simplest case, several attributes may occur as parameters. Then all those objects are searched that possess the passed values as attributes. For example, `find(String a2)` will return all the `A`-objects that possess the parameter `a2` as value of `attr2`. For advanced query capabilities, SQL-like conditions are possible. This can either be done by passing a string to `find`, specifying the condition in a query language, or by means of a `Query` class that allows constructing queries. Anyway, the language should rely on the object model in the sense of an ObjectSQL [11]; relational SQL is no good choice because it is based upon tables which are supposed to be hidden by the layer. The result of a query is represented by a `Set` as it is typically provided by class libraries, e.g., templates `RWTPtrSlist<T>` in RogueWave's Tools.h++. The collection templates offer methods to iterate over the result set.

Associations of the object model are reflected by traversal methods. Invoking `getRelB` for a given `A`-object returns the `B`-object that is related to by `RelB`. In case of to-n associations, e.g., `RelC`, a set of objects is yielded. The methods `setRelB` and `addRelC` can be used to establish new relationship instances. Due to to-n, `addRelC` adds a new instance to the already established ones, while `setRelB` overwrites a to-1 association. Analogously, `removeRelC` removes a relationship.

The (pure) virtual `get`-methods are necessary, because class `DBM_A` needs to access the attributes of class `A` when implementing the methods `create/store`. The protocol is as follows: `DBM_A` only declares a protected method `<datatype> get<Attr>()` that gives the DBM layer access to any attribute `<Attr>`. The implementation of the method must be done in class `A`. Analogously, the implementation of `DBM_A::find` has to create `A`-objects and to fill the attributes with values found in the database. Hence, class `A` must possess a constructor to set all the attributes.

Owing to inheritance, using the database functionality is very easy and intuitive. Here is a sample program:

```
DBM_Transaction t;
t.start();
// create an A-object in memory:
A* objA = new A(...);
// store object in the database:
objA->create();
// modify the object in memory:
objA->setAttr2("new value");
// write the object back to database
// (the object must already exist):
objA->store();
// get related B-object (via RelB):
B* objB = objA->getRelB();
// get the related C-objects:
Set>C*>* setC = objA->getRelC();
// establish association between objA,objC:
C* objC = new C(...);
objA->addRelC(objC);
// query: find all A-objects with name
// "Ms. Marple"
Set<A*>* result = A::find("Ms. Marple");
A* obj;
// iterate over result:
while (obj = result->next())
{ ... obj->method() ... }
// commit all the modifications:
t.commit();
```

Obviously, the interface smoothly fits to C++; the handling is similar to an ODBS. The usage of the interface is closely related to the object model, as the object-oriented concepts are directly reflected. The principle is schematic, all objects are handled in the same manner. Hence, the database functionality is immediately comprehensible. Transparency is given as the underlying relational database is invisible. Objects and associations are handled, no matter whether they are found in one or more tables. Traversals can be performed without knowing how to join which tables. Consequently, there is a strong independence between application programming and the relational database.

The DBM classes in total implement a layered system in the sense of [2]. The generic interface yields enough abstraction from "physical" database details allowing for an efficient implementation.

Implementing the DBM classes is very schematic and straightforward. A sample implementation may look as follows. We consider the method `create` for inserting new objects in the database. To ease embedding SQL in C++, we here use the class library DBTools.h++. Programming thus becomes simpler and the code is independent of a specific database vendor.

```
static const RWCString tabName("tabA");
// table tabA(a1,a2) for class A (cf. 4.4)

int DBM_A::create()
{
   int failure = 0;
   RWDBConnection* cPtr =
      DBM_ConnectionMgr::getConnection();
   /* get connection to database,
      if not already existing    */
   RWDBTable tab(tabName);
   RWDBInserter ins = tab.inserter();
   try
   { /* store atomic attributes of object
        in tabA  */
     ins << getAttr1() << getAttr2();
     ins.execute(*cPtr);
     cache.insert(new Entry
       (Key<A>(getAttr1()), // key
        this,       // pointer to object
        NEW,        // status of object
        1);         // reference counter
   }
   catch(const RWxmsg& x)
   { DBM_ErrorHandler::set(failure); }
   return failure;
}
```

The code aims at keeping the class-specific part as low as possible. In fact, this is the part to be generated. Consequently, connecting to a database and error handling are put in separate classes `DBM_ConnectionMgr` and `DBM_ErrorHandler`.

Handling database errors in DBTools is quite easy. An error handler can be installed by means of `cPtr->setErrorHandler(DBM_ErrorHandler::handler)`. Any occurrence of a database error will then invoke the method `DBM_ErrorHandler::handler`. This routine analyses the error, converts the DBTools/DBS error into a DBM error number, and throws a `RWxmsg` exception. A `catch`-block catches the exception and lets the variable `failure` be set with the DBM error. Doing that way, errors such as "unique constraint violation" are handled and changed to "object already exists". In fact, we let the database system check for uniqueness of tuples (objects) by defining primary key constraints.

The piece of code is intuitive and shows the principle. One point is worth mentioning, a *cache* that controls the usage of objects. This is important because an object can be retrieved several times by multiple `find` or `getRelB/C` invocations. A (too) simple implementation would maintain several copies of this object in main memory. As a consequence, the copies would evolve independently of each other. If objects are written back to the database, then object changes get lost, as the last one would override all the previous ones, unless there is some synchronisation mechanism [12]. The cache now cares for synchronisation in the following manner. Any object occurs only once in main memory, other pointers refer to that copy. Hence, all modifications are made to the same copy, and writing this copy back will include all changes. This implies that the implementation of `find` looks in the cache before fetching an object from the database.

The cache maintains a hash table of all objects currently fetched from the database. An entry of this list consists of the object's key value (encapsulated in a class `Key<A>`), a pointer `A*` to the object, a status (`NEW`, `DELETED` etc.) and a counter that counts how often an object is referenced. Committing a transaction, all the modified objects in the cache are written to the database.

The example is intentionally kept simple. Class `A` possesses only atomic attributes. Complex attributes and embedded objects (by aggregation/composition) must be handled similarly. Furthermore, subclasses will require some additional actions. In case of `find`, an object of the most specific subclass must be determined in order to support polymorphism correctly. In order to set attributes of the superclass (which are inherited to a subclass), a table related to the superclass must often be accessed, according to some mapping strategy.

Please note this is just one proposal for a persistence layer. We here want to keep the layer simple and understandable for reasons of a later generation, but also to show how simple a layer may be. Nevertheless, the layer can be enhanced in any way to bring it into line with specific requirements. The reader is referred to [13,8] that discuss important aspects of designing scaleable object-persistence layers in more detail.

4. Generative approach

4.1 The principle

This section describes how to implement a generator that produces persistence layers such as the one discussed. In order to keep the effort for development as low as possible, we integrate the generator in an object-oriented design tool. This has the advantage that the graphical user interface, especially for modelling the structure, can be used. There is no need for developing a user interface of its own for the generator. The generator can focus on its real task, generating code.

All the well-known object-oriented design tools are extensible in a certain sense. They possess

- a repository that maintains all the information about existing object models (e.g., in .cat-files in ROSE),
- an interface to access the meta-data in the repository,
- an opportunity to define additional properties for object models, and
- means to implement and run scripts within the tool.

These are technical prerequisites to integrate a generator in a tool. The procedure of implementing a generator can then be done stepwise. We here focus on Rational ROSE to give the discussion more technical depth. The ideas can be transferred directly to other design tools, too.

1. Starting point is a database model (as part of the application's object model) specified in ROSE by means of UML ("Unified Modeling Language") [10]. UML allows for an object-oriented modelling of the application structure. Moreover, dynamic aspects can be described by sequence diagrams etc., but this not relevant here. We are interested in modelling the structure since the resulting ROSE model is a superset of the persistent objects. It does not matter whether the database model is subset of the object model or whether is apart from it.
2. The ROSE model is enhanced with additional "properties". Those properties can be attached to ROSE concepts to carry additional information, which is needed by the generator and must be supplied by the user of the generator. Examples are markers for the persistent classes, their persistent attributes, key attributes, what attributes are used for querying (in `find`) etc. Furthermore, properties may represent mapping strategies, how to map set-valued attributes onto tables, i.e., how to handle subtyping and so on. Properties are organised in new folders (named "tools" in ROSE) that occur in ROSE specification windows (cf. the tool "CODE" in Figure 3).
3. Generators are implemented in a script language similar to VisualBasic Script. The script language includes special classes and functions to access the repository, i.e., to get information about the modelled classes, their attributes and associations, inheritance hierarchies and so on. Using this meta-data, SQL statements can be generated that install tables in the relational database. Information about key attributes and mapping strategies can be obtained by querying the generator-specific properties.
4. Taking each persistent class of the ROSE model, a generator can produce the corresponding DBM class. The set of all DBM classes together with some predefined classes constitute the persistence layer.
5. In order to have a complete description of all the software, the model of the DBM classes can be generated and inserted into the repository. That is, the DBM part does not need to be modelled with the design tool manually!

Subsections 4.3 to 4.6 will explain how to perform the steps 2 to 5. The repository of ROSE is the central part for the steps. The essentials of the repository and its meta-model are summarised beforehand in Section 4.2.

Having developed the generators, usage is as follows from an end user's point of view. It is first necessary to mark the persistent classes in the ROSE model, and to indicate the persistent attributes and associations. Classes are specified in a "Class Specification" diagram in ROSE. The diagram possesses a property that can be switched to "persistent". Alternatively, the user can click on persistent classes in ROSE before starting the generator. The generator then takes the chosen classes into account.

Similarly, the user has to indicate what attributes and associations of a class become persistent. To this end, newly defined properties are used. Figure 3 displays the attributes of class `A` in the "Attributes" folder of the "Class Specification" window (left). Selecting "attr1", a "Class Attribute Specification" diagram appears. This window possesses a new folder "CODE"; being activated it presents the new properties for interactive modification. Hence, the user can mark "attr1" as a key by filling in "yes" for the "Key" property. Any attribute is assumed to be persistent by default, but this value can be overwritten. Similarly, special strategies for handling multi-valued attributes and subtype hierarchies can be managed. Analogously, the "Class Association Specification" is to be handled by means of the folder "Relations".

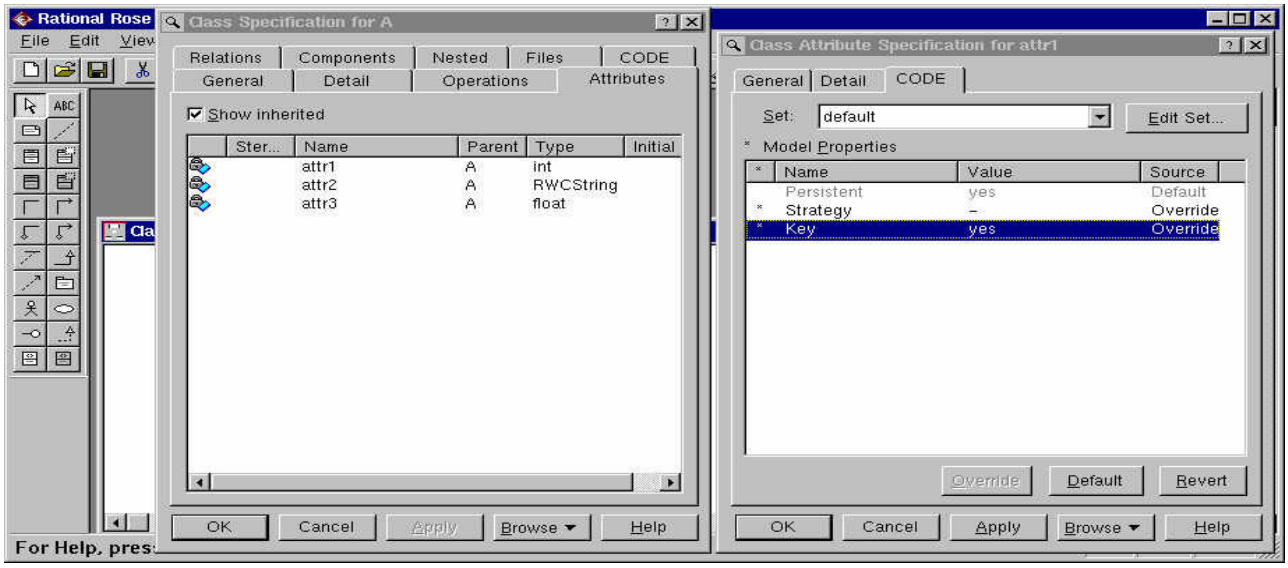


Figure 3: Class attribute specification

Having enriched the ROSE model in that way, the ROSE script implementing the generator can be selected in the main menu and executed. Consequently, enhancing an object model with generator-specific information is completely integrated in ROSE!

4.2 The Meta-model of ROSE

Design tools possess a repository, which contains the information about all the object models. The organisation of this meta-data is done according to a meta-model. The meta-model of ROSE exposes all the UML concepts by corresponding meta-classes such as `Class`, `Attribute`, `Association`, `Operation`, `Role`, and `Parameter`, reflecting the concepts directly. Hence, the meta-class `Class` maintains all the classes of a ROSE model as instances. Every meta-class possesses a method `Name` to determine the name as a string: Having an instance `aClass` of `Class` (i.e., a class of the model), `aClass.Name` computes the name of that class. Traversal functions relate meta-classes. A function `Attributes` computes the attributes of a class. Similar functions can be used to determine the associations of a class (`Associations`), the roles of an association (`Roles`), the superclass(es) etc.

Each concept possesses a corresponding collection class `...Collection`, e.g., `AttributeCollection` maintains a collection of `Attribute` instances. This is because some functions return a collection. For example, `aClass.Attributes` returns an `Attribute-Collection`, which contains all the attributes of `aClass`. Those collection types have `GetAt(int)` and `Count` functions to iterate over the collection:

```
FOR i%=1 TO aClass.Attributes.Count
' for any attribute of the class:
   SET anAttr = aClass.Attributes.GetAt(i%)
   ...
NEXT i%
```

The meta-model comprises all ROSE-specific concepts, even those that have an organisational purpose such as packages and categories. They are used to modularise and partition large ROSE models. Hence, an object model can be split into several categories, each one modelling a part of the software system. It is sensible in our case to put the DBM classes in a category of its own.

The repository is the central point to get information about object models. Particularly, there are means to ask the settings of generator-specific properties. The properties are an elegant way to enhance an object model with additional information, which is not purely related to modelling, but rather to generating code.

4.3 Definition of new properties

As mentioned in Subsection 4.1, the generator needs additional information to control generation. ROSE can be extended by properties that can hold additional input. Adding generator-specific properties to ROSE is done by means of the meta-class `DefaultModeProperties`. Having an object model loaded in ROSE, the following lines will introduce a new property “Persistent” for the “Class Attribute Specification” diagram.

```
DIM prop AS DefaultModeProperties
SET prop =
    RoseApp.CurrentModel.DefaultProperties
prop.AddDefaultProperty
  ("Attribute", "CODE", "default",
   "Persistent","String", "yes")
```

The first parameter of `AddDefaultProperty` determines the concept to be enriched, here “Attribute”. Any property must belong to a folder in the diagram. We name the folder “CODE”. Any further call of `AddDefaultProperty` with the same folder name adds a property to the already existing folder. The next parameters define the new property “Persistent” of type “String”; the property is initialised with the value of the

last parameter "yes". This value can in fact be modified by the user, if s/he selects the "Class Attribute Specification" and clicks on the "CODE" tool.

4.4 Generating tables

A first generator will show the principle of how to create SQL scripts for setting up tables. Mapping object classes onto tables can be done according to several mapping strategies [5,9,14]. Frequently, each class `A` will result in a base table `tabA` that is able to hold all the atomic attributes of the class. Multi-valued attributes will be stored in an additional table with a foreign key referring back to the base table. Associations require foreign keys. The following tables are adequate for the introductory object model in Figure 1.

```
tabA (attr1 integer, attr2 varchar,
      primary key attr1);
tabB (b1 integer, b2 varchar,
      attr1 integer, primary key b1,
      foreign key attr1
      referencing tabA.attr1)
tabC (c1 integer, c2 float, attr1 integer,
      primary key c1, foreign key attr1
      referencing tabA.attr1)
```

Table `tabA` takes all the atomic attribute of class `A`. Using the value of the "Key" property, attribute `attr1` becomes a primary key: There cannot be two tuples in the tables with the same `attr1`-value. Both tables `tabB` (for class `B`) and `tabC` (for class `C`) possess a foreign key `attr1` that refers to table `tabA`. The foreign keys express the associations of `B` and `C` with `A`. Since table `tabC` can contain several entries with the same `attr1`-value, several `C`-objects are related to the one `A`-object in `tabA` that possesses this `attr1`-value.

The literature discusses several mappings for handling associations, multi-valued attributes, and subtype hierarchies. Hence, we want to refer to this work, e.g., [5,4,9].

A generator now produces a text file that contains corresponding CREATE TABLE statements. In fact, the repository of ROSE must be accessed to get the information about class names, attributes, keys etc. The basic principle is easy.

```
SUB Main()
 DIM classes AS ClassCollection
 DIM aClass  AS Class
 DIM attrs   AS AttributeCollection
 DIM anAttr  AS Attribute
 DIM tabName AS String
 OPEN "create.sql" FOR OUTPUT AS #1
 SET classes =
  RoseApp.CurrentModel.GetSelectedClasses()
 ' for any class marked in the ROSE model
 FOR i% = 1 TO classes.Count
   SET aClass = classes.GetAt(i%)
   tabname = "tab" + aClass.Name
   PRINT #1,"create table " + tabname + "("
   SET attrs = aClass.Attributes
   ' for any attribute of aClass
   FOR j% = 1 TO attrs.Count
     SET anAttr = attrs.GetAt(j%)
     IF anAttr.Type <> "int" AND
        anAttr.Type <> "float" AND ... THEN
        handleComplexAttr(anAttr)
     ELSE PRINT #1, anAttr.Name + " " +
                 mapType(anAttr.Type) + ","
     END IF
   NEXT j%
   handleAssociations(aClass)
   printPrimaryKey(aClass)
   PRINT #1, ");"
 NEXT i%
 CLOSE #1
END SUB
```

There is a loop over all classes that the user has selected in the current model. An inner loop gets information about the attributes of the current class. A function `mapType` is responsible for transforming C++ data types into SQL types. Complex attributes, i.e., those with a domain `Set<T>`, require special treatment by a function `handleComplexAttr`. Those attributes can be recognised by their type (`anAttr.Type`). Furthermore, functions `handleAssociations` and `printPrimary-Key` are needed to handle associations (e.g., by foreign keys) and to define primary keys. These can be implemented similarly. Associations are obtained by `aClass.Associations` yielding an `Association-Collection`. Having an `Association` instance, the roles (`Roles`, `Role1`, `Role2`), the cardinality of a role (`Cardinality`) etc. are accessible. Depending on the cardinality, a foreign key or a relationship table can be installed. Du to space limitations, we cannot show the full implementation. Particularly, subclass hierarchies are completely left out. But an adequate handling can be incorporated easily. [4] discusses several strategies to map hierarchies on tables with pros and cons, e.g., a vertical, a horizontal, a flag-based strategy and a complete materialisation. Functions `GetSubClasses/ GetSuperClasses` help to get the corresponding meta-information.

There are several strategies for handling complex attributes and subtype hierarchies. Instead of implementing one fix strategy, it is more flexible to let the user choose one. This can be done by means of the property "Strategy" (cf. Figure 3). Implementing a generator, we are then concerned with accessing those generator-specific properties. This can be done as follows for a given `anAttr`:

```
DIM aProp   AS Property
DIM propset AS PropertyCollection
...
SET propSet = anAttr.GetProperties
FOR k%=1 TO propSet.Count
   SET aProp = propSet.GetAt(k%)
   IF aProp.Name = "Strategy" THEN
      ... act according to aProp.Value ...
   END IF
NEXT k%
```

Thanks to the script language, the value of the generator can be increased with a graphical user interface. All the features of VisualBasic Script are usable. For example, boxes can be displayed asking for a database system (for which CREATE TABLE statements are to be generated), the names of output files can be chosen in pull-down menus, and so on.

