SMILE/MARVEL: Two Approaches to Knowledge-Based Programming Environments

Gail E. Kaiser
Peter H. Feiler*

October 1986

CUCS-227-86

Abstract

This technical report consists of three related papers in the area of intelligent assistance for software development and maintenance. *Intelligent Assistance without Artificial Intelligence* describes SMILE, a software engineering environment that assists teams of programmers without using AI technology. *An Architecture for Intelligent Assistance in Software Development* presents an AI approach to generalizing the capabilities of SMILE. *Granularity Issues in a Knowledge-Based Programming Environment* briefly describes MARVEL, an intelligent assistant based on this AI approach, and compares it to SMILE.

The development and maintenance of SMILE is supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and in part by ZTI-SOF of Siemens Corporation, Munich, Germany. The designs of the architecture and MARVEL were conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA. *Dr. Feiler's address is the Software Engineering Institute, Carnegie-Mellon University, 580 South Aiken Avenue, Pittsburgh, PA 15232.

Intelligent Assistance without Artificial Intelligence

Gail E. Kaiser*
Columbia University
Department of Computer Science
New York, NY 10027

Peter H. Feiler
Carnegie-Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

28 August 1986

Abstract

SMILE is a distributed, multi-user software engineering environment that behaves as an intelligent assistant. SMILE presents a 'fileless environment', derives and transforms data to shelter users from entering redundant information, automatically invokes programming tools, and actively participates in the software development and maintenance process. Unlike other intelligent assistants, SMILE is not a rule-based environment: its knowledge of software objects and the programming process is hardcoded into the environment. We describe SMILE's functionality and explain how we achieved this functionality without reliance on artificial intelligence technology.

The development and maintenance of SMILE is supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and in part by ZTI-SOF of Siemens AG, Munich, Germany. *This paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

Final version to appear in Thirty-Second IEEE Computer Society International Conference, San Francisco, CA, February, 1987.

1. Introduction

In 1973, Winograd discussed his dream of an intelligent assistant for programmers [29]. More recently, artificial intelligence researchers have extended programming languages and environments (primarily Lisp environments) with knowledge about the relationships among program units [26] and the rules governing the software development process [3, 1, 22] in an attempt to turn the dream into reality. The resulting systems support 'exploratory programming' by an individual programmer very well [21], but they do not provide the assistance necessary to manage large-scale development and maintenance. However, as AI projects, such as 'expert systems', have become larger and commercially viable, researchers have turned their efforts towards developing this kind of assistance [11, 18], and we believe they will produce excellent results.

In the meantime, it is possible to build production-quality software engineering environments that provide seemingly intelligent assistance without requiring new breakthroughs in AI research. There is already (at least) one such system—the Software Management and Incremental Language Editing system (SMILE)—that provides seemingly intelligent, interactive support for teams of software developers and maintainers. SMILE does not use artificial intelligence techniques; it is not even written in Lisp. SMILE was written in C and runs on Unix™.

Although SMILE is several years old, it has not been discussed in the literature, except in acknowledgements by researchers who used it to develop their own systems). SMILE was developed by one of the authors, starting in 1979, originally as a tool for developing research prototypes for the Gandalf project [16]; it has been used extensively by both authors and by many others since 1980. SMILE has been relied on by the Gandalf and Gnome [7] projects at CMU and by the Inscape project [17] at AT&T Bell Labs; it has been distributed to at least forty sites. SMILE passes the crucial test of supporting its own maintenance. The purpose of this paper is to present the goals of SMILE and explain how they were achieved.

The original, high-level goals of SMILE were as follows.

- To hide the file system and the operating system from the users. SMILE presents a 'fileless environment'; that is, SMILE exposes its users only to the logical structure of the target software system. The normal alternative is for users to deal with the physical storage of the software in terms of directories and files, which often do not correspond nicely to the logical structure.
- To shelter the users from the tedious task of maintaining redundant information.

SMILE requires its users to enter each item of information only once; it automatically transforms the data as needed by tools. SMILE derives necessary information that can be calculated from the data supplied by users.

- To automate invocation of tools at appropriate points. SMILE assists the users by automatically performing trivial software development activities such as calling grep, lint, cc, make, and other Unix utilities [12] with the appropriate arguments at appropriate times. In some cases, the tool is invoked as soon as its input is ready; in other cases, the tool is not called until its results are required, such as to answer a user query or to provide input to another tool. SMILE hides the particularities of Unix and presents a uniform programming model different from the model imposed by Unix.
- To actively participate in the software development and maintenance process. SMILE is an interactive system, and all programming activities take place within the environment. In addition to calculating auxiliary information and automatically invoking tools, SMILE anticipates the consequences of user actions and automatically presents appropriate warning messages.
- To be sufficiently robust and reliable for supporting relatively large academic development projects. It automatically recovers from inconsistent states after user-initiated aborts and machine crashes; it also stores information redundantly to support recovery from disk errors or its own bugs.

All of these goals have been achieved. SMILE maintains source code, object code and other software development information in a database mapped onto the Unix file system. Knowledge of software objects and a model of the software development process are hardcoded into SMILE's commands. SMILE incorporates a large collection of Unix utilities, plus several special tools developed as part of the Gandalf research. SMILE has supported the simultaneous activities of at least seven programmers, and the largest software system developed and maintained in SMILE has approximately 61,000 lines of source code [13].

The following sections present the goals and achievements of SMILE in more detail. Section 2 explains SMILE's external architecture. Section 3 describes how SMILE assists individual programmers, while Section 4 describes the facilities oriented towards projects involving many programmers and long lifetimes. Section 5 discusses SMILE's implementation and current status. Section 6 compares SMILE to other software engineering environments. We conclude by summarizing the significance of SMILE as an example of intelligent assistance without artificial intelligence.

2. Architecture

SMILE is intended for use by small teams of programmers (5 to 20) developing and maintaining medium-size software systems (10,000 to 250,000 lines of source code) written in C, taking maximum advantage of the Unix file system and utilities.

2.1. GC

GC (Gandalf C) [28] is an enhancement of C that lists the types of formal parameters within the argument list (as in Pascal) and provides a module interconnection language (MIL). The MIL defines modules consisting of four types of source code objects (called *items*): procedures, variables, types, and macros. Each module has an import list indicating the items required from other modules and an export list indicating the items accessible to other modules. GC was adopted by the Gandalf project for all implementation efforts. SMILE supports GC, but automatically transforms source and header files from standard C to GC and *vice versa* as needed to import existing source code and to take advantage of C-specific programming tools. Throughout the rest of this paper, we mean "GC" when we say "C".

2.2. Databases

SMILE maintains all information about a software system in a *database* similar to the 'objectbases' of more recently developed programming environments [2, 15]. Each object has several attributes, representing auxiliary information, and is typed, enabling SMILE to provide object-oriented commands that apply type-specific tools.

A database consists of one or more 'projects', each representing a distinct software system. Most databases contain exactly one project, so we say 'database' and 'project' interchangeably. A project contains a number of 'modules' corresponding to GC modules. Each module contains a set of procedures, a set of variables, a set of type definitions, a set of macros, a list of import items, and a list of export items, as illustrated in Figure 2-1. The source text of procedures, variables, types, and macros are written in GC. Each module and item is attributed with status information, such as whether or not it has been compiled since it was last modified. Modules also contain object code, but this is never explicitly visible to users.

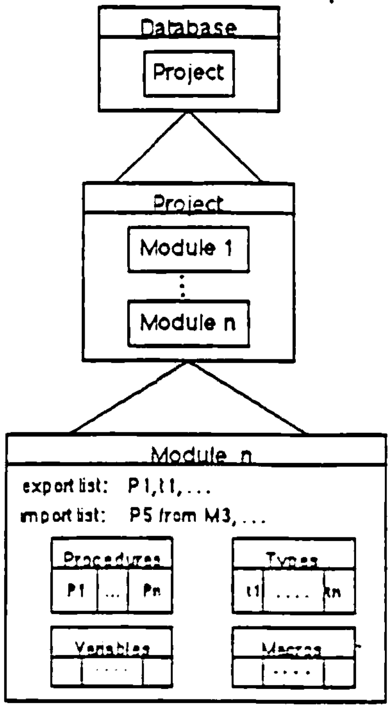

Figure 2-1: Database of Software Objects

2.3. User Interface

SMILE's user interface is script-oriented, and does not take advantage of windows or menus.[1] However, some tools included in SMILE, *e.g.*, screen-oriented editors, behave differently for bitmapped screens than for dumb terminals.

The user interface is 'friendly' and includes on-line help facilities. It is not necessary to remember either commands or arguments. The user can type a carriage-return after entering any part of the command line, and SMILE will prompt, one at a time, for remaining arguments; each prompt indicates a default value based on the user's most recent activities, and the user types a carriage-return to accept this default. If the user types "?" at any point, SMILE lists the currently valid alternatives according to the user's context. If the user instead enters "help", then SMILE explains the selected command and its argument. The user can also hit the interrupt key to any prompt to abort the current command. SMILE permits the user to abbreviate commands and arguments of commands to the shortest unambiguous form, and prompts with the possible choices if an abbreviation is ambiguous.

[1]The workstation implementations of SMILE do support windows. In particular, a user can modify a database in one window while browsing through the database in a second, read-only window; see Section 5.

3. Programming Assistance

SMILE assists individuals in writing programs. It maintains C source code, object code, and the status of these objects in its database, and automatically performs menial development activities. For example, it warns the programmer of the implications of changing the specifications of source code items, and it automatically recompiles after changes.

3.1. Browsing

SMILE helps the user navigate through a software system. The user selects a particular module—the user's *focus*—which is then indicated in SMILE's top-level prompt. SMILE assumes that further commands refer to this module and its contents until a different module is selected.

Browsing is object-oriented, in the sense that SMILE automatically invokes the appropriate viewing tool according to the type of the selected object. For C source code, this is normally a screen-oriented text editor; an earlier version of SMILE also provided a syntax-directed editor. Although SMILE assumes all commands are with respect to the current focus, it can shift focus automatically as the need arises. For example, if the user asks to **visit** an item that is not in the current module but is in some other module, SMILE changes the focus before invoking the appropriate viewer tool.

SMILE also supports general searches. A query can apply to an individual item, a module, or the entire database and SMILE can further filter the results of queries to display only items of a particular type (import item, procedure, variable, *etc.*) or only items that match some pattern. Pattern matching can be applied to the name of an item or to its source text, and a search can be applied to either the definition or usage of items, or SMILE can generate a full cross-reference table. The results of queries are displayed on the screen in the form of a transcript, which can be scrolled if SMILE is run from within a text editor that supports user shells. SMILE can also direct its answers to an external file or a printer. SMILE remembers past activities on a user-by-user basis; this supports, for example, a special option for the printer to spool only those items that have changed since last printed by the particular user.

3.2. Editing

SMILE creates and deletes modules and items within modules. If the user asks to remove an item that is in another module, SMILE requests confirmation before automatically changing focus to the other module to carry out the command. Thus, SMILE is forgiving of minor user errors. The **add** command requires the type of the new item; if this is not given, SMILE prompts for the missing argument. SMILE invokes the type-specific tool, and the low-level commands provided by the tool are used to construct the content of the item. If the user enters a command to write, save, save-and-exit, *etc.*, then the new item is stored in the database; if the user tells the tool to abort or exit (without saving), *etc.*, SMILE aborts the original **add** command. SMILE does this by monitoring the tool; no changes to the tools themselves are required.

Similarly, an existing item can be added or removed from the imports or exports list of the current module. When a new item is created, SMILE automatically asks the user whether or not it should be added to the exports list. When an item is removed from the exports, SMILE warns the user if it is imported by other modules and requests confirmation; if confirmation is given, it automatically removes these imports as well. When a user tries to delete an existing item, SMILE reminds the user if it is exported and requests confirmation before removing the corresponding item from the export list.

The user can make changes to items through the **edit** and **change** commands; SMILE invokes the appropriate editing tool. **Edit** restricts the user to making local changes to the body of an item, whereas **change** allows the user to make changes to both the specification and the body, which may have side effects on other items. For example, **edit** invokes the editor tool only on the body of a C procedure; if the tool supports multiple windows, then the header of the procedure is displayed for reference in another, read-only window. In the case of a C variable, **edit** permits the user to modify the initialization, but not the actual declaration. The **edit** command does not apply to types and macros, because any modifications can affect usages.

Sometimes changing the specification of an item has implications beyond those anticipated by the user. Therefore, SMILE always informs the user of potential problems before the damage is done. When the user selects the **change** command, SMILE queries its database to find all the other items that may be affected by the proposed change and informs the user of the extent of this change, in terms of how many other items might subsequently have to be modified to maintain consistency; it displays the actual dependencies on request. The user can abort or go ahead with the change with full knowledge of the implications.

3.3. Error Detection and Error Reporting

After a user adds, removes, or modifies an item, SMILE supplies rapid feedback regarding static semantic errors. The semantic analysis is applied only to the changed item rather than to other items affected by the change. SMILE propagates the change by updating the status information for dependent items. If the user requests it, SMILE submits these for analysis; this is explained in the following section.

The analysis is performed in a background process, so that the user does not have to wait for the tool to complete before continuing other activities. When processing completes, all error or warning messages are saved as an attribute of the current module (the focus), and the prompt is changed to indicate the errors. The user can ignore the errors, or ask SMILE to display the messages; thus, SMILE separates error detection from error reporting. Both the messages and the visual cue in the prompt remain until the user edits the offending item, so the user does not need to remember the particular errors or even the fact that there are errors within the particular module. It is less intrusive to indicate errors by appending a notice to the prompt than to display the errors themselves. An earlier version of SMILE dumped all the error messages on the user's screen as soon as the tool completed. This behavior was judged unacceptable because it interrupted the user's activities; the user was forced to read the messages then and remember them, because they were not stored.

3.4. Bookkeeping

SMILE maintains status information for each item. For example, each C item has a status field that indicates whether or not its static semantic analysis is up to date, whether the analysis was successful, or whether analysis is in progress in the background. SMILE maintains the correct value for the status field. Furthermore, SMILE automatically propagates changes to items by updating the status field of other items that might be affected by the change. The user can examine the status information for any item or display all items with a particular status. A user might use this information, for example, to request re-analysis of a particular item or all items affected by a change or to look for items that still have errors and need correction.

SMILE performs code generation by compiling at the granularity of a module. Therefore, it maintains a status field for each module indicating whether or not its object code is up to date, or being generated in the background. After compiling a module, SMILE indicates the resulting status in this status field. SMILE invalidates generated code by setting the status field accordingly under any one of several conditions:

- a new item is added to the module;
- an existing item is moved between modules, removed, edited or changed;
- an item is added to or removed from the importlist, and this item is actually referenced by an item of the module;
- an exported item is changed, and this item is imported into another module, where it is actually referenced by an item in the importing module.

3.5. Code Generation and Linking

SMILE recompiles modules and relinks the system as needed. It recognizes several alternative notions of 'as needed'. There is a tradeoff between recompiling immediately after a item of a module changes and delaying until the user requests system execution: Late recompilation requires the user to wait, but early recompilation may be wasted due to further changes to the same module; it also affects response time after each change. An earlier version of SMILE automatically recompiled as soon as an item changed, but recompiled only the item rather than the entire module. This was changed because the time and space overhead was unacceptable. The processing performed by the compilation tool after every modification led to slower response due to the cycles taken by the background job. Space was a problem because a separate object code file was generated for each item. SMILE now compiles an entire module rather than individual items. This optimization was done without affecting the interaction with the user.

SMILE automatically generates a makefile, including the appropriate command lines, and invokes make to construct an executable system. If a file name is given as an argument, the executable code is placed in this file; otherwise, standard Unix practice is followed and the output goes to the "a.out" file in the current working directory.

3.6. Modes

Modes permit the user to control and adapt SMILE's behavior. Users can set modes explicitly with a command or implicitly in their SMILE *profiles*. Every mode has a type and a default value. The boolean Autocompilation mode permits the user to indicate to SMILE whether it should temporarily refrain from automatically carrying out analysis and code generation. This is a desirable feature when the user starts making major changes to the application. Another boolean mode related to compilation indicates whether or not the compiler should generate more elaborate debugging information. The Verbose mode indicates the level of verbosity of SMILE's warnings and suggestions.

Somes modes are used to tailor SMILE to a particular operating environment. CMU mode permits SMILE to take advantage of some special CMU utilities. Home mode defines SMILE's home directory in the local file system, and Print mode names the local tool for spooling to the printer. SMILE is also tailored by the search paths and other environment variables defined in the user's Unix profile.

4. Development and Maintenance Assistance

SMILE assists software teams with their long-term development and maintenance activities. It coordinates simultaneous activities by multiple users, encourages reuse of existing code, and logs source code changes.

4.1. Reservations

SMILE prevents multiple users from modifying the same module at the same time by requiring the user to reserve a module before making changes to the module. If a user tries to modify a component of a module that is not reserved, SMILE explains that reservation is necessary. Only one user can reserve a module at a time. If another user attempts to reserve a previously reserved module, SMILE informs the user about who has reserved the module; users can also query reservation status explicitly.

SMILE helps users avoid making incompatible changes. If a user tries to change the specification of an exported item, SMILE checks to make sure that that all the modules that import this item are also reserved by the same user. If not, SMILE informs the user of their reservation status.

4.2. Experimental Databases

Reservations are always made with respect to a private workspace called an *experimental database*. Figure 4-1 shows the relationship between experimental databases and the *public database*, which contains the baseline version of the software system. The modules in the public database are available to all members of the software team, while the contents of an experimental database are private to its owner. An experimental database is a logical copy of the public database; SMILE employs a copy-on-write strategy to conserve space. Only modules reserved in the current experimental database can be modified. Additional modules can be reserved at any time, provided they are not already reserved by another user. SMILE automatically prelinks non-reserved modules (in a background process) to improve the response time of system generation.

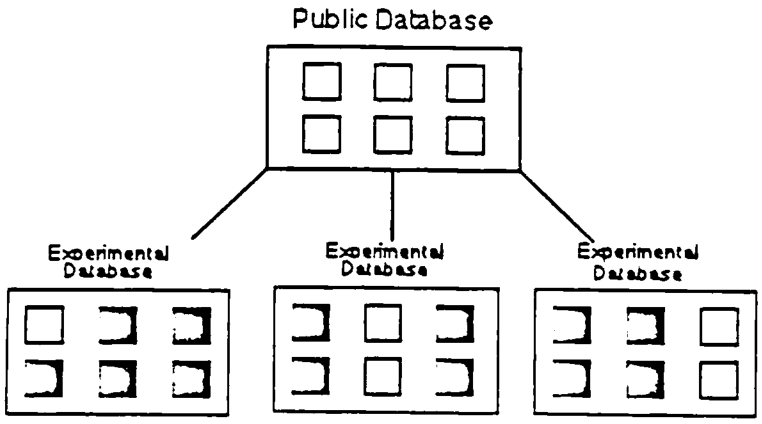

Figure 4-1: Experimental and Public Databases

When a user completes a set of changes, the user gives either the **update** or **deposit** command to return all the reserved modules to the public database. **Update** retains the reservations, so the user can make further changes, while **deposit** removes the reservations. In either case, SMILE makes the changes available to the rest of the software team by replacing the previous versions in the public database with the changed modules from the experimental database. SMILE permits users to back out of a proposed change by releasing the current reservations, so other users can reserve these modules in their original state.

At the beginning of an **update** or a **deposit**, SMILE checks the status of all reserved items to ensure that they have been analyzed and compiled successfully, without any errors. If there are inconsistencies, SMILE aborts the command; otherwise, SMILE locks the public database while it copies the modified objects back into the public database. Thus, **update** and **deposit** behave as transactions with respect to the public database.