4.5 Generating a ROSE model for the DBM classes

The persistence layer consists of several DBM classes, which all should become part of the ROSE model for documentation reasons. We assume that a category DBM has already been established in ROSE. The following script puts all the DBM classes in this category:

```
DIM theDiagram  AS ClassDiagram
DIM theDbmCat   AS Category
DIM allCats     AS CategoryCollection
...
' search DBM category
SET allCats =
  RoseApp.CurrentModel.GetAllCategories
FOR i%=1 TO allCats.Count
   IF allCats.getAt(i%).Name = "DBM" THEN
     SET theDbmCat = allCats.GetAt(i%)
   END IF
NEXT i%
' create new DBM public class diagram (PCD)
SET theDiagram = theDbmCat.AddClassDiagram
                  ("PCD_DBM_Interface")
' create a DBM class DBM_A for each class A
SET classes =
  RoseApp.CurrentModel.GetSelectedClasses
FOR i%=1 TO classes.Count
  SET aClass = classes.GetAt(i%)
  dbmClassName = "DBM_ " + aClass.Name
  SET dbmClass =
     theDbmCat.AddClass(dbmClassName)
  theDiagram.AddClass(dbmClass)
  ' establish a subclass relationship
  SET theInhRelation = aClass.AddInheritRel
                ("inherits from",dbmClass)
  theDiagram.AddClass(aClass)
  ' find-method (analogous for others)
  SET theOp = dbmClass.AddOperation
              ("find", aClass.Name + "*")
  SET theParam = theOp.AddParameter
                   ("key", <datatype>,1)
  theOp.AddProperty("OperationKind",
                   "Static")
NEXT i%
```

At first, a new public class diagram (PCD) is created. This diagram will contain the DBM classes. Then again, all the classes selected in the ROSE model are treated. For each class `aClass`, a DBM class is created in the PCD. All the methods such as `find` are added by invoking the function `AddOperation`, passing the name of the method and the return type. Calling `AddParameter`, the parameters are defined with a name, a data type, and a position. Each `Operation` instance has a pre-defined property "`OperationKind`" that can be set to make the method `static`. Similarly, a property "virtual" can be set for the `get` methods. The original class `aClass` is then inserted into the diagram to make it visible; the class is not duplicated by ROSE! The class is needed to establish the inheritance relationship between `aClass` and `dbmClass`.

Starting the generator creates the DBM class in ROSE automatically. It can be modified afterwards within ROSE, e.g., adding further `find`-methods.

4.6 Generating the persistence layer

The previous sections have already presented the basics of generation. A generator for the persistence layer can be implemented analogously. In principle, the code sketched out in Section 3 must be written into a file. All the generic parts such as class and attribute names must be filled in by accessing the repository. The following procedure generates the code for the `create`-method.

```
SUB printCreate(dbmClass AS Class,
                aClass AS Class)
 PRINT #1, "int " + dbmClass.Name +
           "::create()"
 PRINT #1, "{"
 PRINT #1, "   int failure = 0; "
 PRINT #1, "   RWDBConnection* cPtr =
     DBM_ConnectionMgr::getConnection();"
 PRINT #1, "   RWDBTable tab(tabName);"
 PRINT #1, "   RWDBInserter ins =
                  tab.inserter();"
 PRINT #1, "   try"
 PRINT #1, "   {"
 txt =     "      ins"
 FOR j% =1 TO aClass.Attributes
     txt = txt + " << anAttr.Name + "()"
 NEXT j%
 PRINT #1, txt + ";"
 PRINT #1, "      ins.execute(*cPtr); "
 PRINT #1, "      cache.insert
       (new Entry(Key<"+ aClass.Name +">" +
       "(" + handleKey() + "),this,NEW,1);"
 PRINT #1, "   }"
 PRINT #1, "   catch(const RWxmsg& x)"
 PRINT #1, "   {" DBM_ErrorHandler::
                            set(failure); }"
 PRINT #1, "   return failure; "
 PRINT #1, "}"
END SUB
```

Information about both classes is needed: `aClass` provides the attributes to build the `get` methods. The `dbmClass` is necessary since new methods might have been included in addition to the automatically generated ones, e.g., `find`-methods with other parameters.

5. Experiences

The presented approach has several advantages over commercial tools with regard to flexibility.

- The persistence layer can be brought into line with specific requirements and needs. The layer is customisable on several levels, in accordance to the

required functionality:

First, the modelling concepts can be restricted. For example, one can think of prohibiting subtyping in the database model, since this is difficult to handle and may cause a lack of performance. But the easier handling is dearly paid by the fact that subtype hierarchies cannot be made persistent as they are. Modelling is forced in those cases to introduce a simpler database model beside the original object model. This database model models the persistent part, simple enough to get handled. A transformation between both models becomes necessary.

Next, the interface can be enhanced or simplified, e.g., by introducing advanced transaction control or omitting explicit transactions, respectively. Sometimes, there is no need for explicit transaction; each method can be treated as a transaction of its own. In case of complex objects, which are fetched as a whole from the database [14], the transaction contains all those operations that are necessary to perform the method consistently.

We can also think of reducing the functionality of the persistence layer. The caching of objects is a possible candidate, if there is no danger of synchronisation.

Hence, the approach can be very simple, but it can also support a full implementation of the ODMG standard for ODBSs [6] on the other extreme. The more complex the modelling concepts and the interface are, the more complex will be the implementation, and consequently the generator.

- Since *we* implement the persistence layer, we are able to understand how it works. This gives us the flexibility to modify the code afterwards, e.g., for tuning the layer. In contrast to commercial tools, the layer is no longer a black box!
- The approach is extensible to access existing tables. This requires a data re-engineering in the generator for producing the DBM classes, as now reverse strategies [3] must be specified.

We are using the approach with Siemens AG in a huge project in the field of telecommunication. The project possesses some important characteristics, which form a fertile soil for the generative approach:

(a) There are several work packages implementing the overall system functionality. The developers are mostly unfamiliar with relational database technology, with table design and SQL. That is the reason why a work package DBM ("Database Management") has been installed with the goal to centralise database competence and to release other work packages from database aspects. Hence, DBM can take the part of implementing generators. Each work package then gets a very comfortable, easy-to-use interface for handling objects. It is obvious that application developers do not need to care about any database aspects except invoking DBM methods. This accelerates the productivity of programmers.

(b) The schematic, model-based persistence approach makes sure that there is a strong de-coupling of application programming and persistence. The database aspects are closely related to an object model and thus easy to understand. Developing and generating the persistence layer can be done in parallel to the real application development. This reduces the development times of working packages.

(c) There are more than 100 persistent classes in the system. Consequently, the effort for implementing a generator is modest on a per-class calculation.

(d) Some work packages have very specific requirements on the database. For example, some do not need transactions as each method is consistent in itself by means of a transaction per method mechanism. Others require very specific queries in the sense of CMIS filters ("Common Management Information Service"), which are very popular in telecommunication standards. We could directly support corresponding query classes in our layer. Another group of work packages performs an internal caching of data to speed up access in real-time. They read the data of a class completely in an internal cache. Using a commercial tool would have doubled caching, leading to a loss of performance.

To satisfy these various needs, several persistence approaches have been developed, grouping the required functionality. This emphasises the good adaptability of the generative approach.

(e) Database competence is also necessary as there are several applications with hard performance requirements. Hence, it is inevitable to have potential for tuning database tables and accesses. However, the interface should be stable for applications.

Consequently, the advantages of our approach come to light. Do commercial persistence tools support the points (a) to (c) to some degree, they lack of flexibility with respect to (d) and (e). We are reporting on experiences we made for point (e).

We had a work packages that caches more than 100000 complex objects during start-up of the system in order to guarantee accesses in main memory speed. This took about 3 hours using the generated implementation; firmly too long to be acceptable. Indeed, commercial tools would not have behaved better. Consequently, tuning the layer was necessary. We re-designed the tables (not the database model), and reorganised the accesses and their order of execution. The interface gave us enough potential for tuning database and accesses without affecting the application. That is, the interface to the database was as before, but the implementation of the interface changed completely. Tuning the system, i.e., re-implementing the layer manually, finally brought up results in the area of 3 minutes. The essential advantage in this respect was that the code is understandable and thus modifiable. Certainly, this action breaks the round-trip engineering as any modification of the model must be either adapted in the current implementation, or the

generated code must again be re-implemented. But having such gains in performance does worth it anyway.

The approach is obviously suited if the amount of work for implementing the generators is not too high. We spent four person weeks for designing the various persistence layers in accordance to requirements. The effort for developing the generators tool took about three person weeks. Certainly, these numbers depend on the skill of the people who design and implement.

6. Conclusions

In this paper, we demonstrated how to generate object-oriented persistence layers for relational database systems with modest effort. The approach consists of integrating code generators in Rational ROSE. Taking a ROSE model, a persistence layer is automatically generated. The usage is easy as persistence can be modelled as part of a ROSE model. The paper discussed in detail how such a generator can be implemented.

In contrast to commercial tools that also provide or generate persistence layers, our approach has the advantage that it is highly customisable: The persistence layer can be tailored to the real needs of an application.

In spite of causing expenses, the approach is a good alternative to commercial tools if

- special requirements are to be fulfilled (e.g., advanced query mechanisms)
- the functionality and the overhead of the persistence layer must be scaleable
- a high performance of database accesses is required demanding for extensive tuning
- the number of persistent classes is large.

Future work is dedicated to enhance the generator to allow for data re-engineering, i.e., accessing existing tables in a relational database. Here, the focus lies on building real object-oriented views for relational data. Hence, it is important to have flexible strategies to bridge the gap between an object model of an application and the tables. Otherwise, an application would be forced to adjust its object model to what is achievable by remodelling, which might be not enough. We developed an approach that is based on powerful syntactical specification language [4]. This language allows one to remodel tables in several ways. It has the drawback that remodelling has to be done by writing down specifications. We feel the need for a graphical support [5]. Consequently, we think of integrating the powerful re-engineering strategies in ROSE.

References

[1] Legacy Systems. Special Issue of IEEE Software 12(1), 1995

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M.Stal: Pattern-Oriented Software Architecture - A System of Patterns. John-Wiley & Sons, 1996

[3] W. Premerlani, M. Blaha: An Approach for Reverse Engineering of Relational Databases. Communications of the ACM 37(5), May 1994

[4] U. Hohenstein: Bridging the Gap between C++ and Relational Databases. 10. European Conference on Object-Oriented Programming (ECOOP'96), Linz 1996

[5] U. Hohenstein, C. Körner: A Graphical Tool for Specifying Semantic Enrichment of Relational Databases. In: 6th IFIP WG 2.6 Working Group on Data Semantics (DS-6) "Semantics of Database Applications", Atlanta (Georgia) 1995

[6] R. Cattell, D. Barry (eds.): The Object Database Standard: ODMG2.0. Morgan-Kaufmann Publishers, San Mateo (CA) 1997

[7] G. Kappel, S. Preishuber, E.Pröll, S. Rausch-Schott, W. Retschitzegger, R. Wagner, C. Gierlinger: COMan - Coexistence of Object-Oriented and Relational Technology. In: Proc. of 13th Int. Conf on Entity-Relationship Approach (ER'94) - Business Modelling and Re-Engineering, Manchester 1994

[8] T. Salo, J. Hill, K. Williams: Scalable Object-Persistence Frameworks. Journal of Object-Oriented Programming, Nov/Dec. 1998

[9] W. Keller: Object/Relational Access Layers - A Road-map, Missing Links and More Patterns. EuroPLoP 1998

[10] H.-E. Erikson, M. Penker: UML Toolkit. John-Wiley & Sons, Inc., 1998

[11] U. Hohenstein, R. Lauffer, P. Weikert: Object-Oriented Database Systems: How Much SQL Do They Understand? In D. Karagiannis (ed.): 5th Int. Conf. on Data and Expert Systems Application (DEXA) 1994, Athens (Greece)

[12] K. Brown, B. Whitenack: Crossing Chasm, A Pattern Language for Object-RDBS Integration. In J. Vlissides, J. Coplien, N. Kerth (eds.): Proc. PLoP 1995, Addison-Wesley 1996

[13] P. Heinckiens: Building Scaleable Database Applications. Addison-Wesley 1998

[14] W. Keller, W. Coldewey: Relational Database Access Layers - A Pattern Language. In Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences, Washington University, Department of Compuer Science, Technical Report WUCS 97-07, Feb. 1997

[15] J. Coldewey, W. Keller: Multilayer Class. In collected papers from the PLoP'96 and EuroPLop'96 Conferences, Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995

[17] W. Keller, J. Coldewey: Accessing Relational Databases. In: R. Martin, D. Riehle, F. Buschmann (eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998

[18] G. Pernul, H. Hasenauer: Combining Reverse with Forward Engineering - A Step forward to Solve the Legacy System Problem. In: Int. Conf. on Database and Expert Systems Applications, 1995

Development of a Visual Requirements Validation Tool

Paul W. Parry and Mehmet B. Özcan

Sheffield Hallam University,
School of Computing and Management Sciences,
Sheffield S1 1WB,
U.K.
Email: P.W.Parry{M.B.Ozcan}@shu.ac.uk

Abstract

This paper describes work associated with the development of a requirements visualisation tool. Our work builds on the strength of existing executable formal specification systems in that a software model can be described using an executable specification language as the basis for the construction of a prototype system. In addition, it employs a visualisation tool based on graphical dynamic animations so as to facilitate a flexible and customisable user validation approach that distances the visual representation away from formality. In this context, this paper firstly details the issues and principles associated with visual animation of an executable specification system that underpin and influence the development of our software tool. Secondly, it describes a generic mechanism that facilitates visualisation of specifications. Thirdly, the architecture design of our tool is outlined, together with the design decisions that were made. Finally, our experience as a result of exploitation of our tool will be highlighted.

Keywords: Requirements, Visualisation, Animation, Prototyping, Formal Specifications, Validation

1. Introduction

Requirements validation through feedback with users is of paramount importance in producing a high quality software requirements specification document. We argue that an eclectic approach that offers an effective combination of formalism and pragmatism may encourage software developers to move towards software engineering practices necessary for software systems that satisfy user requirements. This approach involves the use of an executable formal specification for the construction of software prototypes, which can be used to validate software requirements with users at an early stage through feedback.

However, executable formal specifications are often ineffective in the user validation process [1]. An executable formal specification has traditionally been used as an effective tool specifically for developer validation; that is the developer can, via specification execution, either individually or in a peer review format, explore the consequences of the specification. However, its use in user validation is often not user orientated. This is exacerbated by the fact that the execution behaviour of a prototype may not always be comprehensible to users due to the abstract nature of notations supported by these specification languages and by the way their execution behaviour is presented to users. This may in turn render the requirements validation process ineffective.

Use of requirements visualisation as an animation technique has the potential to facilitate the communication between the stakeholders. The use of a visual technology supported by a human-centred process to elicit and validate software requirements allows complex concepts and information to be presented in ways that are easier to understand. It is evident from the literature that human processing can clearly benefit from pictorial information and growing body of research work continues to support this view [2]. Application of visualisation techniques to requirements engineering requires the use of a process with an appropriate tool support which maps abstract formal representations to concrete representations that can readily be understood by users. Unlike a programming language translator, the translation between the two representations should be bi-directional so that users can directly interact with a software system.

Literature is relatively sparse about software systems using a visual technology to animate a formal specification. Cooling [3] attempted to visualise requirements specifications based upon VDM. To do this a separate 'script' was developed from VDM that derives the visualisation and animation. The script itself contains both a model of the system and the visualisation detail. Evans [4] described a system in which Coloured Petri Nets were integrated with the formal specification language Z to specify and visualise concurrent systems. The Teamwork/EDS [5] system was designed to execute

and visualise real-time structured analysis specifications based on SA/RT graphical notation. Mosel-Meta-Frame [6] is an approach that provides simulation and visualisation of hardware circuits. The system uses directed graphs as a visual formalism to support analysis and verification. Visualising the behaviour of telecommunications systems is the focus of the approach described in [7]. Coloured-Petri-Nets were used as the specification formalism, but are augmented with a more abstract visual representation to support user interaction. The CheckOff-M environment [8] facilitates the verification of application specific integrated circuits through visualising their behaviour. This approach employs the symbolic timing diagram as a visual formalism to specify and depict the dynamic behaviour of model of integrated circuits.

Although the above approaches offer powerful visualisation capabilities, the resulting visualisations are still developer oriented in that they simply shift the issue of incomprehensibility from the formal and textual notation to the potentially cryptic visual form. Our work is consistent with the systems described above. It builds on the strength of the executable formal specification systems in that the system model can be described using an executable specification language as the basis for the construction of a prototype system. In addition, it employs a visualisation tool based on graphical dynamic animations so as to facilitate a flexible and customisable validation approach that distances the visual representation away from formality. The remainder of this paper details the development of such a visualisation tool. Firstly, the issues and principles associated with visual animation of a specification system that underpin and influence the development of our tool will be outlined. Secondly, specifics of our tool in terms of its architecture and its components descriptions will be described, together with design decisions that were made. Finally, our experience as a result of exploitation of our tool will be highlighted.

2. Issues and principles

This section describes a set of general issues and general principles that are pertinent to the development of a visualisation tool to support requirements validation with users. These issues and principles associated with them form the basis of the detailed requirements of a tool described in this paper. They have been derived from a synthesis of the analysis of the existing tools identified in the previous section and general software engineering principles and software quality factors are detailed next: It should be noted that these do not preclude other very important software quality goals such as reliability, robustness, etc.

Provision for different types of representations. The type of representation employed in the visualisation process may have two attributes. Direct representation types provide a realistic presentation of an object or system under consideration, which is close to its real-world counterpart, such as a video-clip or a photographic image. This type of representation is best used when trying to communicate highly detailed information where a diagram could possibly hide some of the content. On the other hand, abstract representations provide an abstract image of the underlying information, and thus provide an appropriate mechanism to emphasise certain details while hiding others. Examples of abstract representations, are charts, graphs or diagrams that show relationships between information. The exact ratio of direct representations to abstract representations is ultimately a choice for the developers and users, but we suggest that, in practice, it could be dependent upon the nature of a software system being validated and the sophistication of users who validate it. From our perspective, a software tool should for requirements visualisation should have the potential to provide support for both types of representations for effective visualisation and validation.

Provision for visual representations familiar to the users domain. From the comprehension point of view, merely seeing an image of an object is not enough for comprehension. To be useful, the image itself must be presented in some potentially meaningful way. In order to understand what facilitates the ease of comprehension of a representation the following issues need to be examined: (a) general characteristics of images that make them amenable to comprehension; (b) the abilities of humans to understand visual representations; (c) the cognitive processes that may occur in the brain to make this understanding possible. Note that proper treatment of these issues is beyond the scope of this paper. From our perspective, it suffices to state that in order to facilitate and enhance users' comprehension, a software tool should have the potential and produce representations that are expressed using visual cues from the users' own domain, which can be tailored to match their sophistication. A software tool should also support a wide range visual techniques (such as multimedia support) to achieve this.

Provision for animation. This issue is concerned with capturing the process of depicting dynamic and static behaviour and may be addressed in terms of a generic animation model. The level of animation has two attributes. *Static* refers to an unchanging still image, such as diagrammatic notations. This form of representation is useful when attempting to convey relationships between objects, and for analysing the structural properties of systems, processes or data. *Dynamic* implies a continuously changing set of images that correspond to some execution process undergoing successive change. This is useful when depicting, analysing, and understanding dynamic processes. It is widely accepted that effective validation is performed when the users observe a dynamic representation of the system's requirements [9]. In this context, the notion of "animation" (in the graphical sense) becomes a prominent issue. In order to reflect the dynamic nature of a process or activity, an appropriate representation should use a

dynamic component, i.e. the representation should appear to change so as to present the change in the execution behaviour of the prototype. In contrast, should a particular aspect of a prototype's execution behaviour remains constant, then this should also be reflected in the representation. Hence, a software tool should provide support for the visualisation of both the static and dynamic aspects of a prototype.

Provision for maintaining the integrity of visualisation. Integration of informal and formal techniques brings its own problems. A notable one is that use of visualisation as an informal technique has an undesirable side effect in that ambiguity and misinterpretation may be introduced into the validation process. This is due to the possibility of applying arbitrary visualisations in inappropriate contexts. The creation of a visualisation is a subjective process. It is not possible to produce formal semantics for informal visualisations represented by video-clips and photographic images, etc, and there are no defined semantic rules or translations that can force a visual representation to possess one and only one meaning with respect to elements in a specification. From a more philosophical perspective, it could be possible to argue or reason about the semantics of a particular representation as opposed to another, as perceived by an individual, but this would be beyond the realms of software engineering. This issue does not necessarily invalidate the usefulness of the application of visualisation to requirements validation. However, it has to be addressed, at least partially, to minimise its potential negative impact upon the outcome of the validation process.

Provision for maintainability. During the requirements validation process, initial requirements of a software system undergo a number of changes as a result of users' feedback. These changes will inevitably alter the structure and content of an executable specification on which a prototype is based. Consequently, it is inevitable that visual representations associated with the specification should also be changed. Considering the number of iterations during user validation and the importance of timeliness in a prototyping activity, it is important that a software tool should support rapidity so as not to impede on productivity and that it should be flexible enough to enable modifications to the existing visualisations to be carried out without undue difficulty. In order for a software tool to be flexible, it should provide comprehensive facilities for creating and editing visual representations with ease. The provision of such facilities necessitates the consideration of appropriate interaction styles.

Provision for re-use of visual representations. Although reusability is arguably difficult to achieve and that examples of software reuse in practice are rather rare, it is clearly an important technique for reducing software production costs. The notion of reusability is highly relevant to requirements visualisation since during the process of creating and editing visual representations, a large number identical visual images are often shared by different applications within a domain (i.e. vertical reuse, such as images of books in different library applications) as well as by applications within different domains (i.e. horizontal reuse, such as images of people, bank notes, machines, etc., in library or ATM applications). Hence, a software tool that provides support for visual component reuse can be beneficial to this process. Equally importantly, a software tool should also provide a mechanism for the effective retrieval of components within a reuse repository to facilitate productivity and rapidity during the process of creating and editing visual representations.