4.3. Transactions

Every SMILE command is a *transaction*, in the sense that it is impossible to apply a second command within the same database until the first command terminates.[2] Background processing is coordinated in such a way that its results are recorded without conflicting with the transaction

[2]In the workstation implementations of SMILE, a user can access unaffected parts of a database in a read-only window during a transaction; see Section 5.

model. An earlier version of SMILE saved its internal state on disk after each transaction in order to record the changes in a fail-safe manner. This led to poor responsiveness when there were five or more simultaneous users in a time-sharing environment (on a VAX™ 780) and was discontinued.[3] Currently, SMILE saves state after the number of transactions indicated by the Checkpoint mode, and always saves state before and after commands that cause major changes, such as **change**, **update**, and **deposit**. A user can select full state saving by setting the Checkpoint mode to 1; alternatively, the user can explicitly give the **chkpoint** command after particularly crucial changes.

SMILE coordinates changes among the experimental databases owned by the members of a software project. A user can add a new module only within an experimental database, but SMILE records the addition in the public database to prevent another user from adding another new module with the same name. Similarly, SMILE records addition of import items in the public database, since another user may attempt to delete the item in a different experimental database. When the public database is locked during a transaction, other actions that affect the public database are blocked until completion of the transaction. Since **update** and **deposit** often take several minutes, blocked commands time out after thirty seconds and SMILE advises the user to try again later. This enables users to perform other development activities while they wait.

4.4. Change Logs

When programming teams are large, it is useful to maintain on-line change logs. Whenever a user updates or deposits the contents of an experimental database, SMILE prompts for a log entry for each modified module. SMILE automatically includes the user's name, the time/date, and the module name with the text provided by the user. Users can also append log entries for their reserved modules at any time. A user can query the entire log for a database or only the log for a particular module, and request entries since a particular date and/or by a particular user. SMILE prevents tampering with previous log entries, so a full audit trail of past changes is always available.

[3]This performance problem is reduced when SMILE runs in a distributed workstation environment; see Section 5.

4.5. Maintenance and 'Old Code'

As software systems become older, the modular structure tends to degenerate. Import and export lists grow and rarely shrink, even though an imported item is no longer used in the importing module and an exported item is no longer used outside the module, or even inside the module. SMILE assists users in restructuring old systems by moving items from one module to another and adjust the imports and exports accordingly, by adjusting the imports and exports throughout the database to reflect the actual interconnections determined by cross-references, and by detecting unused items.

SMILE provides facilities to bring externally developed 'old code' into a database, so it can assist future maintenance and enhancement activities. SMILE can also copy modules from one database to another. SMILE makes it easy to use software maintained outside of a SMILE database: Every module and every database may have a *prelude*, which lists external files and definitions of outside procedures; the corresponding object code files and Unix libraries are listed in SMILE library items. The **add**, **remove**, and **change** commands apply to libraries, as do the browsing facilities. The names of necessary libraries are given as arguments to the **build** command to incorporate them in an executable system. SMILE helps users create new Unix archives and libraries. It can produce a Unix archive from the C items in the database and can generate a single object code file that can be used as a library outside SMILE or within other SMILE databases.

5. Implementation

SMILE was originally called IPC, for Incremental Program Constructor, but the name was soon changed to SMILE. A prototype implementation was written in the Unix shell language during August 1979; it was used in September 1979 to bootstrap to a more advanced implementation in GC. These two versions ran on a PDP™ 11/70 under Unix Version 7.

SMILE was soon ported to a VAX (both 750 and 780) under Berkeley Unix, where it supported the intensive Gandalf prototype [10] implementation in 1980 and 1981 and the development and maintenance of the production-quality Gnome environment starting in 1982. SMILE was ported to the Sun Workstation™ in 1984 and to the MicroVAX™ workstation in 1985. The MicroVAX version is distributed by virtue of the Mach variant of Unix 4.3 BSD. The current implementation consists of 15,000 lines of GC source code, which is available on request from the Gandalf project at CMU.

Details

Although the original IPC was implemented in the Unix shell language, neither IPC nor the later versions of SMILE should be thought of simply as user shells. SMILE maintains its own database of all information about a software project and provides its own commands for carrying out development and maintenance activities; in effect, SMILE presents its own model of the programming process.

SMILE maps its database onto the Unix file system in a hierarchical manner. Each database corresponds to a directory, which contains a subdirectory for each project, which in turn contains a subdirectory for each module. Each module directory contains two files listing the imports and exports, respectively, and four subdirectories, one each for procedures, variables, types, and macros. The text of each item is stored in a separate file. This mapping to the file system is not visible to users. Cross-referencing information, status, and other derived attributes are maintained in a graph structure. This graph is dumped in binary form to a file within the database to persist between invocations of SMILE. A backup copy of the graph is also maintained, but if both the original and backup are corrupted, the graph can be regenerated from the database.

SMILE protects its users from operating system crashes, which might leave a database in an inconsistent state. SMILE automatically checks its database at the beginning of every session: If derived information such as error messages or object code has been lost, SMILE resets status information to make sure they are rederived. If the most recent session with this database was done using a previous version of SMILE itself, SMILE automatically reformats the graph structure and the database and adds default values for any new kinds of attributes. Approximately 30% of SMILE's source code is for disaster recovery and self repairs.

SMILE hides the Unix file system and its tools and utilities from its users, with the exception that it calls the user's favorite text editor. The default text editor at CMU is Emacs [9], but a different default can be substituted at each site. SMILE invokes lint to detect static semantic errors in source code objects, cc to compile modules, and make to generate executable systems. The variants of grep support SMILE's searches through source text and other objects.

We have not modified these tools; instead, SMILE automatically transforms objects into the format required by each tool. For example, SMILE combines the items of a module into a single file in the correct order for input to the compiler. This made it easy to port SMILE from one version of Unix to another and to use new tools as they became available (*e.g.*, lint replaced cc for static semantic analysis in 1982) without these changes being visible to the users. We believe it would not be difficult to port SMILE to a non-Unix operating system, providing it supplied similar tools; the only local tools that are mandatory are a text editor, a C compiler, and a linker.[4]

6. Related Systems

SMILE is similar to knowledge-based programming environments, advanced programming language environments, language-based editors and software engineering environments. In the following paragraphs, we describe the advantages and disadvantages of these systems with respect to SMILE.

Knowledge-Based Environments: The CommonLisp Framework (CLF) [6], Refine™ [22] and other knowledge-engineering environments can provide SMILE-like automation via condition/action rules [5]. However, they cannot recognize the alternative results of actions, *e.g.*, the compiler may terminate successfully, producing object code, or unsuccessfully, producing error messages. None of these environments support multiple simultaneous users. On the other hand, SMILE is not extensible, so it is not as easy to add new kinds of objects and new tools.

Language Environments: Advanced programming languages such as Interlisp [26], Loops [23] and Smalltalk-80™ [8] include run-time environments that are indistinguishable from single-user programming environments. Although they provide SMILE-like facilities, these are strongly tied to the programming language. The implementation of SMILE is specific to the GC, but it would not be very difficult to reimplement for another conventional programming language, provided corresponding tools were available. However, language environments can integrate debugging facilities with the other tools.

Language-Based Environments:[5] Language-based environments add many of the advantages of language environments to conventional languages such as Pascal. The Synthesizer [25] and Pecan [19] are examples of specific environments, while the Synthesizer Generator [20] and Gandalf are systems for generating such environments from formal descriptions. Most language-

[4]Early versions of SMILE used only the tools and utilities provided by Unix, but recent versions also include special tools for language-oriented programming environments. These tools are not relevant to the facilities described in this paper.

[5]We use the term 'language-based environment' synonymously with 'language-based editor', 'structure-oriented environment', 'structure editor-based environment', 'syntax-directed editor', *etc.*

based environments provide advanced user interfaces with menus and pointing devices, and perform various activities in response to programmer actions, but they are unable to anticipate the potential results of actions and warn users before the damage is done. The practicality of these environments is limited, since the entire software system is maintained as a single abstract syntax tree; further, it is difficult to incorporate existing programming tools into these environments.

Software Engineering Environments: SMILE is most similar to Cedar [27], DSEE™ [14], Arcadia [24] and other large-scale environments for software development and maintenance. Like SMILE, these environments provide an interface between programming tools and the user on the one hand, and between programming tools and the software database on the other. Such environments typically provide more advanced version control and project management facilities than SMILE, but they leave individual programmers to the standard edit/compile/debug cycle supported by traditional tools.

7. Conclusions

SMILE's primary contribution is the apparently intelligent assistance that spans both the activities of individual programmers and the coordination of multiple programmers. SMILE provides this assistance by

- maintaining all information about a software project in a database;
- integrating Unix tools into a new model of development and maintenance that hides the particularities of Unix tools;
- actively participating in the development and maintenance processes by deriving data when possible from previously stored information, automating the invocation of these tools and anticipating the consequences of tool processing;
- imposing a structure on software development activities that permits it to 'know' what the programmers are doing at all times, to 'infer' what they are likely to do next, and to 'judge' what it can appropriately do for them;
- recovering from external and internal failures and repairing its databases automatically, making it sufficiently robust and reliable for production use.

SMILE provides this assistance without a knowledge base of rules describing the software development process. Instead, certain 'common sense' about software development activities has been programmed directly into the environment, resulting in a production-quality intelligent assistant that several projects have relied on to develop and maintain their software.

Acknowledgements

In addition to the authors, other past and present members of the Gandalf Project at Carnegie-Mellon University have been active in SMILE's evolution over the past seven years. Raul Medina-Mora and David Notkin were involved in the early development. Barbara Denny, Bob Ellison and Charlie Krueger have been responsible for maintaining and enhancing SMILE at various times. Many other people have developed tools that are presently incorporated into SMILE. Nico Habermann is the principal investigator of the Gandalf Project.

References

[1] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[2] Robert M. Balzer.
Living in the Next Generation Operating System.
In *Proceedings of the 10th World Computer Congress (IFIP Congress '86)*. Dublin, Ireland, September, 1986.
To appear as a book published by Springer-Verlag.

[3] David R. Barstow and Howard E. Shrobe.
From Interactive to Intelligent Programming Environments.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984, pages 558-570.

[4] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[5] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin.
Programming Expert Systems in OPS5.
Addison-Wesley Publishing Co., Reading, MA, 1985.

[6] CLF Project.
Introduction to the CLF Environment.
March, 1986.
USC Information Sciences Institute.

[7] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Pittsburgh, PA, April, 1984.

[8] Adele Goldberg and David Robson.
Smalltalk-80 The Language and its Implementation.
Addison-Wesley Publishing Co., Reading, MA, 1983.

[9] James A. Gosling.
Unix Emacs
Carnegie-Mellon University, Department of Computer Science, 1982.

[10] Gail E. Kaiser, Robert J. Ellison, David B. Garlan, David S. Notkin and Steven Popovich.
Gandalf User Manual and Tutorial.
In *The Second Compendium of Gandalf Documentation*. Carnegie-Mellon University, Department of Computer Science, 1982.

[11] Beverly L. Kedzierski.
Knowledge-Based Project Management and Communication Support in a System Development Environment.
In *Proceedings of the 4th Jerusalem Conference on Information Technology*. Jerusalem, Israel, May, 1984.

[12] Brian W. Kernighan and John R. Mashey.
The UNIX Programming Environment.
Software — Practice and Experience 9(1), January, 1979.
Appears in IEEE Computer, 12(4), April 1981 and in [4].

[13] Charlie Krueger.
Private communication.
August, 1986
Regarding largest system (ALOE) maintained in SMILE.

[14] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[15] John R. Nestor.
Toward a Persistent Object Base.
In *Proceedings of the IFIP WG 2.4 International Workshop on Advanced Programming Environments*. June, 1986.
To appear as a book published by Springer-Verlag.

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

[17] Dewayne E. Perry.
Position Paper: The Constructive Use of Module Interface Specifications.
In *Third International Workshop on Software Specification and Design*. London, England, August, 1985.

[18] C.V. Ramamoorthy, Vijay Garg and Rajeev Aggarwal.
Environment Modelling and Activity Management in Genesis.
In *Proceedings of SoftFairII: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, pages 2-10. December, 1985.

[19] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Pittsburgh, PA, April, 1984.

[20] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Pittsburgh, PA, April, 1984.

[21] B. A. Sheil.
Power Tools for Programmers.
Datamation Magazine , 1983.
Reprinted in [4].

[22] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[23] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[24] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wiledon and Michal Young.
Arcadia: A Software Development Environment Research Project.
In *2nd International Conference on Ada Applications and Environments*. IEEE Computer Society, Miami Beach, FL, April, 1986.

[25] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Reprinted in [4].

[26] Warren Teitelman and Larry Masinter.
The Interlisp Programming Environment.
IEEE Computer 14(4):25-34, April, 1981.
Reprinted in [4].

[27] Warren Teitelman.
A Tour Through Cedar.
IEEE Software 1(2):44-73, April, 1984.
Also appears in Proceedings of the Seventh International Conference on Software Engineering, 1984.

[28] Walter F. Tichy.
Software Development Control Based on Module Interconnection.
In *4th International Conference on Software Engineering*. September, 1979.

[29] Terry Winograd.
Breaking the Complexity Barrier (Again).
In *Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, pages 13-30. Gaithersburg, MD, November, 1973.
Reprinted in [4].

An Architecture for Intelligent Assistance in Software Development

Gail E. Kaiser*
Columbia University
Department of Computer Science
New York, NY 10027

Peter H. Feiler
Carnegie-Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

14 August 1986

Abstract

We define an architecture for a software engineering environment that behaves as an intelligent assistant. Our architecture consists of two key aspects, an objectbase and a model of the software development process. Our objectbase is adapted from other research, but our model is unique in that is consists primarily of rules that define the preconditions and multiple postconditions of software development tools. Metarules define forward and backward chaining among the rules. Our most significant contribution is *opportunistic processing*, whereby the environment automatically performs software development activities at some time between when their preconditions are satisfied and when their postconditions are required. Further, our model defines strategies that guide the assistant in choosing an appropriate point for carrying out each activity.

*The research presented in this paper was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

1. Introduction

In 1973, Winograd [30] presented his dream of an intelligent assistant for programmers that would understand what it does: It would be based on an explicit model of the programming world. Winograd described an imaginary programming environment, A, that would assist programmers by providing early error checking, by answering questions about the program and the interactions among program parts, by handling trivial programming problems, and by automating simple debugging tasks.

Artificial intelligence research has moved closer to achieving this dream by developing a *knowledge-based* approach to programming, which includes relationships among program units, both in the abstract and with respect to a particular target system. The Masterscope package of Interlisp [27] and the CommonLisp Framework (CLF) [5] maintain cross-referencing information to answer queries about interactions among program units. Also, CLF's knowledgebase understands the abstract relationships among program units. For example, CLF 'knows' that a system consists of modules and individual software objects (functions, variables, *etc.*) and that object classes have particular properties, such as a maintainer and whether it has been compiled.

A knowledge-based programming environment also includes the rules governing the software development process. For example, wide-spectrum languages such as V [23] and Gist [1] have been augmented with rules that aid the programmer in translating from higher- to lower-level specifications and from specifications to executable code. CMS [13] provides a formal representation of the software project model, time (for scheduling), and software development activities. For example, if a 'capability' is a desired feature of the target system, then it must be realized by a 'component' of the system; a 'task' must be defined to specify who is in charge of the component and when it is due. Genesis's Activity Manager [19] provides similar facilities.

While work in AI was progressing, researchers in traditional software were addressing their version of Winograd's dream. *Tools* were developed that automated certain aspects of the programming process. For example, Make [7] automatically rederives an executable system when part of the source code changes. SCCS [22] requires programmers to reserve modules for change, thus ensuring orderly software evolution. RCS [28] supports multiple versions of software objects and manages separate lines of development.

Collections of tools were integrated into interactive programming environments that support a particular programming language. The Synthesizer [26] combines language-oriented editing that prevents syntactic errors with immediate feedback about static semantic errors; it also permits programmers to interleave execution and debugging with editing. The Gandalf Prototype [16] added a module interconnection language with incremental, intermodule consistency checking to a C programming environment similar to the Synthesizer.

Unfortunately, the knowledge-based approach and the tools approach have progressed more-or-less independently. The individual tools incorporate a small bit of knowledge about a particular programming problem, but this knowledge cannot be augmented. The programming environments are hardcoded with a particular view of the software development process that defines the interaction between the programmers and the target system, but this knowledge is not available to the users of the environment. The knowledge-based environments are much more general, but represent only a fraction of human expertise about software development and maintenance.

Working within the tools approach, the members of the Gandalf project (including the authors) developed a distributed, multi-user software engineering environment called SMILE [24, 12], which is relatively close to achieving intelligent assistance. SMILE presents a 'fileless environment' to its users, answers queries, and automatically invokes various tools. However, SMILE's knowledge of software objects and the programming process is hardcoded into the environment.

Our experience with SMILE provided insights into the development of practical environments and convinced us that a generalization of SMILE's internal architecture would aid in developing an intelligent assistant for software development and maintenance. Our architecture for intelligent assistance combines tools with knowledge. From the tools approach, we gain the years of experience of other computer scientists building and using particular tools and environments. From the knowledge-based approach, we gain a suitable structure for choosing among tools and automating the invocation of tools. Our architecture defines a basis for intelligent assistance that consists of two key aspects: an *objectbase* and a *model of the software development process*.

The objectbase maintains all software objects, including tools, and provides the environment with insight into the various classes of objects and the relationships among objects. For example, one object is a component of another, and a particular object may be applied to another object to produce a third.

The model imposes a structure on programming activities. It consists of an extensible collection of rules that specify the particular conditions that must exist for particular tools to be applied to particular software objects. Metarules permit the environment to understand the rules and support *opportunistic processing*, where the environment performs activities when it knows the results of these activities will soon be required by its users. Opportunistic processing is the primary focus of this paper.

In this paper, we explain our architecture and how it meets certain fundamental requirements for supporting a software engineering environment that understands what it does. Section 2 presents the basis for intelligent assistance defined by our architecture. Section 3 describes how an intelligent assistant built on this framework can perform software development activities automatically to provide intelligent assistance to its users. Section 4 briefly describes our implementation.

2. A Basis for Intelligent Assistance

An intelligent assistant should understand what it is doing [30]. Most software tools are "moronic assistants" that know what they are doing, but do not understand the purpose of the objects they manipulate or how their tasks fit into the software development process. In other words, they may know the 'how' but not the 'why'.

For example, Make has a simplistic world model consisting of files and command lines. A 'makefile' defines dependencies among files and gives the command lines necessary for restoring consistency among dependent files. Make's notion of consistency is based on files and time: If the timestamp of an input file is later than the timestamp of an output file, then the indicated command line should be passed to the Unix™ shell. Make is used widely for generating a new executable version of a system after source files have been modified.

However, Make's 'knowledge' is primitive. Its objectbase consists of files that have a single attribute, their timestamp. Make knows nothing about applying tools to files: it just handles command lines as indivisible units. Make does not understand source files *vs.* object files, modules *vs.* systems, programmers or programming.

To give Make this knowledge we could define a notion of an *object*, which belongs to a *class*, such as 'system' or 'module'. Each class would define the *attributes*, or properties, of its objects. A 'module object code' object might have a 'history' attribute describing its generation and a 'derivation of' attribute pointing to the object representing the corresponding source code.

We could then define *rules* that model the part of the software development process relevant to Make. One rule might be that a 'programmer' object can modify a 'module' object; another rule might state that after such a modification, the 'module' object is no longer consistent with its 'derivation' attribute and there is an obligation to restore this consistency. A third rule might state that a precondition for a 'programmer' to test a 'system' is that all 'module object code' objects that are components of the corresponding 'executable system' must be consistent with their 'module'.