Provision for interoperability. For the purposes of our research, it is assumed that a software tool for the visualisation of requirements will work in conjunction with an executable specification system that exists as a separate entity. Hence, interoperability becomes prominent issue. The interaction is necessary for a visualisation tool to intercept and subsequently visualise the results of computations associated with a specification, thus necessitating some form of a software communication link. At a low level of abstraction, this link needs to take into account of any underlying operating system and environment in which the two software tools co-exists as well as taking into account of necessary communication protocols through which the communication takes place. At a higher level of abstraction, it is necessary to define the behaviour of how visual representations can realistically reflect the elements and computations associated with a specification and how the communication link can facilitate this.

3. A generic visualisation mechanism

Our visualisation technology allows software developers, or visualisers, to choose an appropriate representation for data elements (i.e. factors in an expression) in a specification, and the results of applying an operation to these data elements (i.e. the result of executing an expression), and create dynamic and/or static animations. It provides a generic visualisation model to capture and describe the process of visualising the static and dynamic behaviour of a specification. It is based upon the notion of a state, and is described in terms of visualising the state of a system before the execution of a portion of a specification under investigation, and the modified state (if execution changes the state) after the execution.

In this context, visualisation of the formal specification is a composition of appearance (i.e. direct or abstract) and the corresponding dynamic components. This visualisation is then related to the specification concerned so that during its execution, the visualisation tool will be able to 'play' its corresponding visually animated representation for user validation. To achieve this, visualisations are attributed with an identifier. Each expression to be visualised, i.e. ones that modify the

system state, can be augmented with a visual identifier to facilitate its visualisation. Note that incorporating a visual reference to the specification does not necessarily bias it towards a particular implementation since the visualisation process takes place after the specification has been written. The resulting specification is then processed by the run-time visualisation engine of our tool.

4. Tool description

This section describes the design of our visualisation tool. The design is presented in terms of the capabilities of the tool and how the key principles that have been introduced in the previous section are fulfilled. In addition, the design description is augmented with the design alternatives that were considered during the design process and the design decisions that were ultimately made. The organisation of this section is as follows: Firstly, the architecture of the tool will be given. Secondly, an integrity mechanism that maintains the consistency between a specification and its corresponding visualisation will be described. Finally, an integration strategy will be outlined to describe how an executable specification system can be integrated within our tool to provide appropriate visualisations.

4.1 The architecture

As shown in Figure 1, the software tool is a collection of interrelated software components that comprise of:

i). appearance component editor
ii). dynamic component editor
iii). visualisation editor
iv). visual component repository
v). visualisation engine

Figure 1 describes the relationships between these components and the broader context of how the tool is used in conjunction with other software tools to achieve the objective of requirements visualisation. The direction of arrows indicates that the lower component provides support for the services provided by the upper component. The main rationale for designing the tool in terms of a number of sub-components is now given.

From our perspective, a visualisation is essentially made up of two components as detailed in the previous section. These are: appearance (i.e. direct or abstract) and behaviour (i.e. static or dynamic). The responsibility for creating these components was assigned to individual software components (i.e. appearance component editor and dynamic component editor) to support separation of concerns. Furthermore, in order to compose the products of these tools into a visualisation, which can later be associated with a specification for animation, a third component was developed (i.e. visualisation editor). In addition, a repository was developed to support a classification of visualisations and its components (i.e. visual component repository). This classification has the potential to facilitate component reuse and increase

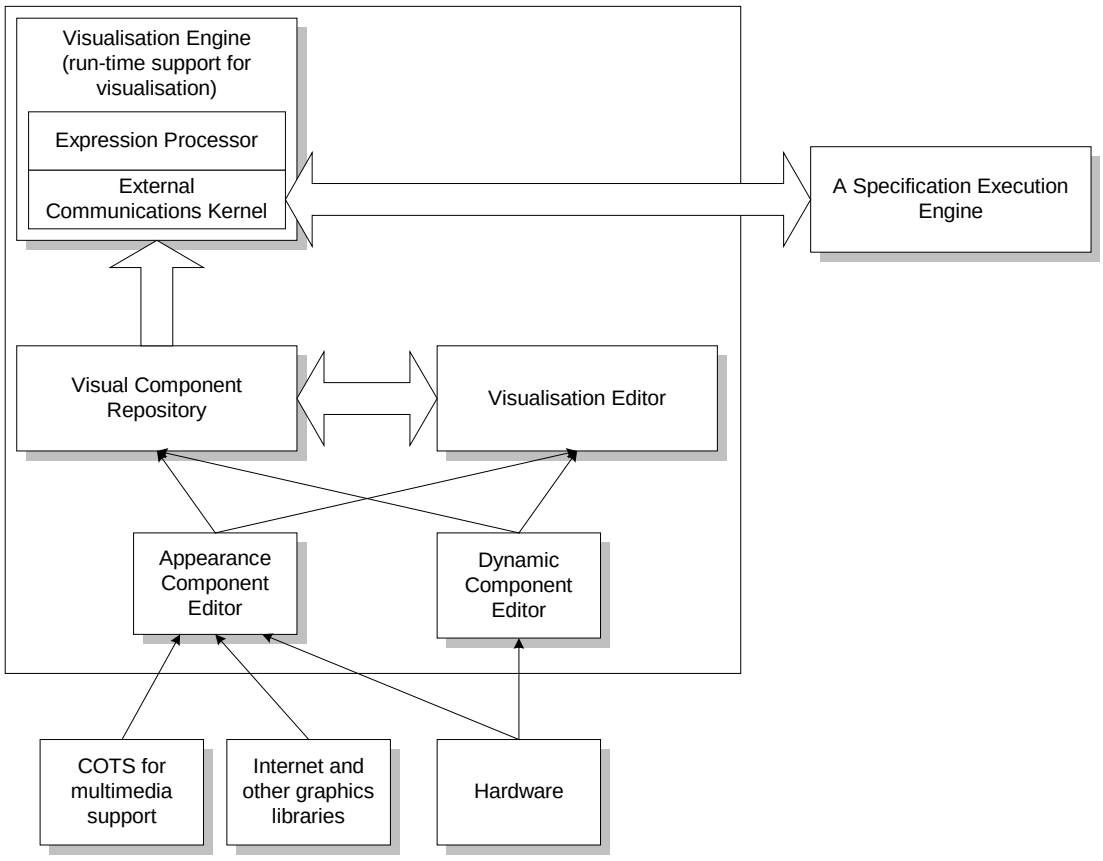


Figure 1. The components of the visualisation tool and their relationships.

productivity during the process of constructing visualisations. Finally, a software component that acts as an interface between a specification execution system and the tool's other components was developed (i.e. visualisation engine). This provides support for establishing communication, data exchange and co-ordination of specification execution and its visualisation, thereby providing an abstraction to a specification execution system. Details of these components are elaborated next.

The Appearance Component Editor

This component was required to provide a straightforward means of expressing appearances that should comprise of a range of possible visual cues, colours, and textual components. Resulting appearance components must be given a unique identifier by a creator/designer. The system enforces the uniqueness of this identifier. Equally importantly, this editor supports a type domain to associate appearances with appropriate types to maintain integrity. The role of appearance identifier and appearance types will be elaborated in Section 4.2.

A number of potential formats for this editor were postulated, ranging from text-based command line forms to fully interactive direct-manipulation style formats. The final design decision was based upon the concept of enabling a visual representation to be expressed in way that promotes correspondence between the developer's/designer's conceptual view of the desired representation and the actual view while editing. To this end, it was decided to use 'graphics to describe graphics' as an interaction and editing style, as this method enables the appearance to be viewed and evaluated immediately. The final result is an editor that enables an appearance's individual visual cues to be expressed separately, in a graphical manner, whilst at the same time giving the creator an opportunity to see the appearance as a whole.

The editor is capable at present of supporting basic geometric shapes, text elements, photographic images, and icons. These can be, if required, provided for by an importation mechanism, whereby other commercial-off-the-shelf (COTS) software packages, such as professional image editing and manipulation tools, are used to capture and enhance images. These are saved in files and imported into the appearance editor as separate visual cues. The design of the appearance editor makes provision for importing image files that are of the popular 'bmp', 'gif', and 'jpg' formats. Other image formats could be accommodated by providing a suitable translator to read and convert the files into a form that is appropriate for the appearance editor.

The Dynamic-Component Editor

This tool is concerned with providing support for the dynamic components of a visual representation that may or may not have already been created. Hence, an appearance and its animation behaviour are independent, in that dynamic components are polymorphic and can be applied to any appearance.

This particular editor must facilitate a range of dynamism, stretching from the static view to the graphically animated form. To represent static visualisations, this editor can be used to describe the on-screen locations of appearances. At run-time, an appearance can be rendered at the location given. To represent dynamic visual forms, the tool allows the path of an appearance to be described in terms of a sequence of nodes, and at run-time, the appearance can be animated smoothly along this path to depict the movement of objects in a scenario. Motion-components must be attributed with a unique identifier. The role of dynamic-component identifier will be elaborated in Section 4.2.

The design of this editor is based upon a direct-manipulation style user interface. This enables locations to be specified and paths to be described by creating and dragging nodes on a 'canvas' – the canvas representing the screen onto which appearances will be rendered. Figure 2 shows the conceptual design of this interface, and illustrates the placement of the path and its associated nodes.

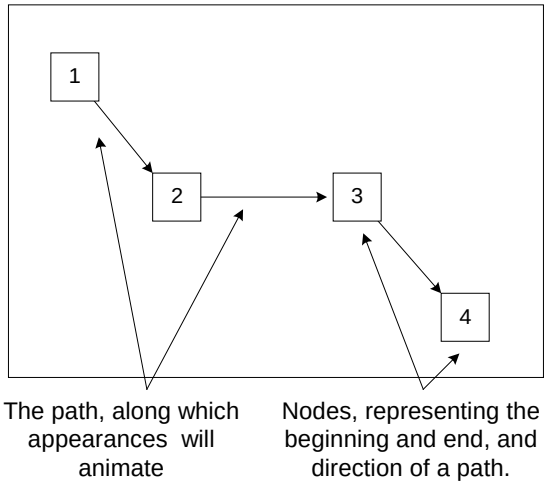


Figure 2. The conceptual design of the motion component editor user interface.

The Visualisation Editor

This component of the visualisation tool is responsible for combining together individual appearance and dynamic components of a representation, that were developed using the tools already described to form visualisations. This is achieved by making references to identifiers that refer to appropriate appearance and dynamic components. Similar to the appearance and dynamic component editors, a complete visualisation containing its appearances and dynamic components are must be given a unique identifier, which enables it to be referenced by an expression within a specification.

One of the key design decisions made when formulating the visualisation editor was to develop a type domain for the creation of a visualisation. The type domain as a whole incorporates the structure of expressions that can be found in specification languages. This domain could easily be populated with types to match the type of expressions in a new specification language. Using a syntax-directed editor, the structure of a visualisation is constrained by the type of expression in a specification language to be visualised.

Component Repository

As shown in Figure 3, a repository to store the visual components that are created during the visualisation process is also provided by the tool. This repository has separate 'containers' in which to store the different categories of components that pertain to the visualisation of a requirements model, i.e. the entire set of appearances, motions, and expression-level visualisations.

In addition to providing storage for these basic components, the repository offers a classification mechanism through the use of 'applications'. These can be likened to directories in a file system, whereby components that are related, by virtue of being applied to a particular validation project, can be partitioned. Applications must also be given a unique identifier, just as other visualisation components, and classification of visualisation components is performed by attributing them with the appropriate application identifier. The role of application identifier will be elaborated in Section 4.2.

The Visualisation Engine - A Vehicle for Tool Integration

This component, which is essentially a run-time system, facilitates visualisation of expressions within a specification. It is made up of two sub-components, as shown in Figure 1. These are the expression processor and the communication kernel respectively.

The expression processor receives the string representing a given specification. It processes the string by traversing it in a recursive manner. To achieve this, for each expression, it communicates with the specification execution engine to request that it be evaluated. In addition, if the expression is associated with a visualisation identifier, then the components of the expression, and the result, are visualised by retrieving the corresponding visualisation from the repository.

To facilitate the communication between the two tools, a design decision was made to implement the link as a client-server architecture to provide support for a wide range of applications, ranging from hosting both visualisation tool and a specification execution system in the same operating environment, to a distributed architecture whereby the tools could communicate via the IP protocol across a wide area network.

At present, the communication link relies upon the DDE protocol to facilitate data transfer within the same operating environment. In order to integrate a particular specification execution system, the following procedure needs to be followed. At the specification language end, the syntax of the language needs to be modified slightly to incorporate visualisation identifiers. At the visualisation-engine end, an expression processor needs

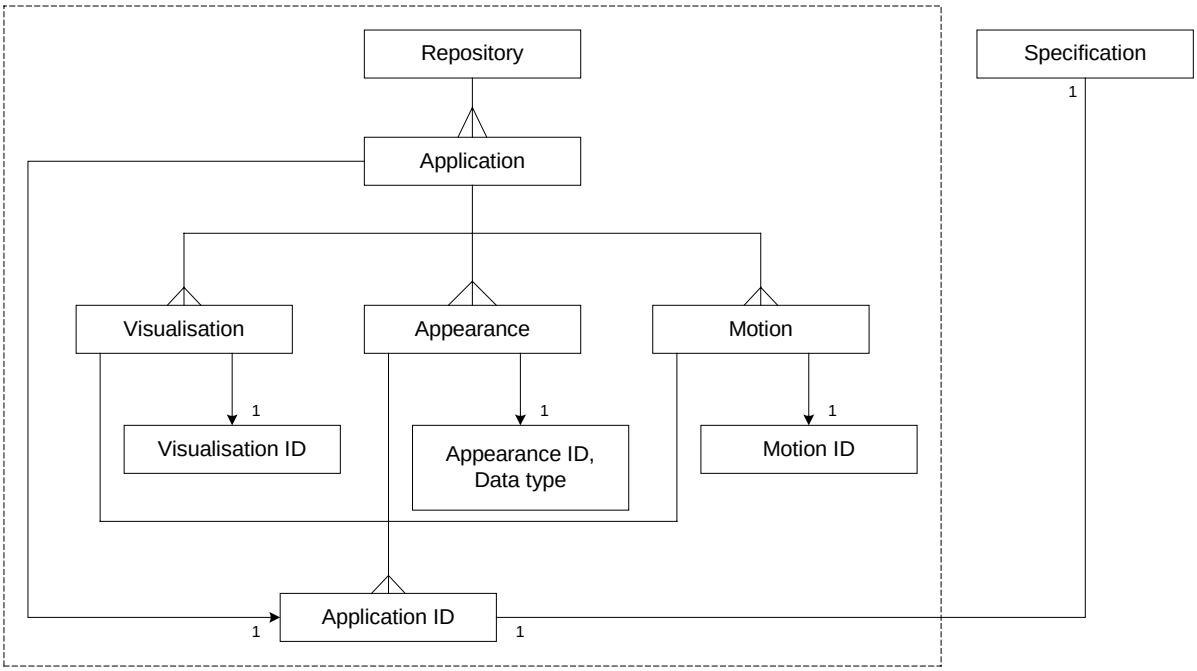


Figure 3. The entities and relationships that exist in the visual component repository and between specifications.

to be implemented to accommodate the syntax and structure of the notation. This can be manifested as a plug-in component to the visualisation engine. In addition, the communication protocol must be implemented (if it does not already exist) to incorporate any protocol supported by the specification execution system.

4.2 An integrity mechanism

Referring back to Section 3, the association between an expression in a specification and the corresponding visualisation via the visualisation identifier mechanism we create a disengagement mechanism between the visualisation and specification. This has the benefit that our visualisation tool can be applied, theoretically, to any executable specification environment that bases its execution upon expressions. However, the main drawback of this approach is that it is possible to create a semantic disengagement between the specification and visualisation. In order to partially address the issue raised in Section 2 associated with integrity, a design decision was made to implement a constraining mechanism to restrict the application of visualisations to inappropriate contexts. This comprises of two complementary aspects.

First, a constraining mechanism was developed, based upon the use of applications (described in Section 3.2). A developer or senior user can partition relevant appearance-, dynamic-, and visualisation components into a suitable application, in order to specify a context in which the components can be used by future developers or users. This places limits on the choice of components that can be applied in a particular validation project. A prerequisite is that the given specification incorporates an attribute that signifies the application.

Second, a type system was developed to constrain the association between the factors of an expression and visual representations (i.e. appearances). A prerequisite is that the given executable specification language should support a type mechanism. If not, it may be necessary to extend the syntax if the language to support one, which would also require modifications to the visualisation engine. Appearances are attributed with a type, that corresponds the type domain of the specification language, by the appearance component editor. Upon creating an appearance, a developer/senior user can attach an appropriate type. This appearance can then only be applied to factors in an expression that are of the same type.

5. Exploitation

Our visualisation tool has been exploited to support a formal specification animation environment called ZAL (or Z Animation in LISP) [10]. ZAL is a LISP-based animation environment, which provides extensions to LISP to form an animation environment. A close correspondence between the Z notation and the ZAL notation is preserved. The hallmark of ZAL is to view it as a generic animator which models Z constructs rather than any particular specification; the subset of Z that can be animated is that for which equivalent constructs have been developed in ZAL. Our tool was used to provide a generic visualisation model to capture the process of visualising dynamic behaviour of a ZAL specification. As far as the ZAL system is concerned, the model is based upon the notion of a state and is described in terms of visualising the present state of a system before the execution of a ZAL specification and the modified state of the system after the execution of the specification. Thus, a visualisation of a ZAL specification involves the fabrication of a composition containing all the representations and the corresponding dynamic components for the states of the system.

A number of standard examples (such as a book lending library system [11] and an automated teller system [12]) and a very large case study of a real-time safety-critical application (a water level monitoring system) [13] have been specified, visualised and validated. Preliminary results of this work was reported in [14] and [15]. In addition, together with the ZAL system, our visualisation tool was made available to undergraduate students and a number of MSc students for use and evaluation, but as yet has not been used on a real industrial project.

For effective user validation, our tool is at present supported by a usage oriented process to capture both functional requirements (i.e. what a software system should do) and software usage aspects (i.e. how the system should behave from the users' point of view). It should, however, be noted that this does not preclude the use of other processes and methods since our tool is not process specific. With the aid of our tool, the users can judge and comment on a scenario being animated within the context of a use case with which it is associated. A visual scenario allows different possible choices to be investigated in the context of an entire system. More importantly, visualisations of scenarios can effectively be used as a catalyst to provoke debate to help elicit additional knowledge to evolve the requirements, thereby contributing to the overall specification rather than just to the clarification and validation of a set of requirements. Figure 4 illustrates the visualisation of a water level monitoring system (WLMS) [15]. The scenario demonstrates that if the water level drops below the designated safety limits then the alarm will sound. The values on this visualisation, such as "Allok" (i.e. all devices are operating properly), "Operating" (i.e. the WLMS is operating normally) and the water level in the main reservoir, are the inputs to the ZAL specification representing the WLMS system, and the "Alarm is Audible" warning message is the result of computation performed by the ZAL execution engine.

Changes in input values are at present carried out at the specification level to give the users an opportunity to explore different possibilities (such as what if the water level is within the designated safety limits, but the device that monitors the water level has failed?) to validate such

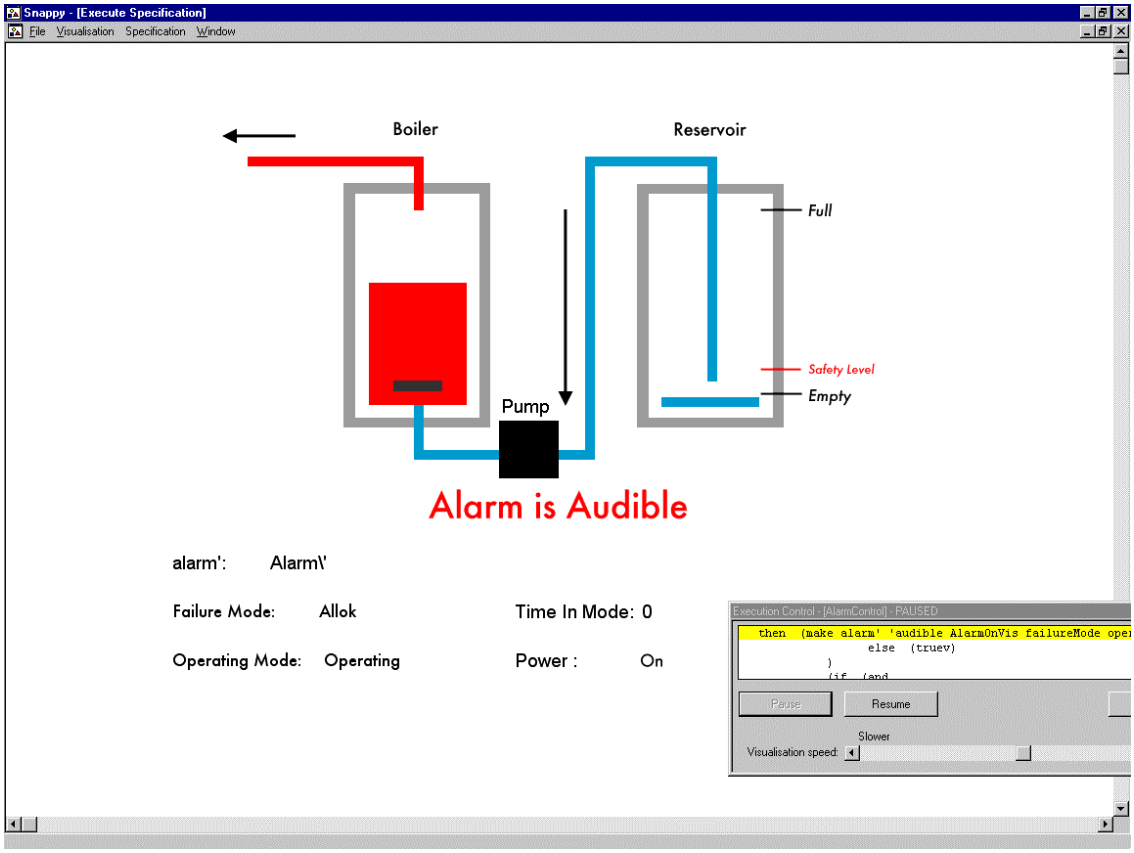


Figure 4: Visualisation of a water level monitoring system.

scenarios and possibly uncover situations that have not been previously thought of so that the requirements can be evolved.