Given this knowledge, Make could be considered to be relatively more intelligent. Make would be easier to integrate with other tools that support configuration management, version control, task management, *etc.*, assuming these tools had similar knowledge of software objects and their roles in the development process.

We believe that an objectbase and a model of the development process are prerequisites to intelligent assistance. An assistant cannot understand why it performs particular activities unless it knows

- the properties of the objects it manipulates,
- the capabilities of certain objects (programmers and tools) to manipulate other objects,
- the preconditions required by each activity,
- the postconditions of each activity.

Therefore, our architecture specifies a general objectbase and an extensible collection of rules describing the preconditions and postconditions of activities, as well as hints and strategies that determine the degree of the environment's contributions. We briefly describe the objectbase here; the rules, hints and strategies are the topic of the following section.

Objectbase

We considered several possible forms for our objectbase. One possibility was the entity-relation-attribute model proposed for Genesis [20]. However, weaknesses of relational databases make them inadequate for software engineering environments [15]. To maximize flexibility, we chose an objectbase similar to those of object-oriented programming languages, such as Loops [25]. In particular, we adopted their support for multiple inheritance and active values. Unlike most such languages, however, we require a 'persistent' objectbase, one that retains its state across invocations of the environment. The same concepts are found in the objectbases supported by other knowledge-based environments, such as AP3 [2] and Refine™ [23].

In our objectbase, each object is an instance of a class, which defines certain attributes of each object and inherits other attributes from its superclass(es). Some attributes define the relationships among objects; others trigger activities when accessed and/or updated. The activities applicable to a class are defined as methods for the class.

This enables an intelligent assistant to expose its users only to the logical structure of the target software system. The environment consists of a set of typed, interconnected software objects representing the system and its history. The interconnections among software objects represent the logical structure of the system. Object types include module, procedure, type, design description, user task (or development step), user manual, *etc.* Typing permits the assistant to provide an object-oriented user interface similar to the Smalltalk-80™ environment [10], where the environment makes available to each user only those commands that are relevant to the object under consideration.

3. Opportunistic Processing

The objectbase also maintains the rules that model the software development process. These rules provide the meta-knowledge required for an environment to apply tools automatically. We call this behavior *opportunistic processing*, which offloads simple activities onto the intelligent assistant—menial activities, such as invoking the compiler and recording any errors found during compilation. This approach contrasts with some intelligent assistance systems, such as the Programmer's Apprentice (KBEmacs) [29] and CHI (previously PSI) [23], which focus on the separate problem of automatic programming.

3.1. Rules

We represent our model as a collection of rules similar to the production rules of Ops5 [4] in that each rule has a condition and action. When the condition is true, the action may be executed. Our rules differ from production rules in that the action is divided into two parts, an activity and a postcondition. Because our rules have postconditions, we refer to the original conditions as preconditions.

The *activity* part of a rule represents an integral software development task. For example, "compile module" is one activity and "change component" is another (a 'component' is a facility defined within a module, such as a procedure, a variable, a type, *etc.*). The specific editing commands applied during the course of the "change component" activity are not considered activities. "Fix bug" is not an activity, since it involves many tasks, perhaps involving several users. Thus our notion of an activity represents a middle-ground granularity.

Each activity is associated in the objectbase with a tool that performs the activity. One attribute of each tool is whether it can be invoked by the environment without human intervention. For example, the "compile module" activity is associated with the compiler, which can be applied by the intelligent assistant; the "edit component" activity is associated with a text editor (or a syntax-directed editor), which requires human interaction.

The *precondition* part of a rule — a boolean expression — must be true before an activity can be performed. The operands of a precondition include software objects and the attributes of software objects. For example, "notcompiled(module)" might be an appropriate precondition for the "compile module" activity. Another precondition for "compile module" would be "for all components c such that in(module, component c): analyzed(component c)", where "analyzed(c)" is true only if a static semantic analysis of component *c* finds no errors. An activity may have multiple preconditions that must be satisfied.

A *postcondition* becomes true when an activity is completed. Both preconditions and postconditions are written as well-formed formulas (wffs) in the first order predicate calculus. Our rules are based on Hoare's assertions [11], where a programming language construct is associated with its preconditions and postconditions; if the preconditions are true before the language construct is executed, then the postconditions will be true afterwards.

However, a programming activity may have multiple postconditions, exactly one of which is true after the activity terminates. Which of the various possibilities is true can be determined only by invoking the corresponding tool. For example, two postconditions for the "compile module" activity might be "compiled(module)" and "errors(module)". Here we follow the extension of Hoare's assertions proposed by Perry [18], where there must be multiple postconditions to represent the exceptional results of executing a procedure. This notion of postconditions distinguishes our architecture from CLF, Genesis' Activity Manager, and other expert systems that rely on condition/action rules. The most important advantages are that we can separate an activity from its results and therefore consider several alternative results within our model.

Two example rules are given in Figure 3.1. The first states the preconditions and the two postconditions for the "compile module" activity. The preconditions are given first, followed by the activity (within braces), followed by the postconditions. The alternative postconditions are separated by semicolons.

3.2. Meta Rules

Our architecture supports the definition of metarules that guide the intelligent assistant's use of rules. One metarule states that if the preconditions of an activity are satisfied, and the activity can be performed by the assistant, then the assistant may perform the activity automatically. Consider the first rule in Figure 3.1. The metarule interprets this rule to mean that the assistant may compile any modules *M* such that all the components of *M* have been analyzed but *M* has not been compiled.

```
notcompiled(module) and
   for all components c such that in(module, component c):
       analyzed(component c);
     { compile module }
compiled(module);
errors(module);

equals(module, focus(userid)) and in(module, component);
     { edit component }
notanalyzed(component) and notcompiled(module);
```

Figure 3-1: Compile Rule and Edit Rule

In this example, "notcompiled(module)" is one of the preconditions to the "compile module" activity; "errors(module)" is included as one of the possible postconditions. If the previous compilation failed, "errors(module)" will be true. The "compile module" activity cannot be performed when "errors(module)" is true, because its preconditions cannot be satisfied. If a user then edits a component, perhaps to fix the error, the second rule of Figure 3.1 states that "notcompiled(module)" will be set to true and the metarule permits compilation.

Importantly, this metarule states that the intelligent assistant may perform an activity when preconditions are satisfied; it does not state that the assistant must perform the activity as soon as the preconditions are true, or at any time thereafter. However, the intelligent assistant may apply the tool and use *forward chaining* to determine additional activities whose preconditions are satisfied by the postconditions of the first activity. Therefore, we call this metarule the 'forward chaining metarule'.

Forward chaining supports behavior similar to language-oriented editors, such as the Synthesizer and Gnome [8]. When the user makes a subtree replacement in the abstract syntax tree representing the program, the editor automatically performs several actions. In the case of editors generated from attribute grammars [21], the editor automatically re-evaluates the values of attributes whose values may have changed. These attributes might represent the content of the symbol table and the object code for the program. Other editor generators automatically invoke action routines for type checking or code generation for modified program parts [6].

A second metarule states that if a user invokes a tool with unsatisfied preconditions, the intelligent assistant should use *backward chaining* to find activities it can perform whose postconditions might satisfy the preconditions of the activity requested by the user. In this case, the metarule states that the intelligent assistant must exhibit this behavior. We call this metarule the 'backward chaining metarule'.

Backward chaining supports behavior similar to Make, DSEE^{TM} [14], Toolpack [17] and other software engineering tools in which a user may request regeneration of an executable system after changes have been made to its source code. The environment uses dependency information previously supplied by the software development team to determine which source files to recompile.

Sometimes our intelligent assistant attempts backwards chaining, but finds that the preconditions cannot be satisfied; in this case, the user is informed of the problem. The intelligent assistant is not expected to, for example, correct source code so that it will compile successfully. For example, our intelligent assistant might support a large team where multiple users should not change the same module simultaneously. Here, each user must reserve a module before changing it. The preconditions and postconditions for the "reserve module" activity are stated in the first rule shown in Figure 3-2 ("saved(module)" is true when the module has been saved. by the version control tool), and the second rule states that the "change component" activity cannot be performed unless the module containing the component is reserved.

```
not reserved(module) and saved(module);
    { reserve module }
reserved(module, userid);

reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

for all components k such that in(module, component k)
    and uses(component k, component c):
        reserved(module, userid);
    { change component c }
```

Figure 3-2: Change Rules and Reserve Rule

The "change component" activity permits the user to modify the specification of a component ("edit component" permits the user to modify only the body). The third rule of Figure 3-2 states that the containing module must be reserved along with any other modules that depend on it (*c* and *k* distinguish multiple objects of the same type). The backward-chaining metarule enables our intelligent assistant to automatically reserve modules whose components may have to be modified to restore consistency with the changed component. The metarule also prevents the user from modifying the specification of a component when dependent modules cannot be reserved (according to the first rule).

3.3. Strategies and Hints

We chose the name 'opportunistic processing' for these chores because the assistant may perform an activity as the opportunity arises any time after its preconditions are satisfied and before another activity whose preconditions depend on its postconditions. Rules may be tagged so their activities are performed immediately after their preconditions are satisfied (*i.e.*, forward chaining applies) while other activities are performed only when their postconditions are required (forward chaining does not apply). Since we need to choose other points on this spectrum, we have included hints and strategies in our model to aid the intelligent assistant in making decisions.

A *hint* is similar to a rule, but without postconditions. The preconditions of a hint are used to guide the intelligent assistant in choosing when to apply a tool whose other preconditions are satisfied. Consider again the first rule from Figure 3.1. Suppose we do not want the assistant to compile a module, even though the preconditions are satisfied, while a user with modification rights is browsing through the module: The user may decide to change some components of the module, and the compilation will have been wasted. So we use a hint, Figure 3-3, giving this precondition for the "compile module" activity (angle brackets are used for parentheses). When the assistant follows a strategy including this hint, compilation is delayed until the user changes to another module.

```
not reserved(module) or
    < reserved(module, userid) and
      not equals(module, focus(userid)) >
    [ compile module ]
```

Figure 3-3: Compile Hint

Since we want the human user to be able to invoke the compiler without changing to another module, we give this precondition to "compile module" in a hint, rather than as part of a rule. Hints apply only to the opportunistic processing of the intelligent assistant, not to activities initiated by a human user. In other words, hints are considered during forward chaining and ignored during backward chaining.

A *strategy* consists of a collection of hints and rules, which apply only when the strategy is in force. The third (and currently final) metarule from our model enables the intelligent assistant to employ strategies by combining its rules and hints with the rules normally considered. Zero or more strategies may be employed at the same time. When this results in more than one rule for the same activity, all their preconditions must be satisfied; only one set of postconditions is permitted.

Currently, our assistant cannot choose its own strategies; the knowledge to support this capability requires additional research on user modeling. Instead, each user will select appropriate strategies by informing the environment that he is, for example, a manager *vs.* a programmer, developing a new system *vs.* maintaining an old system, or making major changes *vs.* a minor revision. A strategy whose rules and hints result in automatically performing type checking immediately after each component is edited would be appropriate for a minor revision, but not for a large-scale changes involving many interrelated components.

3.4. Activities as Side-Effects

Often a tool performs additional activities as side effects. For example, the analysis tool invoked for the "analyze component" activity may change the values of several attributes of components. For the purposes of our rules, setting the value of an attribute is considered an activity, resulting in a situation where one action of the intelligent assistant is embedded inside another rather than being a consequence of forward or backward chaining. This case demonstrates a limitation of our rules: Secondary actions whose arguments cannot be determined in the general case cannot be expressed easily as postconditions. Instead, potential side effects are indicated by attributes of the tool.

In such cases, the secondary activities are often described by their own rules, and these must be considered for further processing. For example, some rules related to the "uses" attribute of a component are given in Figure 3-4. The "uses" attribute lists the components that the component depends on.

```
notanalyzed(component);
    { analyze component }
analyzed(component);
errors(component);

in(module, component c) and
    < in(module, component k) or imports(module, component k) >;
    { component c uses component k }
uses(component c, component k);

exports(module N, component) and
    not equal(module M, module N);
    { import component }
imports(module M, component);

in(module, component);
    { export component }
exports(module, component);
```

Figure 3-4: Analyze Rule, Uses Rule and Import/Export Rules

The first rule gives the obvious preconditions and postconditions for the "analyze component" activity. The second rule states a component *c* cannot use another component *k* unless *k* is in the same module or is imported into the module. The third rule means that a component cannot be imported by a module *M* unless it is exported by another module *N*. The fourth rule states that a component cannot be exported by a module unless it is in that module.

What happens when the analysis tool finds that procedure *p* (a component) calls procedure *q* (another component) and tries to set the "uses" attribute of procedure *p* to include procedure *q*? If *q* is in the same module as *p*, there is no problem; the attribute is set and the analysis continues. If *q* is not in the same module, the intelligent assistant checks whether it is imported. In the case where *q* is not already imported, the assistant notes that "imports(module, component)" is a postcondition of the "import component" activity (third rule) and realizes it can perform the "import component" activity without human intervention. It considers the preconditions of this activity. The assistant queries its objectbase to find the module that contains *q*. If *q* is already exported from that module, the assistant performs the "import component" activity. If not, the backward-chaining metarule permits the assistant to follow the preconditions of the activity given in the fourth rule of Figure 3-4. The assistant can add *q* to the exports of its module, then actually import *q* into the original module, and then permit the analysis tool to set the "uses" attribute of *p*.

In the above scenario, we ignored the possibility that distinct procedures named *q* might be found in more than one module. Sometimes language-specific typing information can be used to narrow down the possibilities, but generally the intelligent assistant must interrupt the human user to explain its dilemma and to ask which *q* is intended. The assistant can then proceed as described in the previous paragraph.

If no component named *q* is in the objectbase, the assistant considers the "add component q" activity, whose postcondition is the existence of *q*. A sufficiently intelligent assistant could carry out this activity by creating a stub for the procedure within the module where the use occurs; the Gnome programming environment for Karel does this automatically [9]. If this is not feasible, an alternative would be to ask the user to create the procedure (or stub) before continuing the analysis, but this would be intrusive; a preferred alternative is to inform the analysis tool of the problem and prevent it from performing the "procedure p uses procedure q" activity. This causes the analysis tool to terminate unsuccessfully, generating the "errors(p)" predicate among its postconditions.

In this discussion, "import component" and "export component" are among the activities that can be performed by the intelligent assistant without human intervention, permitting the assistant to carry out the repairs illustrated by the example. An alternative strategy would require the assistant to take the imports and exports as given. This might be appropriate for languages, such as Ada^{TM}, that include their own module constructs, where reference to an external component without the appropriate "with" clause should be detected as an error.

3.5. Implicit Queries

In the previous example, the assistant automatically queried its objectbase to locate procedure *q*. When the environment performs a query on its own, rather than in response to a user command, we call this an *implicit query*. Implicit queries are necessary to determine whether the preconditions of rules and hints are satisfied and to find the next rules to be applied in forward and backward chaining.

Another application is to anticipate the postconditions of activities, enabling the environment to warn the user when an action is likely to lead to adverse results. Consider again the two rules shown in Figure 3-5. Through forward chaining, changing a component will lead to semantic analysis, which may result in errors. When a user invokes the editor on a particular component, the environment anticipates this forward chaining and notes the possible "errors(component)" postcondition. This causes it to perform an implicit query to determine likely causes of the errors.

The intelligent assistant cannot guess what modifications the user will make and how these will affect other components. However, it can take advantage of the "used-by" attribute to determine those components most likely to be affected. Both the "used-by" attribute and its inverse ("uses") are listed in the objectbase among the potential side effects of the editor tool.

```
reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

notanalyzed(component);
    { analyze component }
analyzed(component);
errors(component);
```

Figure 3-5: Change and Analyze Rules

The environment informs the user of potential sources of semantic inconsistencies by presenting the list of components given by the "used-by" attribute of the component argument to the editor. The user can take this information into account and choose whether or not to abort the "change component" command.

A further application of implicit queries was implied in Figure 3-2. A user gave the "change component" command, and backward chaining led the assistant to query the objectbase to determine whether all the modules affected by the proposed change were reserved by this user. If not, the environment would attempt to reserve all the necessary modules. However, this cannot succeed if some of these modules are reserved by other users. In this circumstance, the assistant presents the results of its implicit queries to the user to explain why the requested activity is not permitted.

3.6. Summary

The main points of our architecture for modeling the software development process are as follows.

- Rules define the preconditions that must be satisfied before a tool can be applied and the alternative postconditions of each tool.
- Hints define the preconditions that must be satisfied before a tool can be applied by the environment; unlike rules, hints do not affect the activities of human users.
- Two metarules define forward chaining from the postconditions of completed activities to the preconditions of other tools and backward chaining from the preconditions of desired activities to the postconditions of other tools.
- Tools may have side effects that cannot be expressed directly as postconditions, but these are nevertheless considered with respect to forward and backward chaining.
- The environment performs implicit queries to determine the attributes of software objects and the potential side effects of tools.
- Strategies group rules and hints appropriate for particular users and for particular phases of software development and maintenance. Our third metarule enables the intelligent assistant to consider these strategies during forward and backward chaining.

4. Implementation

We are implementing our intelligent assistant by by reimplementing the internal mechanisms of SMILE. The advantages of starting with SMILE are (1) we can implement in place, retaining at all times a more-or-less working environment; (2) we have continued to use much of the previous SMILE code, most notably its disaster recovery mechanisms — approximately 30% of SMILE's source code protects against internal failures and supports recovery from external failures (disk full, system crashes, abort signals, *etc.*), and (3) we can continue to support the many systems, including SMILE, that have been developed and maintained using SMILE.

We are currently replacing SMILE's hardcoded knowledge about the software development process with the rules, hints and strategies of our model. The preconditions and postconditions of rules are translated into C routines that perform the corresponding queries and changes to the objectbase. The metarules are currently hardcoded. For example, forward chaining is performed by hashing on the actual postconditions of the most recently completed activity to find rules with potentially matching preconditions, which are then checked by the procedures that implement the preconditions.

We have so far retained SMILE's original objectbase, which is mapped onto the Unix file system, but we are currently designing a more flexible mapping that will support an extensible objectbase. We expect to complete this design and its implementation within three months. We have also retained the same user interface and tools, but expect to later replace the user interface to take advantage of bitmapped displays.

5. Conclusions

Our general architecture for intelligent assistance consists of an objectbase and a model of the software development process. The advantage of an objectbase is it permits the assistant to present a 'fileless environment' to its users, so the users are concerned only with the logical entities associated with software development and not with the details of the underlying file system and operating system. The advantages of a model of the software development process is that it can automate bookkeeping chores and other simple development activities and can constrain the invocation of tools to maintain consistency among the software objects.

These notions have been promoted by other researchers as the fundamental basis for a programming environment that understands what it does. The specific contribution of our research is the formalization of opportunistic processing, including implicit querying. Opportunistic processing is made possible by rules that describe the preconditions and postconditions of software development activities, metarules that permit the intelligent assistant to take advantage of these rules to enable automatic processing, and strategies that guide the assistant's application of the metarules. The result is a significant improvement in the assistance that the environment can provide for software development and maintenance by individuals as well as by teams of programmers, managers and other staff.

References

[1] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[2] Robert M. Balzer.
Living in the Next Generation Operating System.
In *10th World Computer Congress.* Dublin, Ireland, September, 1986.
Proceedings to appear as a book published by Springer-Verlag.

[3] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[4] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin.
Programming Expert Systems in OPS5.
Addison-Wesley Pub. Co., Reading, MA, 1985.

[5] CLF Project.
Introduction to the CLF Environment.
March, 1986.
USC Information Sciences Institute.

[6] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.

[7] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[8] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65-72. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[9] David Garlan.
Private communication.
July, 1986
Regarding capabilities of Gnome programming environments.