6. Concluding discussion

This paper described an eclectic approach that involves the use of an executable formal specification for the construction of prototypes to validate software requirements with the users. Our research is part of an ongoing effort to move towards quality requirements. Our experience suggests that employing formal notations during the software validation process cannot always be effective due to the users' inability to understand the formal specifications and their execution behaviour. We have therefore advocated the integration of visualisation techniques to the formal specification based software development process to make formal notations more accessible to the novice user stakeholders.

Initial evaluations of our tool proved successful. The tool not only help validate existing requirements, but also it was effective in eliciting information to identify 'key' or interesting scenarios to stimulate discussion with the users. However, there are a number of shortcomings that need to be addressed for our tool to be used in an industrial placement. First, we found that the component repository was not effective enough to facilitate immediate reuse and thus improve productivity. To address this problem, the repository needs to be populated with an adequate number of pre-developed visualisations. Second, the appearance component editor needs to be supported by a wider range of media types, such as aural and video capabilities. We are currently investigating the incorporation of hypertext facilities and how this could be instrumental in the user validation process. Last but not least, the architecture of the visualisation engine needs to be supported by a wider range of plug-in components to accommodate common industry standard specification execution systems.

References

[1] M. B. Özcan, "Use of Executable Formal Specifications in User Validation", Software-Practice and Experience, 28(13), 1359-1386, (1998).

[2] G. Roman and K. C. Cox, "A Taxonomy of Program Visualisation Systems", IEEE Computer, 26(12), 11-24, (1993).

[3] J.E. Cooling, T.S. Hughes, "Making Formal Specifications Accessible Through the Use of Animation Prototyping", Microprocessors and Microsystems, 18(7), 285-392, (1994).

[4] A.S. Evans, “Visualising Concurrent Z Specifications”, Z User Workshop - Proceedings 8th Z User Meeting (Editors J.P. Bowen, J.A. Hall) 29-30 June 1994, Cambridge UK, Springer-Verlag, p269-281, ISBN 3 540 19884 9.

[5] R. Blumofe, “Executing Real-Time Structured Analysis Specifications”, ACM Sigsoft Software Engineering Notes, 13(3), 32-40, (1988).

[6] Tiziana Margaria, Volker Braun, “Formal Methods and Customised Visualisation: A Fruitful Symbiosis”, Lectures Notes in Computer Science, 1385, 142-157, Springer-Verlag, (1998).

[7] Carla Capellmann, Soren Christiensen, Uwe Herzog, “Visualising the Behaviour of Intelligent Networks”, Lectures Notes in Computer Science, 1385, 142-157, Springer-Verlag, (1998).

[8] Rainer Sclor, Bernhard Josko, Dieter Wirth, “Using a Visual Formalism for Design Verification in Industrial Environments”, Lectures Notes in Computer Science, 1385, 142-157, Springer-Verlag, (1998).

[9] A. Tsalgatidou, "Modelling and Animating Information Systems Dynamics", Information Processing Letters, No 36, 123-127, (1990).

[10] J. Siddiqi, I. Morrey, C. Roast and M. B. Ozcan, “Towards Quality Requirements via Animated Formal Specifications”, Annals of Software Engineering, 3, 131-155, (1997).

[11] D. Andrews and D. Ince, Practical Formal Methods with VDM, McGraw-Hill, 1991.

[12] I. Sommerville, Software Engineering, 4th Edition, Addison-Wesley, 1992.

[13] A. J. van Schouwen, “The A-7 Requirements Model: Re-examination for Real Time Systems and an Application to Monitoring Systems”, Technical Report 9-276, Queens University, Kingston, Ontario K7L 3N6 (1991).

[14] M. B. Özcan, P. W. Parry, I. Morrey and J. Siddiqi, “Visualisation of Executable Formal Specifications for User Validation”, Lectures Notes in Computer Science, 1385, 142-157, Springer-Verlag, (1998).

[15] M. B. Özcan, P. W. Parry, I. Morrey and J. Siddiqi, “Requirements Validation based on the Visualisation of Executable Formal Specifications”, COMPSAC98, 22nd Annual International Computer Software & Applications Conference, 381-386, 1998.

Extended Object Diagrams for Transformational Specifications in Modeling Environments

Dragan Milicev

University of Belgrade
School of Electrical Engineering, Dept. Comp. Sc. & Eng.
POB 35-54, 11120 Belgrade, Serbia, Yugoslavia
emiliced@etf.bg.ac.yu

Abstract

One of the most important features of software tools for domain-specific modeling is automatic output generation. Since the existing techniques for specifying output generation in customizable modeling and metamodeling environments suffer from some weaknesses analyzed in this paper, a new approach is proposed. The analysis is based on the observation that the output generation is a process of transformation of a model from the source domain into the model from the target domain. If the domains are at distant levels of abstraction, the mapping is difficult to specify, maintain, and reuse. Therefore, the proposed approach introduces one or more intermediate domains. Assuming that the source, target, and intermediate domains are conceptually modeled (metamodeled) using the object-oriented paradigm, the proposed approach uses extended UML object diagrams for specifying the mapping between them. The diagrams specify instances and links that should be created by the transformational process. The proposed extensions are the concepts of conditional, repetitive, and sequential creation. These concepts are implemented using the standard UML extensibility mechanisms. Several examples from different software engineering domains are presented in the paper. They prove some important benefits of the approach: the specifications are clear and concise, easy to maintain and modify. Besides, the approach leads to better reuse of domain models and to remarkably shorter production time.

Keywords: object-oriented modeling, Unified Modeling Language (UML), object diagram, metamodeling, model transformations

1 Introduction

Modeling is a central part of all the activities that lead up to the deployment of good software, as of any other engineering system [4]. Each modeling domain lies upon another model that defines (1) abstractions of the domain; (2) their properties and relationships; (3) their semantics and behavior in the model; (4) their visual appearance (notation) and behavior in the supporting tool. The latter, underlying model is called the *metamodel* of the considered modeling domain. Therefore, metamodeling is the process of defining the metamodel of the considered modeling domain. 'Meta' should be treated as a relative reference, not as an absolute qualification: each modeling domain has its underlying metamodel, which is specified by abstractions of another meta-metamodel, etc. [11]. This paper is focused to the modeling domains that can be metamodeled using the usual object-oriented paradigm [4], as opposed to some other paradigms, such as grammar-based specifications.

Apart from their important roles in specifying, documenting, and visualizing systems, the purpose of modeling tools is most often system construction [4], where 'construction' means producing output from the system specification that may be interpreted by a certain external environment to provide the desired system's behavior. The examples of output include, but are not limited to: documentation, source code in a certain programming language, database scheme, hardware description, or any other formally defined structure. It may be observed that the output generation is actually a transformation of the user-specified model from the domain of his interest into the model from another target domain. (Precisely, this is actually *generation* of another model, but the term *transformation* is used in this context more often.) The problem of specifying output generation may exist in three different contexts. (1) In fixed, non-customizable domain-specific modeling tools, where the source and target domain metamodels, along with the mapping between them are fixed at the time of the tool development (as the problem of designing the output generation feature). (2) In customizable modeling tools, where the metamodels are fixed, but the mapping is customizable by the user. For example, a modeling tool such as a CASE tool may offer interfaces to the built-in metamodels (e.g., the UML metamodel, a metamodel of the target programming language, the relational metamodel, etc.), and the user may specify the mapping. (3) In fully featured metamodeling tools, where the user can specify both the metamodels and the mapping.

This paper discusses the problems of the techniques for specifying output generation implemented so far in the existing (meta-) modeling tools, and proposes a new approach that deals with the problems. The approach has two major contributions.

First, very often the source and the target domains are at distant levels of abstraction, and the mapping is difficult to specify, maintain, and reuse. Therefore, the

```
class FSM;

class FSMState {
public:
  FSMState (FSM* fsm) : myFSM(fsm){}

  virtual FSMState* s1 ();
  virtual FSMState* s2 ();
  virtual FSMState* s3 ();

  virtual void entry () {}
  virtual void exit  () {}

protected:
  FSM* fsm () const {return myFSM;}
private:
  FSM* myFSM;
};
```

(b)

```
class FSMStateA : public FSMState {
public:
  FSMStateA(FSM* fsm) :FSMState(fsm){}

  virtual FSMState* s1 ();
  virtual FSMState* s2 ();

  virtual void entry () { ... }
  virtual void exit  () { ... }
};

FSMState* FSMStateA::s1 () {
  fsm()->t1();
  return &(fsm()->stateA);
}

FSMState* FSMStateA::s2 () {
  fsm()->t2();
  return &(fsm()->stateB);
}
```

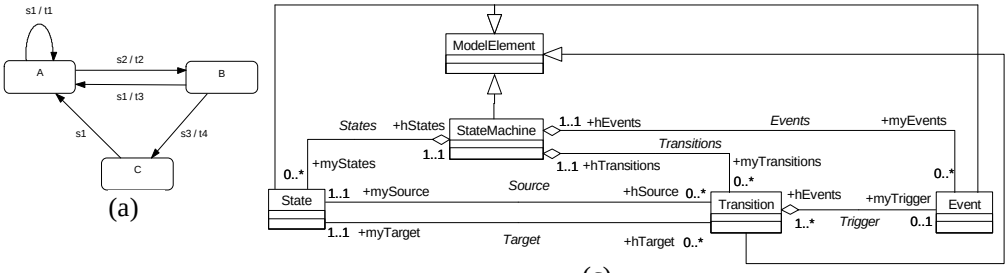


Figure 1: Demonstrational example: Code generation for state machines. (a) A sample state machine. (b) An excerpt from the generated code. (c) The metamodel.

proposed approach introduces one or more intermediate domains. In other words, it simplifies complex and cumbersome transformations of a model into another representation by doing the transformation in multiple steps. This has the advantage that each step becomes simpler and that existing transformation can be reused.

Second, it uses extended UML object diagrams to specify visually the mapping between the domains. The diagrams specify instances and links that should be created by the transformational process. The proposed extensions are the concepts of conditional, repetitive, and sequential creation. These concepts are implemented using the standard UML extensibility mechanisms. Consequently, the specifications are clear and concise, easy to maintain and modify, and lead to shorter production time.

The paper continues as follows. Section 2 reveals the motivation for this work and defines the problem precisely using a simple demonstrative example. Section 3 briefly discusses the related work. The idea of our approach is presented in Section 4. Section 5 shows several examples that illustrate the applicability and efficiency of the approach. The paper ends with conclusions.

2 Motivation and problem statement

The problem and the proposed solution will be demonstrated using a simple example from the field of telecommunication software development. The goal is to develop a simple modeling tool that generates C++ code for state-machine models. The code generation for state machines should be completely customizable: the user should be able to change the code generated from the same model if he needs another execution model due to performance, concurrency, distribution, or other requirements.

The example is shown in Figure 1. It is assumed that the desired code is obtained using the *State* design pattern [6]. It is also assumed that the user has specified a state machine named `FSM` as shown in Figure 1a. For this example, several classes are generated in the output C++ code. The first is named `FSM` and is the interface class whose behavior is specified in the model by the given state machine. It contains operations that correspond to the events of the state machine. The second class is abstract and is named `FSMState`. It contains one polymorphic operation for each event. Finally, one class derived from `FSMState` is generated for each state. It overrides the operations that represent those events on which the state reacts. These operations perform transitional actions and return the target state. Other details may be found in [6]. The metamodel of the domain (state machines) is shown in Figure 1c. (This is a simplified version of the metamodel for state machines from [14].)

Now, the code generation strategy to be applied to each state machine should be specified. Let us consider two possible approaches. A straightforward one is to hard code the output generation scheme in an operation (e.g., a member function of the class `StateMachine` that implements the state machine abstraction in the modeling tool). The operation should read the data from the model

instances (i.e. to navigate through the model and read attribute values) and produce the textual output following the C++ syntax and semantics. An excerpt of such operation that generates the beginning of the declaration for the class `FSMState` may be:

```
// Generate base state class:
output<<"class "<<(this->name+"State")<<"{\n";
output<<"public:\n";
output<<"    "<<(this->name+"State")<<"(";
output<<(this->name)<<"* fsm):myFSM(fsm){}\n";
//...
```

The drawbacks of this approach are obvious:

(1) The process of specifying is extremely tedious, time-consuming, and error-prone.

(2) The user must deal with the complexity of the target domain (C++ syntax and semantics).

(3) The built-in general-purpose and reusable C++ code generator is not used at all.

(4) Any modification is very difficult to apply because the code is not clear and comprehensible.

(5) The code is not reusable.

(6) The user must deal with the technical details such as the correctness of the output stream, opening files (.h and .cpp files must be created in C++), etc.

The core reasons for the listed drawbacks may be revealed by the following observation. The output generation process may be viewed as a creation of a target model from the source model. The source model is the model explicitly specified by the user in the modeling tool and consists of instances of state machines, states, events, and other abstractions from the source domain. The target model is the textual output, i.e. the generated C++ source code whose metamodel is implicitly assumed by the user (C++ syntax and semantics). The code of the given operation is actually a specification of the mapping between the two domains. Since the two domains are at distant levels of abstraction, their direct mapping by the hard-coded special-purpose generator has all these drawbacks.

This mapping between two distant domains has the same disadvantages as the process of object-oriented programming in the target programming language (e.g. C++) without previous modeling at a higher level of abstraction (e.g. with UML). This is because the programming language level of abstraction is too far from the level of abstraction that is suitable for the developer's way of thinking. For our example, instead of directly generating the textual output, it may be reasonable to create an intermediate model based on a metamodel of a higher level of abstraction, such as a subset of UML, which includes abstractions supported directly by a common object-oriented programming language (class, operation, attribute, etc.). Because the general-purpose C++ code generator from UML models may be built in the tool, it may be reused for the generated intermediate model. Hence, the idea is to create needed instances from the intermediate domain using the built-in UML metamodel, and then to invoke the built-in code generator to produce the output:

```
void StateMachine::generateCode () {
// Temporary package for the intermediate model:
    Package& pck = Package::create();

    // Intermediate model:
    // Base state class:
    Class& baseState = Class::create(pck);
    baseState.name = this->name+"State";

    // Base state class constructor:
    Method& baseStateConstr =
        Method::create(pck);
    baseStateConstr.name = this->name+"State";
    Link::create(Members::Instance(),
                 baseState,baseStateConstr);
    //...
    // Code generation:
    pck.generateCode();
}
```

This code excerpt shows the creation of instances for the class `FSMState` and its constructor. It creates instances of UML abstractions Class and Method, using the built-in UML metamodel interface. Then, it sets the values of their attributes. Finally, it creates links between these instances. All these instances are packed into a temporary package for which the output is generated in the end.

This approach remedies most of the drawbacks of the first approach. In the first place, it eliminates the impendance-matching problem between the source and target domains by introducing an intermediate level. By doing this, the process of output generation is split into two steps, where each step is much easier to specify than before. Besides, the second step is supported by the built-in and reusable code generator. Thus, the first-step mapping specification is completely reusable for other target languages, provided that general-purpose code generators from UML are available. However, the specification of the operation body is still tedious and error-prone. Besides, the code may be very complex and difficult to manage. Since it is actually a specification of the process of creating instances from the intermediate domain, where both source and intermediate domains may be formally defined by their metamodels, this specification may be provided in another formal way. The idea is to use a visual specification, preferably one that is compatible with the UML standard. This is the subject of this paper.

3 Overview of the Related Work

Due to the fact that the process of domain-specific metamodeling can be formalized, the need for tool support of this process has been recognized for long [2, 11, 13]. This need was first met in the domain of automatic programming environment generation [10]. By the maturation of numerous software-engineering methodologies and notations, especially of object-oriented ones, which all have been developed with the perspective of CASE tools support, the field of meta-CASE research has evolved [2, 11]. However, we do not constrain our discussion here on the field of software modeling, CASE, and meta-CASE tools, although it is

our major field of interest with a strong research background. The results of our work may be applied to metamodeling domains other than software systems. That is why we use the term "metamodeling environment" rather than the term "meta-CASE tool."

There are a number of approaches addressing a similar problem using structural transformations of grammar-based models and various rule-based techniques [7, 8, 9]. Their goal is to transform a user-defined structural model written in a domain-specific language into another structural model in another target language. Although the goal is similar to the one presented here (transformation of models), there are a number of differences. First, although their principles may be generalized to more abstract terms, they primarily deal with textual models (or, more generally, with strings of entities). Second, their 'metamodels' are expressed with grammars, where the entities are defined hierarchically (using sub-entities), and where recursion is the main difficulty, instead of the object-oriented paradigm that is used here. The main purpose of the supporting environments in that case is to build an internal representation (derivation tree) from the user-defined model (textual program) by parsing it, and then to transform this internal representation into the target internal representation. Thus, the internal structure of the model is inherently a tree. In the modeling environments that use object-oriented paradigm for metamodeling, there is no need for the parsing phase, because the user explicitly creates the instances of abstractions and their links. Therefore, the model representation is a graph of objects (instances of classes) connected with links (instances of associations). This is why the approach presented here may be considered as a more general structural transformation.

The rule-based approaches allow the user to specify the differences between the source and the target grammars ('metamodels') and a supporting tool may help in generating the model transformer but with some intervention of the user [7]. The approach presented here allows the user to specify the mapping, and the transformer is generated without any intervention of the user. Furthermore, defining a grammar for a certain domain and specifying the mapping between the grammars may be a difficult task because it requires more sophisticated work than defining (in meta-environments) or just understanding (in customizable modeling environments) the metamodels specified in object-oriented terms. It is evident that some domains may be metamodeled with much less effort using the object-oriented paradigm instead of grammars. This includes most modeling methods with visual notations. For such cases, the proposed approach is definitely superior. Consequently, the proposed approach may be treated as a complement to the grammar-based structural transformations, more suitable for object-oriented metamodels.

A research field also related to metamodeling is the field of visual programming languages (VPL) [1, 5, 17]. However, the underlying metamodels of VPLs are also grammars [5] or other formal models. Consequently, VPL metaenvironments have the same characteristics as the grammar-based environments described previously. In an automatically generated VPL environment, the user chooses a graphical element and puts it onto a diagram rather arbitrarily. The task of the tool is to check the correctness of the diagram when the translation operation is explicitly invoked, considering the underlying grammar. Then, it should parse the grammar elements and develop an internal representation analogous to the derivation tree in classical compilers. On the other side, in object-oriented modeling environments, the user is usually explicitly constrained in designing diagrams, and the contents of the diagram is determined at the time of its construction. The user creates and manages explicitly *model* (semantic) elements, while visual elements are only views to them. Besides, the problem of the model transformation, which is the subject of this paper, is not considered as an important one in the field of VPLs.

Automatic generation of CASE tools has been an attractive discipline for years, and a lot of extensible CASE and meta-CASE tools, both commercial and academic ones, are available at the moment [18, 19, 20, 21, 22, 23, 24, 25]. A major commonality (and a weakness also) of all existing meta-CASE tools that is of greatest interest to our work is the output generation facility. All these tools provide programming interfaces to their metamodels through which the user may access the models in the generated CASE tools to produce the output. However, output generation is always specified using a scripting language that is proprietary and vendor-specific. Hence, the first hard-coded output generator strategy described in the previous section is available to the user. As they often offer a flexible interface to their metamodels, the user may create an intermediate model as described in the second approach in the previous section. Nevertheless, this intermediate model may be created only using the same scripting language, and there is no other opportunity for doing this at a higher level of abstraction (e.g., visually). None of these tools promotes domain mapping as an explicitly supported strategy available to the user. As a conclusion, to the best of our knowledge, we are not aware of any other approach that is closely related to the one presented in this paper.