[10] Adele Goldberg.
The Influence of an Object-Oriented Language on the Programming Environment.
In *1983 ACM Computer Science Conference.* February, 1983.
Reprinted in [3].

[11] C.A.R. Hoare.
An Axiomatic Approach to Computer Programming.
Communications of the ACM 12(10):576-580, 583, October, 1969.

[12] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *Thirty-Second IEEE Computer Society International Conference.* San Francisco, CA, February, 1987.
Conference article to appear. Now available as CMU Software Engineering Institute Technical Report, SEI-86-TM-14, September, 1986.

[13] Beverly L. Kedzierski.
Knowledge-Based Project Management and Communication Support in a System Development Environment.
In *4th Jerusalem Conference on Information Technology.* Jerusalem, Israel, May, 1984.

[14] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[15] John R. Nestor.
Toward a Persistent Object Base.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments.* June, 1986.
Proceedings to appear as a book published by Springer-Verlag.

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

[17] L.J. Osterweil.
Toolpack — An experimental software development environment research project.
IEEE Transactions on Software Engineering SE-9(6), November, 1983.

[18] Dewayne E. Perry.
Position Paper: The Constructive Use of Module Interface Specifications.
In *Third International Workshop on Software Specification and Design.* London, England, August, 1985.

[19] C.V. Ramamoorthy, Vijay Garg and Rajeev Aggarwal.
Environment Modelling and Activity Management in Genesis.
In *SoftFairII: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, pages 2-10. December, 1985.

[20] C.V. Ramamoorthy, Yutaka Usuda, Wei-TekTsai and Atul Prakash.
GENESIS: An Integrated Environment for Supporting Development and Evolution of Software.
In *IEEE Computer Society's Ninth International Computer Software & Applications Conference*, pages 472-479. October, 1985.

[21] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

[22] M. J. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1:364-370, 1975.

[23] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[24] Barbara J. Staudt, Charles W. Krueger, A.N. Habermann and Vincenzo Ambriola.
The GANDALF System Reference Manuals.
Technical Report CMU-CS-86-130, Carnegie-Mellon University, Department of Computer Science, May, 1986.

[25] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[26] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Reprinted in [3].

[27] Warren Teitelman and Larry Masinter.
The Interlisp Programming Environment.
IEEE Computer 14(4):25-34, April, 1981.
Reprinted in [3].

[28] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[29] Richard C. Waters.
KBEmacs: Where's the AI?
The AI Magazine VII(1):47-56, Spring, 1986.

[30] Terry Winograd.
Breaking the Complexity Barrier (Again).
In *SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, pages 13-30. Gaithersburg, MD, November, 1973.
Reprinted in [3].

Granularity Issues in a Knowledge-Based Programming Environment

Peter H. Feiler
Carnegie-Mellon University
Software Engineering Institute
Pittsburgh, PA 15213

Gail E. Kaiser*
Columbia University
Department of Computer Science
New York, NY 10027

September 1986

Abstract

MARVEL is a knowledge-based programming environment that assists software development teams in performing and coordinating their activities. While designing MARVEL, we discovered several *granularity* issues that have a strong impact on the degree of intelligence that can be exhibited, as well as on the friendliness and performance of the environment. The most significant granularity issues include the refinement of software entities in the software database and decomposition of the software tools that process the entities and report their results to the human users. We describe the many alternative granularities and explain the choices we made for MARVEL.

*The research presented in this paper was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

To appear in 2nd Kansas Conference on Knowledge-Based Software Development, Manhattan, KA, October, 1986.

Table of Contents

1. Introduction

We are developing a knowledge-based programming environment called PROFESSORMARVEL, or MARVEL for short.[1] MARVEL is knowledge-based in the sense that it incorporates knowledge of the logical entities and the activities involved in the software development process. It is an environment rather than a software tool because it actively participates in the software development process rather than remaining passive until explicit demands are made by its users. The primary functions of MARVEL are (1) to interactively answer queries about the current status of the development effort and the relationships among components of the target software system and (2) to automatically perform bookkeeping chores and simple development activities. This is in contrast to some other intelligent assistance systems such as the Programmer's Apprentice (also known as KBEmacs) [29], which focuses on planning, and CHI (previously PSI) [23] and the Formalized System Development system (FSD) [2], which focus on automatic programming. MARVEL is also not concerned with solving the problems of natural language processing; queries are expressed in a simple but formal notation.

Unlike most other knowledge-based programming environments, MARVEL supports multi-programmer software development teams in addition to individual programming efforts. For example, it includes facilities corresponding to Build [7] and SCCS [21] to coordinate simultaneous and sequential activities among multiple developers. However, MARVEL approaches these facilities in a participatory, knowledge-based fashion that enables it to automatically invoke the tools at the proper times without human intervention.

MARVEL is our second multi-user programming environment. Our first system, called SMILE [24, 14], presents a fileless environment to its users, answers queries about the evolving software system, and automatically invokes various software development tools. However, SMILE's knowledge is hardcoded into the environment and is not extensible; SMILE does not really 'understand' what it is doing and users cannot augment SMILE with additional knowledge. SMILE was developed several years ago to support our research on the Gandalf project [20]. It has since been used extensively by other projects at Carnegie-Mellon University and at AT&T Bell Laboratories, and has been distributed to approximately forty sites. SMILE was implemented in C and runs on Unix™.

Although we found SMILE very useful, and in fact relied on it for the implementation and maintenance of our Gandalf research, we became convinced that an environment that 'understands' what it was doing could provide much more valuable assistance. Because of this, we have based our design of MARVEL on a general architecture for intelligent assistance. The architecture consists of an *objectbase* and a *model of the software development process*. The objectbase maintains all software objects, including tools such as the editor and the compiler. The objectbase provides MARVEL with *insight* into the various classes of objects and the relationships among objects, such as one object is a component of another and a particular object may be applied to another object to produce a third object.

The model imposes a structure on programming activities. It consists of a user-extensible

[1]Professor Marvel was the (Kansas) name of 'the man behind the curtain' in the movie *The Wizard of Oz*.

collection of production-like rules that specify the particular conditions that must exist for particular activities to be carried out. The rules enable MARVEL to provide *opportunistic processing*, where the environment performs simple activities automatically, such as satisfying the preconditions of an activity and then carrying out the activity when it knows the results of the activity will soon be required by a user. For example, when a user wants to run the system, MARVEL can automatically link the executable system from its component modules; if necessary, MARVEL recompiles modules automatically to satisfy the precondition to linking that the object code for these modules is up to date.

Insight and opportunistic processing are presented elsewhere [15], and will be discussed only briefly. Our focus is the *granularity* issues that arose during our long experience with SMILE and during the subsequent design of MARVEL. In particular, we ran into several problems regarding the appropriate refinement of logical entities to be maintained as separate software objects and the units appropriate both for tools and for reporting the results of tool processing to the users. Choice of granularity affects the capabilities of the intelligent assistant, the friendliness *vs.* intrusiveness of the programming environment and, of course, performance and responsiveness. We believe that discussion of these issues, including an explanation of the decisions we made regarding MARVEL, will prove useful to other researchers who are in the process of building knowledge-based programming environments.

In the rest of this paper, we briefly sketch MARVEL's underlying basis for intelligent assistance. We then address granularity issues, categorized into three areas: the granularity of structure in the objectbase, its impact on tools, and the granularity of processing automatically performed by the environment. We conclude by summarizing the significance of these issues for achieving intelligent assistance for software development and maintenance.

2. A Basis for Intelligent Assistance

The distinguishing feature of an intelligent assistant is that it understands what it is doing [30]. We believe that both an objectbase and a model of the software development process are prerequisites to intelligent assistance. An assistant cannot understand why it performs particular activities unless it knows

- the properties of the objects it manipulates,
- the capabilities of certain objects (programmers and tools) to manipulate other objects,
- the preconditions required by each activity,
- the results or postconditions of each activity.

Therefore, MARVEL includes a general objectbase that maintains software entities and tools, and an extensible collection of rules that describe the preconditions and postconditions of software development activities.

2.1. The Objectbase

There are several possible forms for MARVEL's objectbase to take [19]. To maximize flexibility, we chose an objectbase similar to the objectbases of object-oriented programming languages, such as Loops [25]. In particular, we adopted their support for multiple inheritance and active values. The same concepts are found in the objectbases supported by other knowledge-based programming environments, such as AP3 [3] and Refine™ [23].

In MARVEL's objectbase, each object is an instance of a class, which defines certain attributes of each object and inherits other attributes from its superclass(es). Some attributes define the relationships among the objects, while others trigger activities when they are accessed and/or updated. The software development activities applicable to the members of a class are defined as methods for the class.

The objectbase enables MARVEL to present a 'fileless environment', exposing its users only to the logical structure of the target software system and hiding the underlying files and directories. As far as the users are concerned, the environment consists of a set of typed and interconnected software objects that represent both the system and its history of development. Object types include module, procedure, type, design description, user task (or development step), user manual, *etc*. Typing of these objects permits MARVEL to provide an object-oriented user interface similar to the Smalltalk-80™ environment [10]. This means that the environment makes available to each user only those commands that are relevant to the object under consideration.

The interconnections among software objects represent the logical structure of the system. The more detailed the structure, the more information is available for browsing and querying, and the more MARVEL can deduce which activities it can suitably perform and understand those tasks that the users carry out. These issues are addressed in Section 3.

2.2. The Model

There are also several possible forms for the rules we use to model the software development process. Again to maximize flexibility, we chose a style of rule developed in the program verification community. Each software activity is associated with preconditions and postconditions, as defined by Hoare [13]. The postconditions of an activity may satisfy the preconditions of future activities.

Our rules are similar to the production rules of Ops5 [5] and the automation rules of FSD in that each rule has both a condition and action. When the condition is true, or satisfied, then the action may be carried out. Our rules are different from productions in that the action is divided into two parts, an activity and its postconditions. Because our rules have postconditions, we refer to the original conditions as preconditions. We use the activity part of a rule to represent an integral software development task. For example, "compile module" is one activity and "edit procedure" is another. The preconditions of "compile module" might be that the module is not compiled and that all its components have been analyzed without errors; the postconditions might be that the module was compiled and this produced either object code or error messages. Preconditions and postconditions are written as well-formed formulas (wffs) in the first order predicate calculus.