4 Domain mapping specification

The idea of the domain mapping (Figure 3) is to create an intermediate metamodel and a specification of the mapping from the source to the intermediate domain. A model transformer is automatically generated from the mapping specification. It is used to create the intermediate model from the user-defined source model. Finally, the built-in code generator produces the ultimate output. The benefit is because each of the transformations

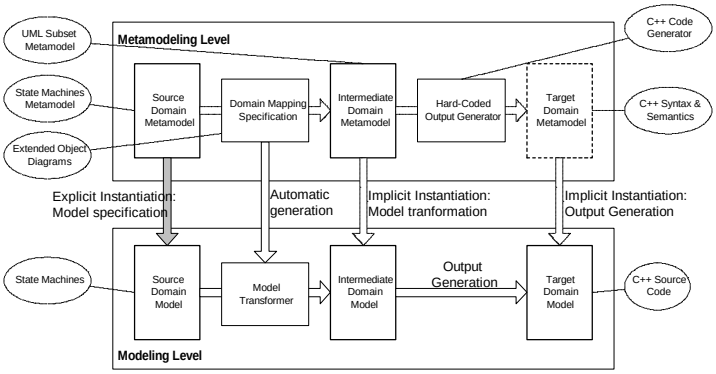


Figure 3: The idea of the domain-mapping strategy in the context of the demonstrational example. The transformation from the source into the target domain is split into two (or generally more) steps in order to cope with the complexity of the mapping specification.

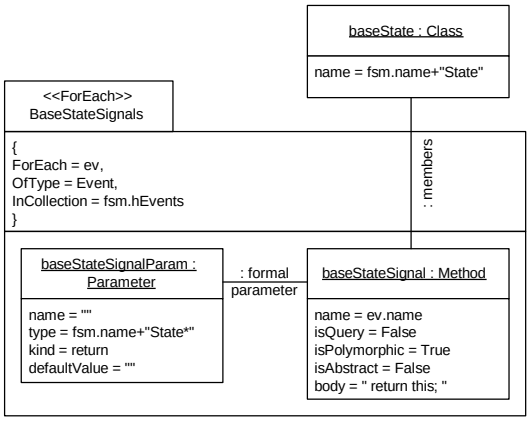


Figure 5: "ForEach" concept for repetitive object specification. The diagram shows only the specification for the base class `FSMState` and its member functions generated for the state machine's events. It belongs to the context of the state machine accessible through the `fsm` identifier.

is much less complex that the direct transformation, and is thus easier to specify, maintain, and reuse.

The specification of the domain mapping should be formal and preferably graphical. Since it is actually a specification of a set of instances of the abstractions (classes) from the intermediate domain that should be created, UML object diagrams may be used. An excerpt for our example is shown in Figure 4. It is assumed that the diagram is defined for one instance from the source model, which is referred to by a certain identifier in the diagram. For this example, it is an instance of the type StateMachine, referred to by the identifier `fsm`. The diagram specifies the set of instances of classes from the intermediate metamodel that should be created for each StateMachine instance `fsm` from the source model. The diagram specifies also the values of their attributes, along with the links between them. The attribute values are defined as expressions that refer to the instances from the

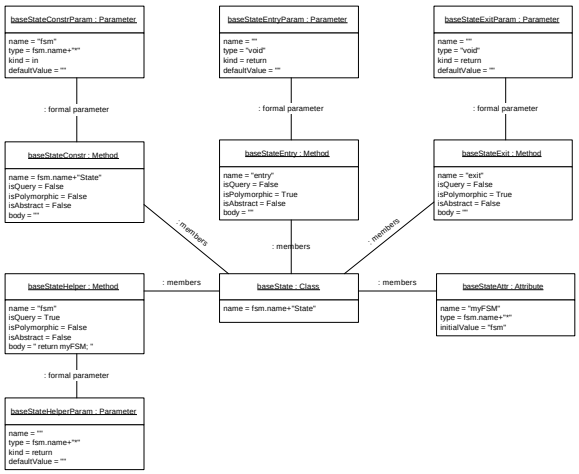


Figure 4: A simple part of the object diagram for the domain mapping specification of the demonstrational example. The diagram shows only the specifications for the base class `FSMState` and its members that are generated by default. The diagram belongs to the context of the state machine accessible through the `fsm` identifier.

source model and their attribute values, using the navigation through the source model. The links are instances of associations from the intermediate metamodel.

A standard object diagram is not sufficient for the mapping purposes. There is also a need for repetitive object creation. For our example, one method in the base state class should be created for each event that the machine reacts upon (see Figure 1). For this purpose, we use a stereotyped package with the stereotype ForEach. The example is shown in Figure 5. ForEach package represents iteration through a collection of instances from the source model and creation of a set of intermediate domain instances for each of them. It contains three tagged values:

- ForEach: An identifier that is introduced into the scope of this package. It may be used inside the scope of the package to refer to the current element of the iteration.
- OfType: The type of the current element. The iteration is type-sensitive, in the sense that only the elements of the specified type from the collection are processed, and the others are ignored (in the case that the elements are polymorphic). The type is from the source metamodel.
- InCollection: An expression that evaluates to a collection of the instances from the source model to iterate.

When a link connects an instance inside a package and another outside that package, then each repetitive instance created by the iteration will be linked to the outer instance. For the expressions that are used to define attribute values or collection in a ForEach package, any formal language for navigation through the source model

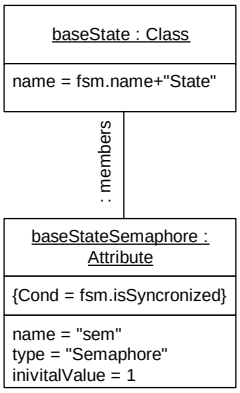


Figure 6: Conditional object creation. The diagram shows only the specification for the base class `FSMState` and its data member (a semaphore) generated for synchronization, only if the state machine is "synchronized."

may be used. For example, Object Constraint Language (OCL) may be used [15] if the tool is capable of parsing these expressions or the programming interface of the model is OCL-compliant. The other option is the scripting language used in the tool.

Another needed concept is conditional creation. An instance, a link, or a ForEach package may be tagged with a condition that is a Boolean expression again in the scope of the source model. If the expression evaluates to False when the intermediate model is being created, the conditional instance or link is not created, or the package is ignored. A simple example is shown in Figure 6. The example assumes that the StateMachine type in the source metamodel has a Boolean attribute named "isSynchronized." If the value of this attribute is True, the generated state machine code should be mutually exclusive in a concurrent environment. This is achieved by an attribute of type Semaphore that is generated in the base state class and the corresponding wait/signal operations in all publicly accessible operations (not shown in the picture).

Since ForEach packages actually represent loops in the process of intermediate model generation, they may be nested. An example is shown in Figure 7. For our example, a derived class should be created for each state. This is specified with the outer ForEach package. For each of the events this state reacts upon, an operation should be generated in this class (specified with the nested package).

A ForEach package introduces a scope of the expressions. The rules for the scope nesting are identical as in the traditional procedural programming languages. An expression may use identifiers from the scope in which it is defined, as well as from its enclosing scopes. A ForEach identifier is local for its package, and hides the same identifiers from the enclosing scopes.

It has been mentioned that the presented specifications belong to the context of one instance from the source model. A certain identifier (fsm in our example) refers to this instance. However, we generalize

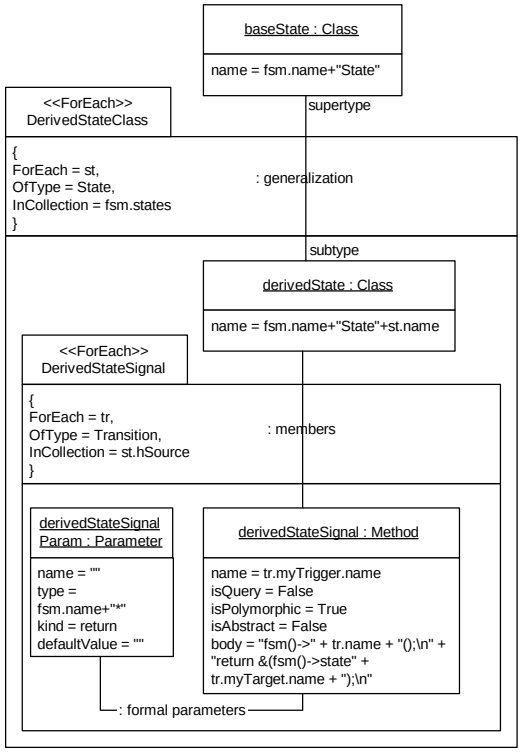


Figure 7: Nesting of "ForEach" packages. The diagram shows a part of the specification for the derived state classes and their member functions for the events.

this context in the following way. The whole mapping is specified following the UML style of hierarchically organizing models in packages. Thus, the mapping specification is actually another model, represented with a package hierarchy, where each package may, but need not be a ForEach one, and may own instances, links, and other packages. (Ordinary packages serve as grouping elements only and map into the same grouping of the elements of the generated model.) Besides, following the UML diagrammatic style, it is allowed that the contents of one package are defined by several diagrams to enhance readability and clearance. Therefore, all the diagrams shown in figures 4 to 7 belong to a ForEach package with the InCollection value referring to a tool-manipulated collection of all instances of the given type in the source model (for our example, something like: `StateMachine::getAllInstances()`).

The generated model is organized as a hierarchy of packages, where each package is an unordered collection of the elements it owns by default. More precisely, the ordering of the elements in a package is implicitly determined by the order of their creation; by default, the ordering of creation is not defined. Sometimes, however, an explicit ordering of the elements is needed. This ordering may ensure a proper sequential traversal through the model elements; for example, if a sequential structure (e.g., text) is to be further generated from that model. If an element *x* is to be created after an element *y*, it may be considered dependent on *y*. This relationship is specified

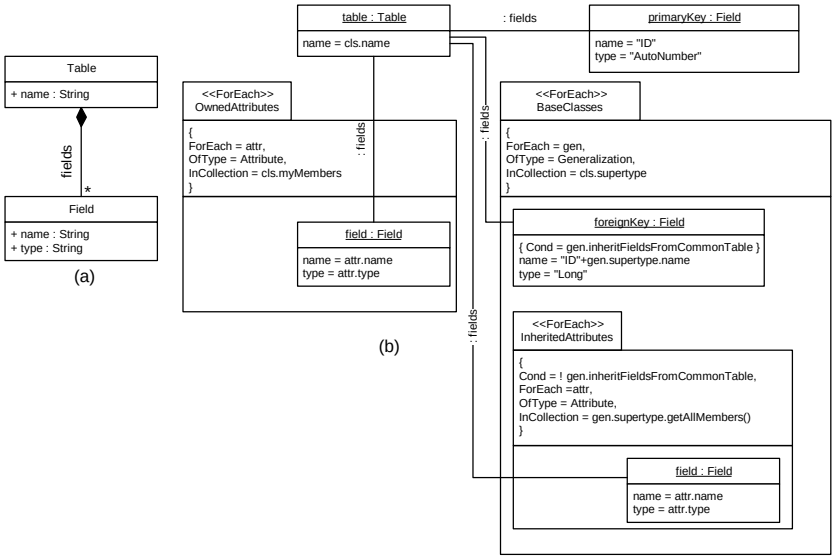


Figure 8: Example: Generation of the relational database scheme from a UML class model. This example focuses on inheritance. The source domain metamodel is UML (not shown here). (a) The target domain metamodel (relational). (b) The domain mapping spe cification. Operation `getAllMembers()` returns the collection of all owned *and* inherited members of a GeneralizableElement (Type in this case).

in the mapping diagram with a dependency from *x* to *y*, stereotyped as <<sequence>> [4]. Consequently, *y* will precede *x* in a traversal of the elements of their enclosing generated package.

From the diagrams formally specified as shown above, the source or the scripting code for the model transformer used at the modeling level may be generated automatically. For our example, the code is shown in the appendix, and the details are reported elsewhere [12]. The algorithm for generation of such code is as follows. For a package, the algorithm is: first introduce implicit sequence dependencies from links to the instances they connect, then sort topologically the owned elements according to the sequence dependencies, and then generate code for each of the elements (recursively for its nested packages). If the package is a ForEach one, the specified iteration will be performed, and one package in the generated model will be created for each iterated element. Each instance generates statements that will first create an object of the specified type and then set its attributes to the specified values, using the programming interface of the modeling tool.

The approach of the domain mapping may be generalized to arbitrarily many intermediate domains. The idea is that a tool may generate several intermediate models as different levels of modeling abstraction, using the domain mapping specifications. The process of creation of intermediate models may be viewed as a descent down the abstraction levels. The tool may allow the user to make changes in each intermediate model, prior to generating the next one, if the user is not satisfied with the automatically generated model. By using different domains for intermediate models, it may be expected that a better understanding of the problem and more complete modeling may be achieved. On the other side, other more abstract domains may be built on top of already designed domains, and the transformation may be easily specified using the mapping from the new domain into the already implemented lower-level one. This is one of the directions for the future work.

5 Case study and evaluation

The example of the modeling tool for state machines has been implemented as a final project for the B.Sc. degree at the University of Belgrade. The specification had about 30 instances and seven ForEach packages. The implementation of the code generation part, using domain mapping, and a built-in C++ code generator, took about ten hours, including testing.

Apart from this example, two more are presented here (these are just small excerpts of much more complex examples from practice). The second example is the problem of transforming object-oriented class model into the relational database model. This is a common task in object-oriented programming when persistence of objects is accomplished by a relational database. Here, the source domain is UML. The target domain is the code that may be used to define database tables and fields, e.g., SQL declarations. However, the direct mapping from the class model into the textual SQL declarations is difficult to specify. Therefore, an intermediate domain is introduced, with the metamodel shown in Figure 8a. It is a simplified version that encompasses tables and fields only. It is now easy to specify generation of SQL declarations from this intermediate domain, because it is almost (if not completely) one-to-one mapping. In this example, the accent is on inheritance, as the most difficult task in this process. It is assumed that the user is offered two strategies of implementing inheritance in relational tables. The first one assumes that a derived class has its own

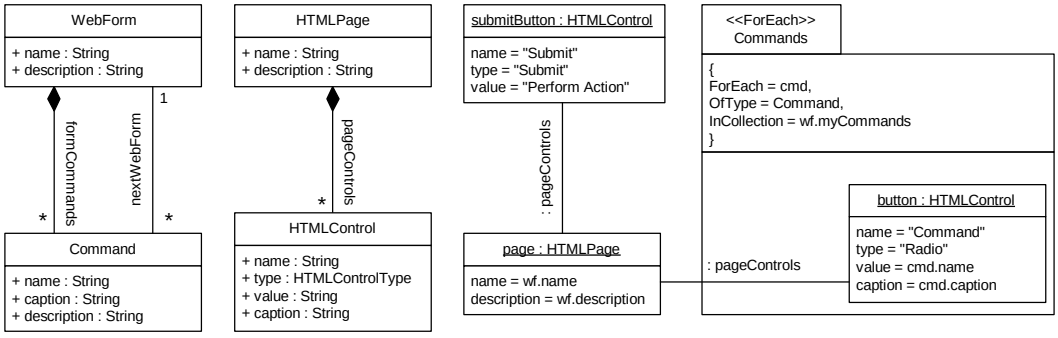


Figure 9: Example: Web design tool. (a) An excerpt from the source domain metamodel. (b) An excerpt from the target domain metamodel. (c) The domain mapping specification.

independent table, with all inherited attributes copied into its own table. In this approach, an object is represented with a single record in the table that represents its class. In the second approach, a derived class has a table without inherited attributes, but its records are dependent on the records from the table that represents its base class. In this second approach, an object is represented by a set of records in the tables that represent its own class and its base class. We assume that the user may chose one of the approaches for each generalization in the class model, by setting the Boolean attribute of the generalization named "inheritFieldsFromCommonTable." This attribute should be added to the UML metamodel as a tagged value of generalization. If this field is set to True, the second approach is chosen. In both approaches, the table should have a primary key (of type "AutoNumber" and named "ID"), and the set of the fields for the attributes of the class. In the first approach, the table should have the fields for all attributes from the base class, for each inheritance relationship tagged with inheritFieldsFromCommonTable = False. In the second approach, the table should have only a foreign key (of type "Long" and named "ID"+<baseClassName>) to link it to the base class table. The corresponding mapping scheme is shown in Figure 8b.

The third example shows a case when UML is not used as any of the domains. It is taken from one of our projects with database-centric web application development. A method and infrastructure for rapid application development have been developed. A very small part of the idea is presented here, just to illustrate the usage of metamodeling and domain mapping. In this approach, application is modeled by the navigation through *web forms*. From one web form, the user can choose a *command*, which performs some actions in the database on the server and displays another web form. The commands are implemented as radio button options in the web form, and a "Submit" button that posts the data from the form to the server. A very small part of the source domain metamodel is shown in Figure 9a. This domain should be mapped into the standard HTML textual output. However, this mapping is complex because the source domain has other concepts not shown here. Therefore, an intermediate model is introduced that may be mapped one-to-one to the target domain. It contains abstractions such as an HTMLPage or an HTMLControl (text box, list box, radio button, etc.). This metamodel is shown in Figure 9b. As in the previous example, generation of HTML from the intermediate domain is straightforward. The mapping scheme for this example is shown in Figure 9c. The author implemented the complete prototype tool in only three days, including metamodeling, code generation, and testing.

In practice, the following method for defining intermediate domains and mapping specifications is proposed. After the source domain is defined and well understood, the most important task is the design of its metamodel. All common principles of object-oriented analysis and design may be applied to this process [3]. Then, the desired target output is informally specified and supported by an example. For this purpose, a simple yet descriptive example from the source domain is developed. Then, the desired code for this example is generated manually. The result of this process corresponds to the example shown in Figure 1. Afterwards, an intermediate domain that will make the output generation less complex is found. It should be very close to the target domain, so that the desired output may be easily generated from it. If it is still conceptually far from the source domain, other intermediate domains should be built upon it, etc. We have successfully found such a domain in all the cases. Reuse of already developed domain models is of much help. For example, if the target output is C++ or any other object-oriented programming language code, we use a UML subset as the intermediate domain. Another useful and reusable example is the relational domain. The metamodel of the intermediate domain should be built, too, if it is not already available. Finally, the domain mapping is specified using the following procedure. The developer goes through the sample output, and tries to find out of which element in the intermediate model that part of the code is an outcome. It is then specified in the mapping object diagram. The procedure is applied iteratively and incrementally. This procedure is much easier than the hard-code approaches, because the elements of the target

output that originate from the same source model element may be spread all over the target model. For instance, in our first example, the events of a state machine produce operation declarations in many separate classes. Therefore, it is easier to go sequentially through the generated output and build incrementally the domain mapping object diagram as the need for each of its elements arises. Other possible heuristics and a more formal approach to this process will be investigated in the future work.

The research team from the University of Belgrade has successfully used the described approach in several other large projects. All the examples confirmed the expectations on possible benefits of the strategy. The specifications of output generation are clear and concise, easy to maintain, modify, and reuse. They are hierarchically organized, visually presented (using multiple consistent diagrams), and thus cope well with a potential complexity of the mapping. It is possible to build the mapping specifications incrementally and iteratively, and to test them using only partially developed object diagrams. (Such incremental testing of partially defined mappings is not available in other techniques.) The process of specification is less tedious and error-prone. As the most important benefit, the development of output generator is shortened a lot. For instance, the first example (state machines) was started by using the conventional hard-coded approaches. It took us several weeks only to specify, without testing and debugging that were extremely difficult. By using the domain mapping strategy, we have reduced the working time to the order of hours. Production time will be shortened even more when a considerable repository of domain models and their transformers to various versions of the target implementation is created. In that case, user-defined models and transformers may be reused for different versions of the target implementation by using different transformers of the intermediate domains from the repository. Besides, as already stated, the mapping from the higher-level domains into the reusable intermediate domains may be defined with less effort than before.

Nevertheless, there are some weaknesses of our approach recognized so far. Although the specification supports conditional, sequential, and repetitive instance creation, it does not support recursion. Namely, one of the most important features of the traditional approaches that traverse the model structure and invoke operations for the model elements is that these operations may be recursive. This issue is particularly important when generating recursive structures, what is sometimes needed in textual output. In the examples we have studied so far, we have not encountered the need for recursion. However, the solution exists, but the future work will investigate this issue more deeply and will try to find a way for specifying recursion that best fits the definition of the existing concepts.

Another issue that may be improved is the visual specification. It is very often the case that a lot of instances and links must be specified in the domain mapping model, in order to describe formally the creation of an instance of a composite abstraction (e.g., a class and a set of its members in Figure 4). If that abstraction has a compound symbol defined in the accompanying notation, it may be much easier to use that symbol instead of the set of instances and links. It is possible to incorporate this feature in our approach, while completely preserving the described semantics.

6 Conclusions

The problem of specifying output generation in the context of modeling environments has been studied in this paper, and a new approach, called domain mapping, has been proposed. The approach is based on the observation that the automatic output generation is a process of creating a model in the target domain from the model in the source domain. If the domains are at distant levels of abstraction, the mapping is difficult to specify, maintain, and reuse. This is why one or more intermediate domains are introduced. The mapping is specified using UML object diagrams that show the instances from the intermediate domain that should be created by mapping. The diagrams are extended with the concepts of conditional, repetitive, and sequential creation. These concepts are implemented using the standard UML extensibility mechanisms.

Several case studies from different software engineering domains have been presented. All the examples have proved the major benefits of the approach. The specifications are clear and concise, thus easy to maintain and modify. The domain mapping strategy leads to a better reuse of domain models and to a remarkably shorter production time.

Acknowledgements

The author is grateful to D. Marjanovic, P. Nikolic, M. Ljeskovac, M. Zaric, and Lj. Lazarevic who contributed to the implementation of a supporting tool and the case study.

References

[1] Anlauff, M., Kutter, P. W., Pierantonio, A., "Montages/Gem-Mex: A Meta Visual Programming Generator," *Proc. 14th IEEE Symp. Visual Languages*, Sept. 1998

[2] Artsy, S., "Meta-modeling the OO Methods, Tools, and Interoperability Facilities," *OOPSLA'95 Workshop in Metamodeling in OO*, Oct. 1995

[3] Booch, G., *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, 1994

[4] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, 1999

[5] Costagliola, G., Tortora, G., Orefice, S., De Lucia, A., "Automatic Generation of Visual Programming Environments," *IEEE Computer*, Vol. 28, No. 3, March

1995, pp. 56-66
[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley Longman, 1995
[7] Garlan, D., Cai, L., Nord, R. L., "A Transformational Approach to Generating Application-Specific Environments," *Proc. Fifth ACM SIGSOFT Symp. Softw. Development Environments*, Dec. 1992, pp. 68-77
[8] Garlan, D., Krueger, C. W., Staudt, B. J., "A Structural Approach to the Evolution of Structure-Oriented Environments," *Proc. ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. Practical Softw. Development Environments*, Dec. 1986
[9] Habermann, A. N., Notkin, D. S., "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Vol. 12, No. 12, Dec. 1986, pp. 1117-1127
[10] Karrer, A. S., Scacchi, W., "Meta-Environments for Software Production," *Report from the ATRIUM Project*, Univ. of Southern California, Los Angeles, CA, Dec. 1994, http://www2.umassd.edu/SWPI/Atrium/localmat.html
[11] MetaModel.com, *Metamodeling Glossary*, http://www.metamodel.com
[12] Milicev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," submitted for publication, available on request
[13] Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proc. IEEE ECBS'98 Conf.*, 1998
[14] Rational Software Corp. et al., *UML Semantics*, Ver. 1.1, Sept. 1997
[15] Rational Software Corp. et al., *Object Constraint Language Specification*, Ver. 1.1, Sept. 1997
[16] Sztipanovits, J. et al. "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proc. IEEE ICECCS'95*, Nov. 1995, pp. 361-368
[17] Zhang, D.-Q., Zhang, K., "VisPro: A Visual Language Generation Toolset," *Proc. 14th IEEE Symp. Visual Languages*, Sept. 1998

Customizable CASE and meta-CASE tools

[18] Advanced Software Technologies, Inc., *Graphical Designer*, http://www.advancedsw.com
[19] Lincoln Software Ltd., *IPSYS ToolBuilder*, http://www.ipsys.com
[20] MetaCase Consulting, *MetaEdit+ Method Workbench*, http://www.metacase.com
[21] mip GmbH, *Alfabet*, http://www.alfabet.de
[22] Platinum Technology, *Paradigm Plus*, http://www.platinum.com/clearlake
[23] Rational Software Corporation, *Rational Rose*, http://www.rational.com
[24] Univ. of Alberta, *MetaView*, http://www.cs.ualberta.ca/news/CS/1998/research/
[25] Vanderbilt University, *Multigraph Architecture*, http://www.isis.vanderbilt.edu

Appendix

The generated C++ code for the model transformer (an excerpt for the diagram in Figure 7). `ForEach`/`EndForEach` are C++ macros that implement type-sensitive iteration.

```
// Temporary package for the intermediate model:
Package& pck = Package::create();
// Intermediate model:
ForEach(fsm,StateMachine,StateMachine::getAllInstances())
  // Generated for objects:
  // Object: baseState
  Class& baseState = Class::create(pck);
  baseState.name = fsm.name+"State";

  // Generated for ForEach packages:
  // Package: DerivedStateClass
  ForEach(st,State,fsm.states)
    // Generated for objects:
    // Object: derivedState
    Class& derivedState = Class::create(pck);
    derivedState.name = fsm.name+"State"+st.name;

    // Generated for ForEach packages:
    // Package: DerivedStateSignal
    ForEach(tr,Transition,st.hSource)
      // Generated for objects:
      // Object: derivedStateSignal
      Method& derivedStateSignal = Method::create(pck);
      derivedStateSignal.name = tr.myTrigger.name;
      derivedStateSignal.isQuery = False;
      derivedStateSignal.isPolymorfic = True;
      derivedStateSignal.isAbstract = False;
      derivedStateSignal.body = "fsm()->" + tr.name + "();\n" +
                                "return &(fsm()->state" + tr.myTarget.name + ");\n");
      // Object: derivedStateSignalParam
      Parameter& derivedStateSignalParam = Parameter::create(pck);
      derivedStateSignalParam.name = "";
      derivedStateSignalParam.type = fsm.name+"*";
      derivedStateSignalParam.kind = Return;
      derivedStateSignalParam.defaultValue = "";

      // Generated for ForEach packages:

      // Generated for links:
      // Link: <unnamed> of Association: members
      Link& link02 = Link::create(Members::Instance(),derivedState,derivedStateSignal);
      // Link: <unnamed> of Association: formal parameters
      Link& link03 = Link::create(FormalParameters::Instance(),
                     derivedStateSignal,derivedStateSignalParam);
    EndForEach(tr)

    // Generated for links:
    // Link: <unnamed> of Association: generalization
    Link& link01 = Link::create(Generalization::Instance(),derivedState,baseState);
  EndForEach(st)

  // Generated for links:

EndForEach(fsm)
```

Design decisions in building STP, a CASE tool

Michael M Werner
Wentworth Institute of
Technology
550 Huntington Avenue
Boston MA 02115
USA
(617) 989-4143
wernerm@wit.edu

ABSTRACT
This is an experience report detailing design decisions made in building a special purpose CASE tool.

STP is a CASE tool designed to facilitate automatic transformation of the source code supporting a set of related object-oriented applications. It uses reverse engineering to permit visualization of the object system by means of a simple arc and node graph. It allows for specification of desired transformations by manipulation of the graph. Submitted transformations are checked to make sure they do not introduce typing errors. The user may also request checking for possible changes in behavior for existing programs. Transformations are carried out correctly, updating both definitions and their usages throughout the program text. Regeneration facilities allow the revised source files to be printed out.

Building STP required a number of design decisions relating to the programming languages and platforms to be used, the means for reverse engineering existing systems and the modeling both in terms of underlying data structures and the visual representation of the object system. STP also has its own language for specifying transformations, a pattern for this language had to be developed. Also, there was a need for some uniformity in the way many of the processing tasks are carried out, in the end, the *Visitor Design Pattern* [5] was employed. The purpose of this paper is to describe those design decisions, and give the rationale for the choices that were made.

Keywords
CASE tool, reverse engineering, transformation program.

1 Introduction

This work is largely motivated by experiences working with evolution issues with Java programs. It is the contention of [15] that it is possible to build a tool to carry out extensive transformations of the source code of an object-oriented program, both safely and correctly. The theoretical interest was to determine what necessary preconditions must be satisfied by a proposed transformation. At the same time, it was felt necessary to demonstrate the practical feasibility of such a tool by actually constructing a prototype. The prototype, named *Schema Transformation Processor* (STP) has been expanded to include many of the features of a CASE (Computer Aided Software Engineering) tool, including modeling, code generation and reverse engineering. Building STP required a number of design decisions which are the subject of this paper. The rest of the paper is as follows: Section 2 gives an overview of STP, the next sections outline the design decisions that were made in the hopes of providing insight for other CASE tool builders, finally Section 10 has a progress update and an outline of future work on the project.

Draft of paper submitted to ICSE 2000.

2 An overview of STP

STP (*Schema Transformation Processor)* is a transformation tool under development for programs written in Java. The approach is described at length in [15]. The goals of STP are as follows:

1. Facilitate visualization of an object system using a simple model.
2. Allow the easy specification of changes to the model.
3. Automate checking to make sure that changes do not impair the type soundness of the system, or the behavior of existing programs.
4. Provide for broad changes such as flattening classes, which would expand into sequences of lower-level changes such as moving individual fields and operations.
5. Facilitate adding new capabilities to existing systems by means of the *Visitor Design Pattern* [5].

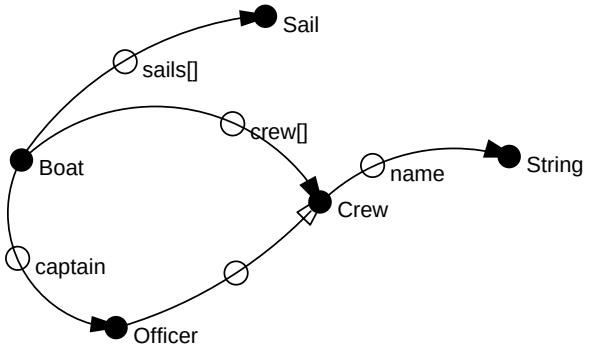


Figure 1: A screen shot from an STP session. The black nodes represent classes. The hollow arrowhead pointing to Crew indicates that Officer is a subclass of Crew. Name and captain are single-valued fields of Crew and Boat respectively. Sails and Crew are multiple-valued fields of Boat, as indicated by the "[]" in their labels.

3 Using STP

STP runs as a Java menu-equipped graphical application, or in batch mode as a console application. It has its own language, CSL (*Change Specification Language*), see Section 8, for specifying transformations.

STP can:

- Import a set of Java source files.
- Extract class graph information and display it using nodes and arcs.
- Display dialog windows for nodes and arcs.
- Display a fully functional text editor for source Java files and CSL files.
- Check preconditions prior to carrying out transformations.
- Apply transformations correctly, for example if a name is changed, the change is made wherever the name is used. STP sufficiently understands scoping rules and expression syntax to determine where changes should be made.
- Generate transformed Java source files.

Preconditions have been associated with each transformation supported by the system. In STP, the various transformations are represented by classes. In the current version, preconditions have been hard-coded into checking methods belonging to these classes. Future versions may allow the user to augment these preconditions using logical expressions. There are two types of preconditions. *Weak* preconditions are designed to protect the the type integrity of the object system, so that programs can continue to be compiled, *strong* preconditions are designed to protect existing programs, by ensuring that they can not only be recompiled, but also that they will behave as before. Since it is not feasible to check if two blocks of code have the same behavior, strong preconditions preclude the replacement of one method body by another. Such replacements might occur, for example, when a method renaming hides or unhides an inherited method. See [15] for a complete discussion of the preconditions that apply to each transformation.

Figure 1 shows a screen from a small STP session. In addition to the above capabilities, STP can also be used to generate Java source files in the first place. In Insert mode, the user clicks on the graph to add new class nodes, and drags with the mouse to add new arcs representing inheritance relationships, fields and operations. Using dialog screens, the new nodes and arcs are annotated with information such as name and multiplicity. STP then generates source files according to user preferences as to constructors, getters, setters, collection types for implementing multi-valued fields, etc. This capability is being extended to also generate C++ header and source file code.

Another feature of STP is to provide support for adding on new capabilities to existing systems by describing paths of navigation to be used by the new functions, and then automatically augmenting class definitions with methods supporting the navigation. Such navigational paths are called *itineraries* and are more fully described in [15]. Using STP a designer can describe an itinerary by clicking out its path in Attach mode.

The following sections will outline the many design decisions that were made in building STP.

4 Choice of a programming language

Java was chosen as the programming language for building STP for the following reasons:

1. The initial systems to be transformed are written in Java. Writing STP in Java as well demonstrates a kind of boot-strapping.
2. The necessary tools are readily available and easy to use in the Java environment. The graphical commands generally work the same on different platforms, such as Microsoft Windows and X-Windows.
3. An excellent compiler compiler, *JavaCC* [13] is freely available. The use of JavaCC is described later on.

4. The *Java Serialization Standard* [8] provides a means for objects to be saved, including recursively saving associated objects. This is used to advantage in STP to save models being worked on and reload them later on.

5 The visual representation of the class graph

STP uses a simple method known as an *IOM Graph* for visualizing class hierarchies and associations. IOM stands for *Implementation Object Model*, so-named because the model conforms closely to the programming language implementations. For example, binary associations are modeled at a lower level of abstraction simply as pairs of fields.

The IOM Graph uses only three constructs, namely circular nodes, directed arcs and labels. Filled circles represent types, arcs represent IS-A and HAS-A relationships as well as operations and attributes. IS-A arcs are from the subclass to the superclass (the target), and are indicated by having hollow triangles for arrowheads. For other arcs, the source is the class defining the field or operation, and the target is the node representing its type. Arcs are circular segments drawn through three points, the center of the source and target nodes, and a third point in between, which is the center of a hollow circle providing a clicking target for the arc itself, and an anchor for the arc label. Arcs were chosen rather than lines because often nodes have more than one link between them. Lines would overlap, but arcs can be shown separately by dragging their middle circles.

The labels are important, operations are distinguished from fields by labelling them using parenthesized parameter lists. Multiple-valued attributes and HAS-A links are additionally labelled by appending "[]" to their names.

It is reasonable to question the need for a new object model, when both OMT [12] and UML [1] are available and widely used. However, OMT and UML are both (1) too rich in describing semantic artifacts such as aggregation, and (2) too abstract in describing associations between classes, in that the implementation of such associations is not indicated. IOM provides all information needed to describe the STP transformations, with a minimum of clutter. This allows more classes to fit on a single screen, facilitating visualization [14]. Also, the arrows in IOM graphs can be highlighted to show itineraries (navigational paths). Figure 2 shows the IOM Graph equivalent of a UML object model.

A problem with any visual representation of an object model is that models of substantial systems become cluttered and hard to read. STP offers some partial solutions. For example, the user can express a preference as to which types of constructs to show, turning on or off the display of attributes, operations, etc. Scrolling is also offered. Even so, consider that the building of STP itself requires more than 250 separate classes, and the limitations of visualization become clear. What would help would be a zoom-in, zoom-out capability, similar to the idea of exploding a process in a data flow diagram [3]. However, this would require some notion of modularization of an object-oriented systems with modules larger than a class yet smaller than an application. Currently there is no agreement on the semantics of such a modularization.

6 Using parsing plus reflection for reverse engineering

Reflection refers to the ability to analyze the capabilities and structure of classes. Java provides support for reflection at run-time using the Java Virtual Machine. It is also possible to do a static analysis of Java class files by loading them using the Class.forName() method. By successively loading the classes which comprise an existing system, and using the operations available to java.lang.Class, such as getSuperclass(), getFields() and getMethods(), and then by using accessors available to java.lang.reflect.Field and java.lang.reflect.Method, it is possible to reconstruct the system of classes and associations. However, the source code is not recovered, making reflection inadequate for reengineering which works by modifying it.

STP uses a combination of parsing and reflection to recover design information. Parsing is used for code which is subject to eventual modification. The parsed code is analyzed to recover the desired design information. Later, modifications can be applied directly to the code contained in nodes of the abstract syntax tree.

JavaCC [13] was chosen for the parsing task. It combines both a lexer and parser description in a single file. The actions associated with each production are used to create objects, whose properties are set by the right hand side of the production. Another advantage of JavaCC is the availability of grammars for all versions of Java, as well as for other languages such as C++. An earlier version of STP used Lex and Yacc [6].

It is possible to reconstruct Java source files from class files, using tools such as Mocha [4]. Java class files actually store the original names of classes, fields, operations and other named constructs as strings [7]. Providing the source files have not been deliberately obfuscated, sufficient design information can be recovered to construct the IOM. This is useful in cases where the original Java source files are not available. See Figure 3.

Reflection is used for library code. Although this code will not be modified, visualization of the system requires that its structure be exposed as well. Library classes are included if they are specifically imported. When

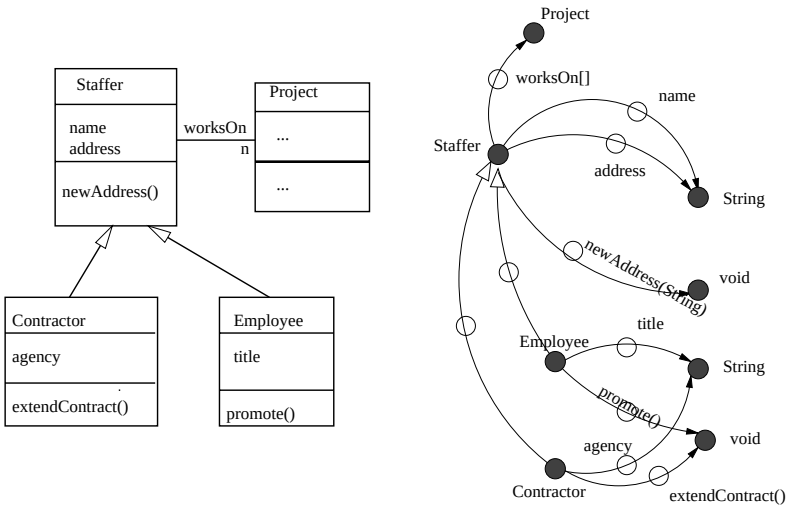


Figure 2: UML object model translated to IOM Graph

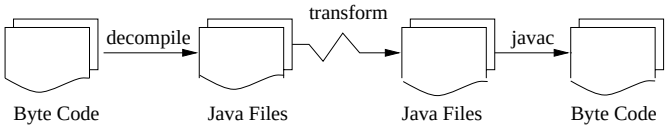


Figure 3: Transforming byte code in class files

an import statement contains a wild card as in "import java.util.*;", only classes actually utilized in the program are reflected.

One reason for reflecting imported classes is to help resolve method invocations inside expressions when the parameters involve imported fields or the return values of imported methods. The types of the parameters must be determined so as to properly resolve the method calls.

7 Internal data structures and algorithms

STP relies on two principal data structures:

1. An *Abstract Syntax Tree* (AST) representing the parsed source files.
2. The *Implementation Object Model* (IOM), which is an internal representation of the class graph. Its main constructs are classes, inheritance relationships, fields and operations and its visual representation is the *IOM Graph*.

The AST is constructed by parsing the Java source files. First, an object system representing the grammar of Java 1.2 was built. A separate grammar class was created for each non-terminal in the EBNF grammar. The class has fields for the items on the right hand side (RHS) of its defining production. When the RHS consists entirely of an alternation, an abstract class was used, with the classes representing the alternatives inheriting from it. Otherwise, if the RHS contains an alternation, as well as other items, fields for all the alternatives are provided. When a repetition appears on the RHS, a vector field is used.

Here are three examples showing the class definitions corresponding to Java grammar productions:

1. ForInit := LocalVariableDeclaration | StatementExpressionList

```
class ForInit{
}
class LocalVariableDeclaration extends ForInit {
    ...
}
class StatementExpressionList extends ForInit {
    ...
}
```

2. PrimaryExpression := PrimaryPrefix (PrimarySuffix)*

```
class PrimaryExpression{
   private PrimaryPrefix primaryPrefix;
   private Vector primarySuffix;
}
```

3. BlockStatement := LocalVariableDeclaration ";"| Statement

```
class BlockStatement{
   private LocalVariableDeclaration
       localVariableDeclaration;
   private Statement statement;
}
```

The parser is provided with actions to construct an abstract syntax tree from the source file. The nodes are objects of the non-terminal classes as defined above. In accordance with the *Visitor Design Pattern* [5], each

grammar class is also provided with a visit method, facilitating a depth-first traversal of the AST carrying an object of class Visitor. Visitor, in turn, has *before* and *after* methods for each grammar class. An example is:

void before(LocalVariableDeclaration host)

Concrete subclasses of Visitor can override these methods to do tasks at nodes visited using the host parameter for access. One such visitor has the task of constructing the IOM. In traversing the AST it extracts definitions of IOM constructs such as classes, fields and operations, and inserts them as nodes into the IOM model. The IOM nodes are provided with pointers into their definitions in the AST. Definitions may refer to types from imported files. Information about these types is extracted directly from the zip or jar files containing the imports. Where a wild card import statement is used, such as *import java.io.*;* only the types actually used in the program are added to the IOM.

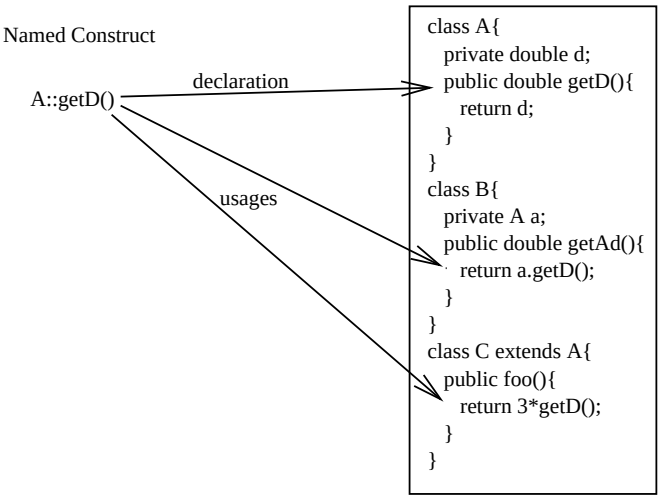


Figure 4: A Named Construct with pointers to its declaration and usages in the program. (Actually the pointers are into the abstract syntax tree built by parsing the program.)

The next step is to determine the usages of each of the IOM constructs by examining AST nodes representing method bodies and initializations. A specialized UsageTraceVisitor does this task. This visitor is aware of the name scoping and inheritance rules, and the expression syntax of the language so that it is able to make correct determinations of which constructs are used in expressions. Each node representing a usage is augmented with a pointer to the IOM node representing its definition. In turn the IOM node is provided with a pointer to the usage. This enables transformations such as renamings to be carried out correctly. Figure 4 shows pointers from an IOM node into its declaration and usages in the AST.

From a design point of view, one problem with Java is that it relies heavily on heterogeneous collections such as Vector and Hashtable, even when they store objects of only one type. In the IOM, fields of collection type are better represented as multiple valued fields of the underlying type. This is easily done when the original code contains arrays, or in the case of C++, templates. Since, as of this writing, proposals for templates [9, 10, 2] have not yet been adopted into the Java standard, the next best thing is to use type inference [11] to discover the underlying types of the collections. This inference can be done by examining the types of objects inserted into particular expressions, and the downcasts used when extracting objects from them. Solving the set of type constraints thus obtained can often determine the underlying type of collections. This capability is planned for STP, currently the STP user must manually infer the base types in order to redirect arcs from collection classes to ones which are more meaningful.

8 A language for specifying transformations

STP is based on a set of primitive transformations, which are stated in terms of the IOM Graph. Lower level primitives are summarized in Figure 5. The meanings of Add, Drop, Rename and Retarget are clear. Remult means to change the multiplicity of a field, Delegate and Reclaim refer to moving a class member along the HAS-A network, Lift and Lower refer to moves up or down the IS-A hierarchy.

Primitives are represented as both commands and objects. For example, the primitive command "RETARGET HAS-A x SOURCE A NEW TARGET C" illustrated in Figure 6 may be represented as an object of class RetargetHasa.

Aside from the lower level primitives, there are also a few upper level primitives such as Factor and Flatten, which expand to a number of lower level Lift and Lower primitives respectively, depending on the current state of the IOM Graph.

The Lower Level Primitives are as follows:

1. Changes to classes
   (a) ADD CLASS *ClassName*
   (b) DROP CLASS *ClassName*
   (c) RENAME CLASS *OldClassName* NEW NAME *NewClassName*
2. Changes to attributes
   (a) ADD ATTRIBUTE *attributeName* SOURCE *ClassName* TARGET *attributeType*
   (b) DROP ATTRIBUTE *attributeName* SOURCE *ClassName*

Figure 5: Construct Update-Type Matrix

Construct	Add	Drop	Rename	Retarget	Remult	Delegate	Reclaim	Lift	Lower
Class	X	X	X						
Attribute	X	X	X	X	X	X	X	X	X
Operation	X	X	X	X		X	X	X	X
HAS-A	X	X	X	X	X	X	X	X	X
IS-A	X	X							

(c) RENAME ATTRIBUTE *oldAttributeName* SOURCE *ClassName* NEW NAME *new attributeName*

(d) RETARGET ATTRIBUTE *attributeName* SOURCE *ClassName* NEW TARGET *new attributeType*

(e) REMULTIPLY ATTRIBUTE *attributeName* SOURCE *ClassName* NEW MULTIPLICITY (0 | 1)

3. Changes to the HAS-A Hierarchy

(a) ADD HAS-A *has-aName* SOURCE *SourceClassName* TARGET *TargetClassName* MULTIPLICITY (0 | 1)

(b) DROP HAS-A *has-aName* SOURCE *SourceClassName*

(c) RENAME HAS-A *oldHas-aName* SOURCE *SourceClassName* NEW NAME *newHas-a name*

(d) REMULTIPLY HAS-A *has-aName* SOURCE *SourceClassName* NEW MULTIPLICITY (0 | 1)

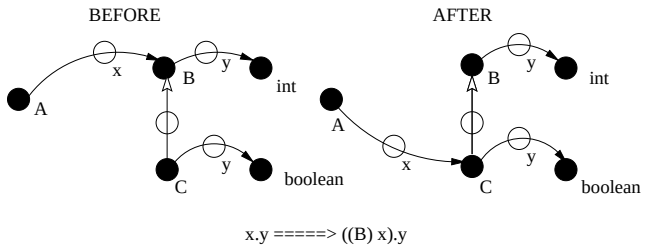


Figure 6: HAS-A A::x is retargeted from B to C, a narrowing of the target. The primary expression x.y is transformed to compensate by including an upcast to B.

(e) RETARGET HAS-A *has-aName* SOURCE *SourceClassName* NEW TARGET *NewTargetClassName*

4. Changes to operations

(a) ADD OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...) TARGET *typeName*

(b) DROP OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...);

(c) RENAME OPERATION *oldOperationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ..) NEW NAME *newOperationName*

(d) RETARGET OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...) NEW TARGET *newType*

5. Changes to the class hierarchy

(a) ADD IS-A *SubclassName SuperclassName*

(b) DROP IS-A *SubclassName SuperclassName*

6. Moving a field [1] or operation

(a) DELEGATE FIELD *fieldName* SOURCE *SourceClassName* USING HAS-A *has-aName*

(b) DELEGATE FIELD *fieldName* SOURCE *SourceClassName* USING OPERATION *operationName*

(c) RECLAIM FIELD *fieldName* SOURCE *SourceClassName* USING HAS-A *has-aName*

(d) RECLAIM FIELD *fieldName* SOURCE *SourceClassName* USING OPERATION *operationName*

(e) DELEGATE OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*

(f) DELEGATE OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*

(g) RECLAIM OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*

(h) RECLAIM OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*

[1] A field is either an attribute or a HAS-A

(i) LIFT FIELD *FieldName* SOURCE *ClassName1* TO *ClassName2*

(j) LIFT OPERATION *operationName* SOURCE *ClassName1* PARAMETERS (*type1*, *type2*, ...) TO *ClassName2*

(k) LOWER FIELD *FieldName* SOURCE *ClassName1* TO *ClassName2*

(l) LOWER OPERATION *operationName* SOURCE *ClassName1* PARAMETERS (*type1*, *type2*, ...) TO *ClassName2*

The commands have a uniform syntax. There is an action verb followed by the object type followed by its name and source class. Finally, the modification parameters are indicated.

9 Carrying out the transformations

Following the parsing of the source files, the construction of the IOM and its annotation by the UsageTraceVisitor, the system is ready to receive transformation commands. The user opens a new CSL file to enter wanted transformations, or loads an existing one. Most commands including renamings, insertions of new classes and members and deletions can be specified by interacting with the graphical IOM using the mouse, or by typing into the dialog boxes that are brought up by double or right mouse clicks. The appropriate CSL commands (see Section 8) are generated into the CSL file. The user indicates a preference as to whether the transformations are to be carried out under both weak and strong precondition checking or only under weak. The user then gives the *transform* command.

Preconditions for each type of transformation are checked in the IOM by traversing it using specialized IOM visitors. If the preconditions are satisfied, the transformation is carried out by additional IOM visitors, which follow pointers from the affected IOM construct to both its definition and all its usages in the AST. Both the IOM and the AST are updated, and the graphical IOM is refreshed on the user screen. Transformations can be applied one-at-a-time or in batch. When a suite of transformations is applied, each individual one is checked within the context established by applying the preceding ones. This is the reason for updating the IOM. The present version of STP lacks support for transactions. However, the user can simulate such support simply by keeping a back copy of the system, and reverting to it if one or more transformations in a suite fail.

Finally, the user can generate the altered source files using a specialized printing visitor which traverses the AST. The current prototype preserves the original comments by attaching them to AST nodes, however they may be moved slightly in the regenerated files.

The whole procedure is illustrated in Figure 7.

10 Conclusion

Building a CASE tool requires a number of decisions as to the modeling language used, the look and feel of the user interface and the types of existing design and programming tools which are best suited to speed the development. This paper reported on the construction of a CASE tool whose primary purpose is to demonstrate the feasibility of an automated software evolution approach. The tool is a prototype, that is to say that functionality has taken precedence over speed of execution, robustness and handling of exceptional cases.

As of this writing a working version of STP has been produced which has the capability of parsing substantial Java 1.2 (or earlier) programs, displaying the underlying class model in graphical form, carrying out automatic transformations, and generating revised source code. It will require further testing and debugging to assure smooth and consistent operation of these tasks.

Future directions for STP include:

1. Reverse engineering starting with binary class files.
2. The use of type inference to discover the base type of collection classes.
3. Automatic updating of serialized files in concert with the transformation of the object system which saved them. This will allow transformed systems to reload instances stored by earlier versions.
4. The expansion of the tool to work with object systems created by other languages such as C++. Some C++ generation is already available.
5. Provision of an alternative user interface employing industry standard UML [1].

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. www.math.luc.edu/pizza/gj/Documents/, 1998.

[3] W. S. Davis. *Systems Analysis and Design*. Addison-Wesley, 1983.

[4] D. Dyer. Java decompilers compared. *Java World*, July 1997.

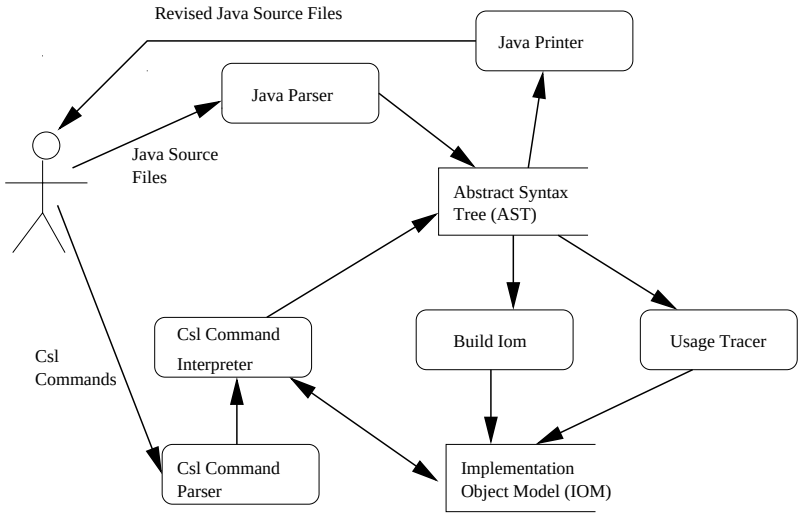


Figure 7: How STP works. The user submits a set of Java source files to the Java parser, which builds the abstract syntax tree (AST). The build iom visitor traverses the AST extracting model information into the Implementation Object Model (IOM). The usage trace visitor traverses the AST linking IOM constructs to their usages in method bodies. CSL commands are submitted to the CSL command parser, which forwards them to the CSL command interpreter. Preconditions for commands are checked using the IOM. The commands are carried out by updating both the IOM and the AST. Finally, the Java printer prints out the revised Java source files for the user.

[5] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] J. R. Levine, T. Mason, and D. Brown. *lex and yacc*. O'Reilly & Associates, Inc., 1990.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[8] S. Microsystems. *Java Object Serialization Specification*. Sun Microsystems, 1998.

[9] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for java. In *ACM Symposium on Principles of Programming Languages*, january 1997.

[10] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages*, january 1997.

[11] J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 146–161, Phoenix, 1991. ACM.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.

[13] S. Sankar. The java compiler compiler. available from Sun Microsystems.

[14] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1992.

[15] M. M. Werner. *Facilitating Schema Evolution With Automatic Program Transformations*. PhD thesis, Northeastern University, 1999. www.public.wit.edu/faculty/wernerm/thesis.ps.zip.

AUTOMATED PROTOTYPING TOOL-KIT (APT)

N. Nada, V. Berzins, and Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943,
{nnada, Berzins, luqi}@cs.nps.navy.mil
Ph. 831-656-4075
Fax 8310656-3225

Abstract

APT (Automated Prototyping Tool-Kit) is an integrated set of software tools that generate source programs directly from real-time requirements. The APT system uses a fifth-generation prototyping language to model the communication structure, timing constraints, I/O control, and data buffering that comprise the requirements for an embedded software system. The language supports the specification of hard real-time systems with reusable components from domain specific component libraries. APT has been used successfully as a research tool in prototyping large war-fighter control systems (e.g. the command-and-control station, cruise missile flight control system, patriot missile defense systems) and demonstrated its capability to support the development of large complex embedded software.

Keywords: APT, Automated Prototyping, Real-Time Systems, Command and Control, Formal Methods, Evolution, Reuse, Architecture, Components, PSDL

1 INTRODUCTION

Software project managers are often faced with the problem of inability to accurately and completely specify requirements for real-time software systems, resulting in poor productivity, schedule overruns, unmaintainable and unreliable software. APT is designed to assist program managers to rapidly evaluate requirements for military real-time control software using executable prototypes, and to test and integrate completed subsystems through evolutionary prototyping. APT provides a capability to quickly develop functional prototypes to verify feasibility of system requirements early in the software development process. It supports an evolutionary development process that spans the complete life-cycle of real-time software.

2 THE AUTOMATED PROTOTYPING TOOL-KIT (APT)

The value of computer aided prototyping in software development is clearly recognized. It is a very effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design. Bernstein estimated that for every dollar invested in prototyping, one can expect a $1.40 return within the life cycle of the system development [1]. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply [8]. Computer aid for rapidly and inexpensively

constructing and modifying prototypes makes it feasible [10]. The Automated Prototyping Tool-kit (APT), a research tool developed at the Naval Postgraduate School, is an integrated set of software tools that generate source programs directly from high level requirements specifications [7] (Figure 1).
It provides the following kinds of support to the prototype designer:

(1) timing feasibility checking via the scheduler,
(2) consistency checking and automated assistance for project planning, configuration management, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,
(3) computer-aided design completion via the editors,
(4) computer-aided software reuse via the software base, and
(5) automatic generation of wrapper and glue code.

The efficacy of APT has been demonstrated in many research projects at the Naval Postgraduate School and other facilities.

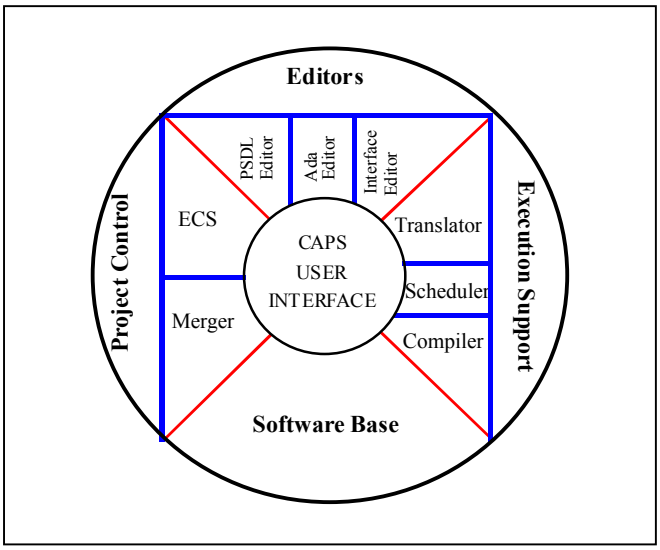


Figure 1. The APT Rapid Prototyping Environment

2.1 *Overview of the APT Method*

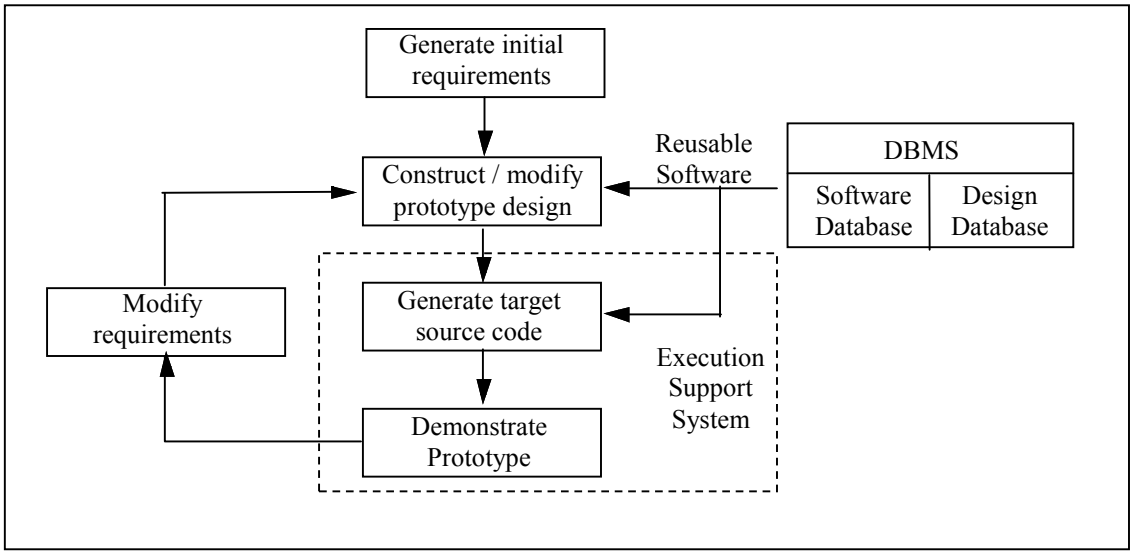


Figure 2. Iterative Prototyping Process in APT

There are four major stages in the APT rapid prototyping process: software system design, construction, execution, and requirements evaluation and/or modification (Figure 2).

The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either informally (e.g. English) or in some formal notation. These requirements may be refined by asking users to verify their completeness and correctness.

After some requirements analysis, the designer uses the APT PSDL editor to draw dataflow diagrams annotated with nonprocedural control constraints as part of the specification of a hierarchically structured prototype, resulting in a preliminary, top-level design free from programming level details. The user may continue to decompose any software module until its components can be realized via reusable components drawn from the software base or new atomic components.

This prototype is then translated into the target programming language for execution and evaluation. Debugging and modification utilize a design database that assists the designers in managing the design history and coordinating change, as well as other tools shown in Figure 3.

2.2 *APT as a Requirements Engineering Tool*

The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and

imprecise, but they are understood best by the customers. The lower levels are more technical, precise, and better suited for the needs of the system analysts and designers, but they are further removed from the user's experiences and less well understood by the customers. Because of the differences in the kinds of descriptions needed by the customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire software development process. APT provides the necessary means to bridge the communication gap between the customers and developers. The APT tools are based on the Prototype System Description Language (PSDL), which is designed specifically for specifying hard real-time systems [5, 6]. It has a rich set of timing specification features and offers a common baseline from which users and software engineers describe requirements. The PSDL descriptions of the prototype produced by the PSDL editor are very formal, precise and unambiguous, meeting the needs of the system analysts and designers. The demonstrated behavior of the executable prototype, on the other hand, provides concrete information for the customer to assess the validity of the high level requirements and to refine them if necessary.

2.3 *APT as a System Testing and Integration Tool*

Unlike throw-away prototypes, the process supported by APT provides requirements and designs in a form that can be used in construction of the operational system. The prototype provides an executable representation of system requirements that can be used for comparison during system testing. The existence of a flexible prototype can significantly ease system testing and integration. When final implementations of subsystems are delivered, integration and testing can begin before all of the subsystems are complete by combining the final versions of the completed subsystems with prototype versions of the parts that are still being developed.

2.4 *APT as an Acquisition Tool*

Decisions about awarding contracts for building hard real-time systems are risky because there is little objective basis for determining whether a proposed contract will benefit the sponsor at the time when those decisions must be made. It is also very difficult to determine whether a delivered system meets its requirements. APT, besides being a useful tool to the hard real-time system developers, is also very useful to the customers. Acquisition managers can use APT to ensure that acquisition efforts stay on track and that contractors deliver what they promise. APT enables validation of requirements via prototyping demonstration, greatly reducing the risk of contracting for real-time systems.

2.5 *A Platform Independent User Interface*

The current APT system provides two interfaces for users to invoke different APT tools and to enter the prototype specification. The main interface (Figure 3) was developed using the TAE+ Workbench [11]. The Ada source code generated automatically from the graphic layout uses libraries that only work on SUNOS 4.1.X operating systems. The PSDL editor (Figure 4), which allows users to specify the prototype via augmented dataflow diagram, was

implemented in C++ and can only be executed under SUNOS 4.1.X environments. A portable implementation of the APT main interface and the PSDL editor was needed to allow users to use APT to build PSDL prototypes on different platforms. We choose to overcome these limitations by reimplementing the main interface (Figure 5) and the PSDL editor (Figure 6) using the Java programming language [2].

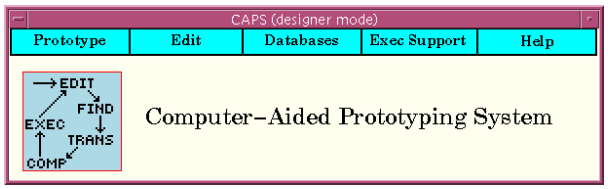


Figure 3. Main Interface of APT Release 2.0

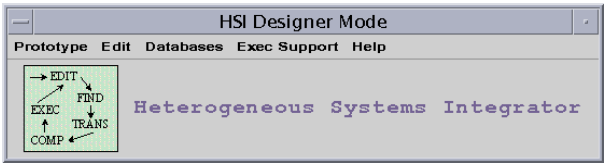


Figure 5. Main Interface of the new APT

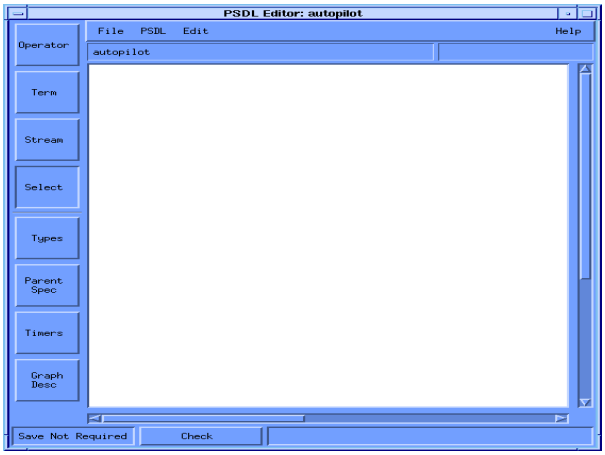


Figure 4. PSDL Editor of APT Release 2.0

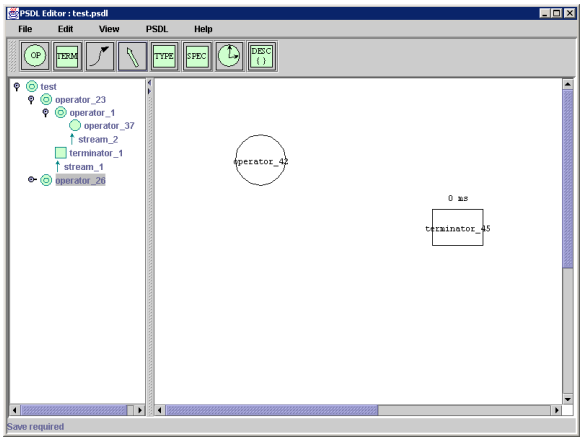


Figure 6. PSDL Editor of the new APT

The new graphical user interface, called the Heterogeneous Systems Integrator (HSI), is similar to the previous APT. Users of previous APT versions will easily adapt to the new interface. There are some new features in this implementation, which do not affect the functionality of the program, but provide a friendlier interface and easier use. The major improvement is the addition of the tree panel on the left side of the editor. The tree panel provides a better view of the overall prototype structure since all of the PSDL components can be seen in a hierarchy. The user can navigate through the prototype by clicking on the names of the components on the tree panel. Thus, it is possible to jump to any level in the hierarchy, which was not possible earlier.

3 A SIMPLE EXAMPLE: PROTOTYPING A C3I WORKSTATION

To create a first version of a new prototype, users can select "New" from the "Prototype" pull-down menu of the APT main interface (Figure 7). The user will then be asked to provide the name of the new prototype (say "c3i_system") and the APT PSDL editor will be automatically invoked with a single initial root operator (with a name same as that of the prototype).

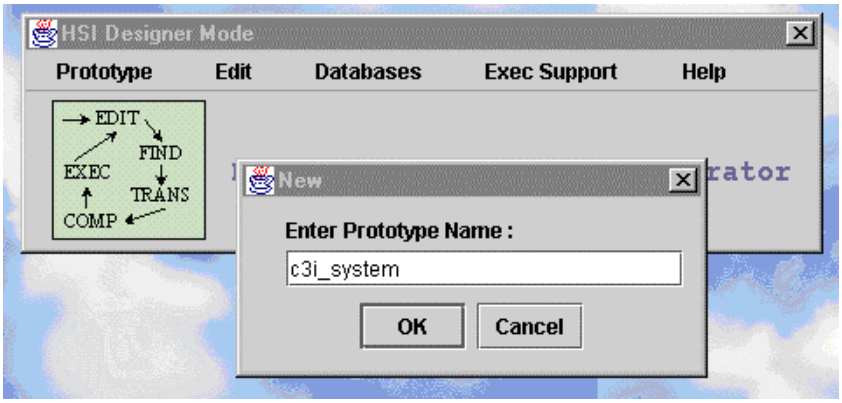


Figure 7. Creating a new prototype called C3I_System

APT allows the user to specify the requirements of prototypes as augmented dataflow graphs. Using the drawing tools provided by the PSDL editor, the user can create the top-level dataflow diagram of the c3i_system prototype as shown in Figure 8, where the c3i_system prototype is modeled by nine modules, communicating with each other via data streams. To model the dynamic behavior of these modules, the dataflow diagram is augmented with control and timing constraints. For example, the user may want to specify that the weapons_interface module has a maximum response time of 3 seconds to handle the event triggered by the arrival of new data in the weapon_status_data stream, and it only writes output to the weapon_emrep stream if the status of the weapon_status_data is damage, service_required, or out_of_ammunition. APT allow the user to specify these timing and control constraints using the pop-up operator property menu (Figure 9), resulting in a top-level PSDL program shown in Figure 10.

To complete the specification of the c3i_system prototype, the user must specify how each module will be implemented by choosing the implementation language for the module via the operator property menu. The implementation of a module can be in either the target programming language or PSDL. A module with an implementation in the target programming language is called an atomic operator. A module that is decomposed into a PSDL implementation is called a composite operator. Module decomposition can be done by selecting the corresponding operator in the tree-panel on the left side of the PSDL editor.

APT supports an incremental prototyping process. The user may choose to implement all nine modules as atomic operators (using dummy

components) in the first version, so as to check out the global effects of the timing and control constraints. Then, he/she may choose to decompose the comms_interface module into more detailed subsystems and implement the sub-modules with reusable components, while leaving the others as atomic operators in the second version of the prototype, and so on.

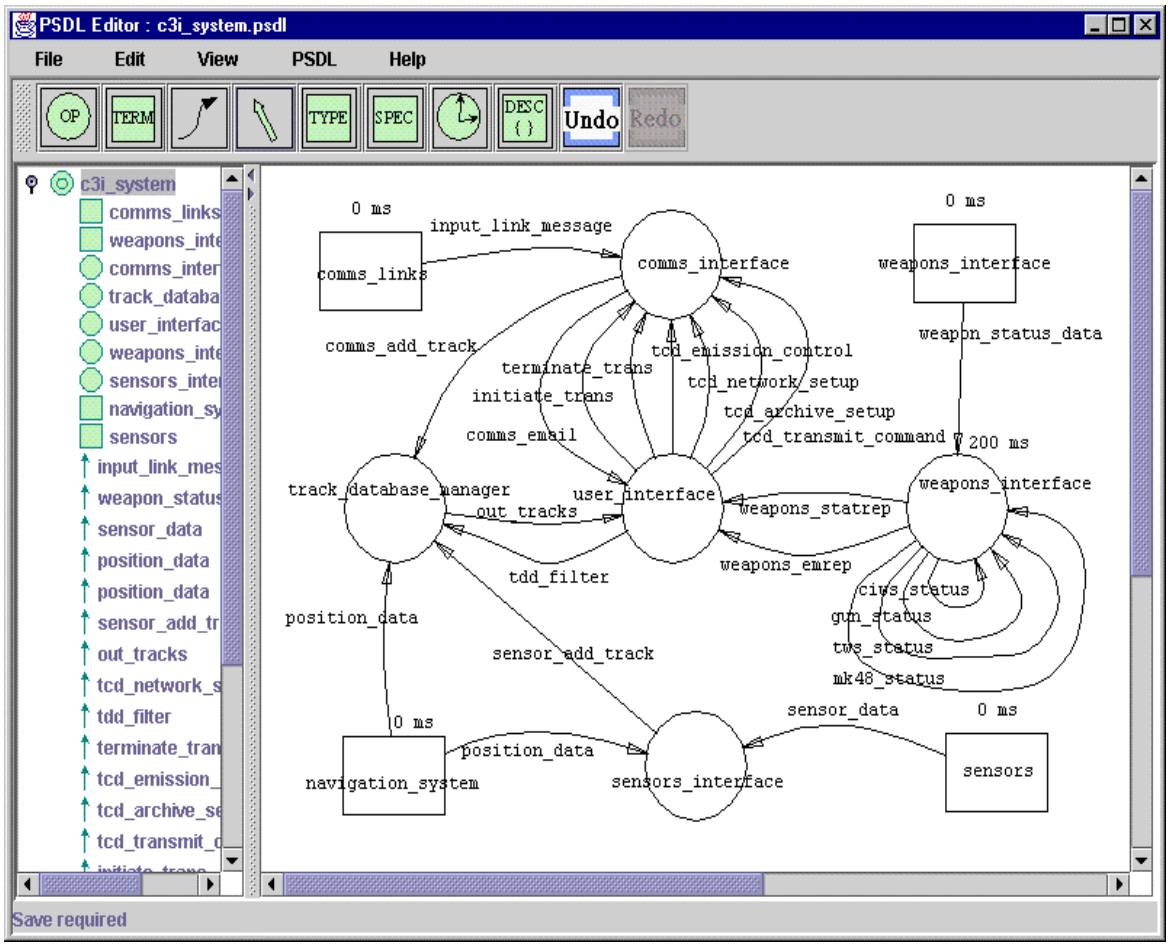
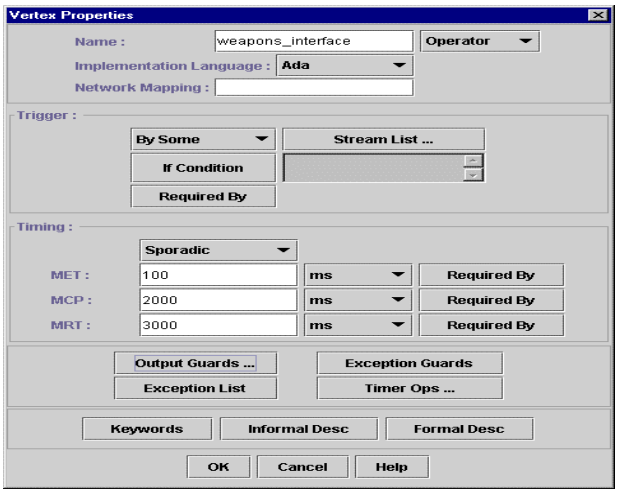


Figure 8. Top-level Dataflow Diagram of the c3i_system.



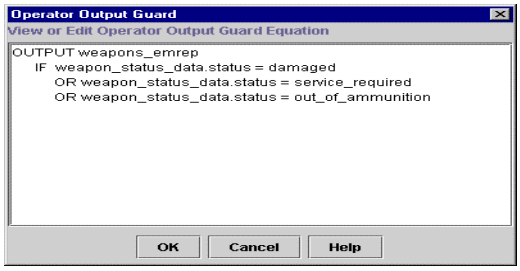
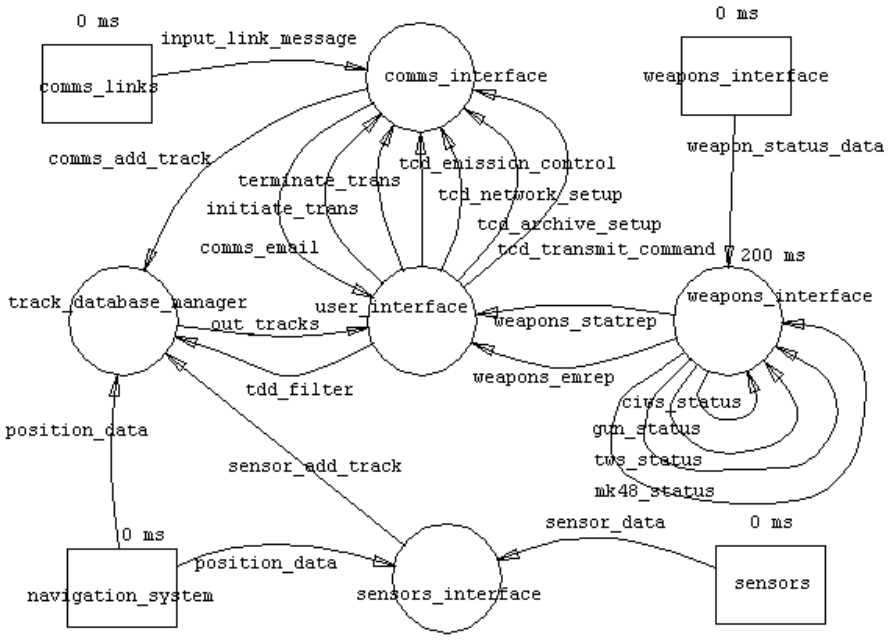


Figure 9. Pop-up Operator Property Menus

```
OPERATOR c3i_system
SPECIFICATION
   DESCRIPTION
      {This module implements a simplified version of
       a generic C3I workstation.}
END
IMPLEMENTATION
    GRAPH
```




```
    DATA STREAM
       -- Type declarations for the data streams in the graph go here.
    CONTROL CONTRAINTS
       OPERATOR comms_links
          PERIOD 30000 MS

       OPERATOR navigation_system
          PERIOD 30000 MS

       OPERATOR sensors
          PERIOD 30000 MS

       OPERATOR weapons_systems
          PERIOD 30000 MS

       OPERATOR weapons_interface
          TRIGGERED BY SOME
             weapon_status_data
          MINIMUM CALLING PERIOD 2000 MS
          MAXIMUM RESPONSE TIME 3000 MS
          OUTPUT
             weapons_emrep
             IF weapon_status_data.status =
                      damaged
             OR weapon_status_data.status =
                      service_required
             OR weapon_status_data.status =
                      out_of_ammunition
END
```

Figure 10. Top-level Specification of the c3i_system

To facilitate the testing of the prototypes, APT provides the user with an execution support system that consists of a translator, a scheduler and a compiler. Once the user finishes specifying the prototype, he/she can invoke the translator and the scheduler from the APT main interface to analyze the timing constraints for feasibility and to generate a supervisor module for each subsystem of the prototype in the target programming language. Each supervisor module consists of a set of driver procedures that realize all the control constraints, a high priority task (the static schedule) that executes the time-critical operators in a timely fashion, and a low priority dynamic schedule task that executes the non-time-critical operators when there is time available. The supervisor module also contains information that enables the compiler to incorporate all the software components required to implement the atomic operators and generate the binary code automatically. The translator/scheduler also generates the glue code needed for timely delivery of information between subsystems across the target network.

For prototypes which require sophisticated graphic user interfaces, the APT main interface provides an interface editor to interactively sculpt the interface. In the c3i_system prototype, we choose to decompose the comms_interface, the track_database_manager and the user_interface modules into subsystems, resulting in hierarchical design consisting of 8 composite operators and twenty-six atomic operators. The user interface of the prototype has a total of 14 panels, four of which are shown in Figure 11. The corresponding Ada program has a total of 10.5K lines of source code. Among the 10.5K lines of code, 3.5K lines comes from supervisor module that was generated automatically by the translator/scheduler and 1.7K lines that were automatically generated by the interface editor [9].

4 CONCLUSION

APT has been used successfully as a research tool in prototyping large war-fighter control systems (e.g. the command-and-control station, cruise missile flight control system, missile defense systems) and demonstrated its capability to support the development of large complex embedded software. Specific payoffs include:

(1) Formulate/validate requirements via prototype demonstration and user feedback
(2) Assess feasibility of real-time system designs
(3) Enable early testing and integration of completed subsystems
(4) Support evolutionary system development, integration and testing
(5) Reduce maintenance costs through systematic code generation
(6) Produce high quality, reliable and flexible software
(7) Avoid schedule overruns

In order to evaluate the benefits derived from the practice of computer-aided prototyping within the software acquisition process, we conducted a case study in which we compared the cost (in dollar amounts) required to perform requirements analysis and feasibility study for the c3i system using the 2167A

process, in which the software is coded manually, and the rapid prototyping process, where part of the code is automatically generated via APT [3]. We found that, even under very conservative assumptions, using the APT method resulted in a cost reduction of $56,300, a 27% cost saving. Taking the results of this comparison, then projecting to a mission control software system, the command and control segment (CCS), we estimated that there would be a cost saving of 12 million dollars. Applying this concept to an engineering change to a typical component of the CCS software showed a further cost savings of $25,000.

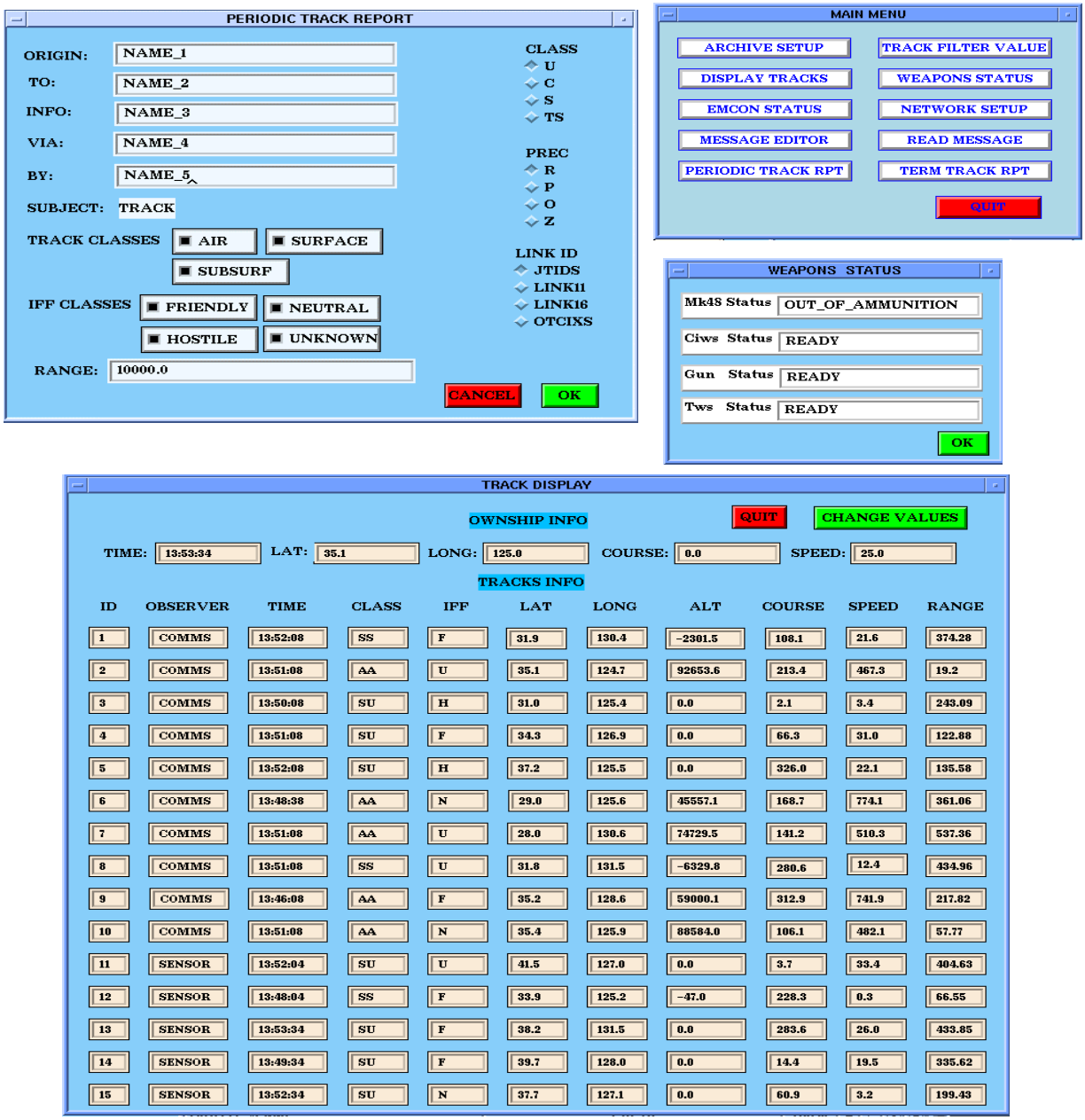


Figure 11. User Interface of the c3i_system

5 REFERENCES

[1] L. Bernstein, "Forward: Importance of Software Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 9-14, 1996.

[2] I. Duranlioglu, *Implementation of a Portable PSDL Editor for the Heterogeneous Systems Integrator*, Master's thesis, Naval Postgraduate School, Monterey, California, March 1999.

[3] M. Ellis, *Computer-Aided Prototyping Systems (APT) within the software acquisition process: a case study*, Master's thesis, Naval Postgraduate School, Monterey, California, June 1993.

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 1995.

[5] B. Kraemer, Luqi and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Transaction on Software Engineering*, 19(5), pp. 453-477, 1993.

[6] Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transaction on Software Engineering*, 14(10), pp. 1409-1423, 1988.

[7] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, 5(2), pp. 66-72, 1988.

[8] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer*, pp. 111-112, September 1991.

[9] Luqi, "Computer-Aided Prototyping for a Command-and-Control System Using APT", *IEEE Software*, 9(1), pp. 56-67, 1992.

[10] Luqi, "System Engineering and Computer-Aided Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 15-17, 1996.

[11] TAE Plus Programmer's Manual (Version 5.1). Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

SESSION 4

PANEL SESSION

Author Index

CoSET2000 Proceedings
ISBN 0 86418 725 4