Our rules are maintained as part of the MARVEL's objectbase. They permit MARVEL to perform activities and to explain activities in terms of the rules. Forward chaining allows that if the preconditions of a rule are satisfied, then MARVEL can perform the activity; the postconditions may satisfy the preconditions of other rules, which can then be applied. Backward chaining allows that if a user requests a particular activity, then MARVEL can attempt to satisfy its preconditions; this often requires the environment to first perform other activities whose postconditions match the preconditions of the original rule.

Forward and backward chaining, together with the rules and the objectbase, support insight and opportunistic processing. One simple example of insight occurs when a user invokes the "change procedure" command, which enables editing of both the specification (header) and body of a particular procedure. MARVEL uses the results of its incremental analysis of dependencies among software components to inform the user of the potential consequences of this action; for example, each calling procedure will have to be modified if the number or order of the parameters are changed. A simple example of opportunistic processing occurs after a user completes the "change procedure" command by writing out the procedure from the editor. MARVEL notices that a postcondition of this activity is that the analysis of the procedure is not up to date, which is also the precondition of the "analyze procedure" activity; MARVEL uses forward chaining to automatically update its incremental analysis.

3. Granularity of Structure

The degree of intelligence that can be demonstrated by MARVEL, or any knowledge-based programming environment, is intimately tied to the granularity of independent entities maintained by its objectbase and to the granularity of the processing tools it has available. The granularity of structure refers to the extent the target software system is decomposed into *separately stored* entities. An entity is considered separately stored when it is represented as an object; in other words, it is not necessary to parse or analyze a 'larger' object to derive the entity. For example, a procedure might be represented as an object; its attributes might include its name and a list of the other procedures it uses, but not the specific statements that make up the procedure. It would be necessary to parse the procedure's source text to find any particular statement.

There are several reasons why it is desirable to have the logical entities of the target system separately accessible. In the first place, an object can be referenced from other parts of the system, while it is not possible to refer directly to only a portion of the information within an object. For example, MARVEL can support separate access to modules, procedures, macros, and global variables in the program domain, sections and subsections in the document domain, and plans, tasks, and developers in the management domain. These logical entities depend on each other in various ways, such as actual use dependency among software components such as procedures, macros and variables, intentional use dependency as expressed through export and import clauses of modules, compilation order, or referential use such as reference to a section or a citation in a document. If this interconnection structure is accessible, MARVEL can detect inconsistencies among the components of the target system; for example, a module *M* might import a procedure *p* from module *N*, even though there is no procedure within *M* that actually calls *p*. If this situation persists, it may imply an error, and MARVEL can bring this to the user's attention.

Certain logical entities should be separately accessible as objects because then it is possible to associated status information with the entities by represented the status as one or more attributes. The status may represent the need to or the result of processing an entity, such as analysis of a procedure, code generation for a module, or running a section of a document through the document processor. It can also indicate coordination information among multiple users and between users and tools, *i.e.*, synchronization and version information [27].

There are situations where it is desirable to store the status of an object as an attribute that references another object with its own subparts. This can improve the degree of intelligence exhibited by the knowledge-based programming environment. For example, to support 'smart recompilation' [28], it is necessary to store status information about each intermodule symbol definition and use. This enables MARVEL to recompile only those entities that actually use modified symbols, as opposed to recompiling all modules that depend in any way on the module whose interface has changed. In contrast, SMILE supports import and export clauses, but does not permit more detailed relations among individual components, so it is not able to provide this level of intelligent processing.

It may be desirable for users to navigate and manipulate the target system according to the structural units represented by logical entities. Syntax-directed editors [26] usually support cursor movement and manipulation at the language construct level; Rational^{TM} [1] takes this to an extreme by performing all processing in terms of the Diana [6] representation of Ada^{TM} programs. Even text editors support a certain amount of structure, such as the electric-lisp mode in Emacs [12] recognizing matching parentheses. Graphical editors support manipulation of basic graphical symbols such as lines, circles and icons as well as composite graphical units. Word processing systems support character, word, sentence and paragraph manipulation. MARVEL supports viewing at the level of objects, using certain of their attributes as paths that can be followed by the browser to other objects.

User actions, especially modifications, may have different effects for different logical entities. For example, editing a comment does not affect analysis or code generation but is relevant to updating hardcopy listings, whereas modification to a design specification not only affects other parts of the design, but also the implementation. Thus, MARVEL provides object-oriented commands to reflect these distinctions. MARVEL can then recognize a user's focus of attention, *i.e.*, the software objects currently under consideration, as well as the extent of his or her modifications in order to intelligently and effectively restore consistency of the target system. SMILE's more primitive user interface requires the user, for example, to give the "edit procedure" command to edit the body of a procedure and the distinct "change procedure" command to edit its specification (header).

4. Impact of Structure Granularity on Tools

The question arises as to the choice of the smallest logical entity that should be separately stored. The answer must consider that there are two ways to have logical entities separately accessible. One is by analyzing a 'larger' entity that is stored as a separate object in the objectbase, and deriving the 'smaller' entities as needed. The alternative is to represent each logical entity as a separate object. The appropriate choice is a tradeoff between the space cost of the proliferation of objects and explosion of information to be stored on the one hand and the time cost of reprocessing of information on the other. This manifests itself in a variety of ways.

Tools provided as part of a knowledge-based programming environment may require a special interface to the objectbase as they may not be able to cope directly with composite objects. For example, a compiler requires adaptation to be able to process modules as separate compilation units when they consist of sets of references to objects representing imported entities and a composition of objects representing the procedures, *etc.*, comprising the module. It is preferable to bring in existing tools without modifications; DSEE™ [18] does this for version control by providing a virtual interface between each tool and the version manager. We would like to do this in the general case, so MARVEL provides multiple *views* corresponding to the normal interfaces of the tools [9]. SMILE does not support views, and so is forced to store objects in the form expected by its tools; this sometimes results in duplication of information when the same kind of object is processed by multiple tools.

The status information of all components of a composite object may be accessed frequently. For example, the result of analysis of all components of a module must be positive before code generation should be done. MARVEL can either compute the composite status value every time it is desired or it can cache the value in the module object and maintain it incrementally, as components are modified and re-analyzed, using, for example, finite differencing techniques [11].

Another example of status information is the error messages resulting from unsuccessful processing. Users tend to query them at times other than the time they are produced by the processing tool, except when the activity is performed on user demand. MARVEL stores error messages explicitly for strongly-typed languages, since recomputation is costly.

Interconnection information, such as actual use, may not require explicit representation, but this is orthogonal to whether the information is explicitly stored or dynamically determined. In other words, an attribute can be calculated only when needed, and information can be stored as part of some composite attribute rather than separately. One example of the first category is use dependency of local variables. This is rarely queried because all use sites are often displayed simultaneously. A user can visually search or use a viewer (editor) search command. Where explicit representation is necessary, the environment may explicitly store only the module in which an exported procedure is actually used (because the whole module has to be recompiled) and dynamically analyze the module when queried about the calling procedures, or it may explicitly store the procedure that calls the exported procedure but not every callsite within the procedure as those are easily found by direct search when needed. SMILE follows the former strategy, but the response time is widely variable and sometimes unacceptable, so MARVEL follows the latter course.

An objectbase permits tracking of modifications to objects. Representing logical entities as separate objects at appropriate granularities eliminates additional processing such as recognition of changes within regions of an object by the editor or content comparison algorithms (such as Diff [27]). For example, procedures may be represented as composite objects consisting of the specification, a description, and the implementation. Using this as the lowest level structural granularity permits MARVEL to limit side-effect propagation considerably, since other entities can be affected only when the specification is changed.

Decomposition of the target system into separate objects at a small grain places certain requirements on the object-viewing and browsing capabilities of the environment. On one hand, a user should not be starved of information as is the case with single-level object viewers. For example, viewing a module may result in display of only the names of components, without even an indication of their type. More context should be provided to the users.

On the other hand, a user should not be overloaded with information, which may lead to disorientation and confusion. An example of this is the presentation of the target system as a single textual unit, decorated with all available status information — possibly encoded in a range of symbols. A balance must be struck as to the amount of information to be displayed at any time and the desire to reduce explicit querying for information. This may change over time as the users carry out different activities. For example, while making major changes to the system a user has little interest in code generation status. Similarly, when examining an imported module, the user's view should be limited to its specification. SMILE solves these problems with distinct commands for different levels of detail. MARVEL includes an objectbase viewing and browsing capability supporting multiple views and multi-level viewing, attempting to make best use of available screen space through multiple viewing panes.

5. Granularity of Processing

As previously discussed, the tools as well as the users can take advantage of multiple views. A related issue is whether or not the users should have multiple 'views' of the tools. The granularity of processing determines the responsiveness of the environment as well as the intelligence perceived by its users. Responsiveness refers both to feedback to the users regarding inconsistencies in the target system and to processing of the target system to derive other representations, *e.g.*, to prepare for execution or for formated printing. MARVEL and other knowledge-based programming environments are interactive environments that attempt to increase user productivity. This means that each user should get feedback while in the context of the problem spot; the environment displays intelligence by understanding the user's notion of context and relating it to the results of the tools.

This also means that a user should not have to wait excessively for the computer to complete its share of the work. This is accomplished by processing at the appropriate level of granularity and by processing opportunistically. The availability of both forward and backward chaining permits MARVEL to perform activities any time between when the preconditions are satisfied and when the postconditions are required. Furthermore, not all processing has to be done at the same level of granularity. Granularity of processing that results in feedback to the user is strongly influenced by the context and time in which feedback is expected. Note that feedback may involve simple visual cues, such as changing the font of the prompt, rather than immediately dumping all the error messages on the user's display. Granularity of processing resulting mainly in derived entities such as object code is primarily influenced by the following tradeoff. On the one hand, we have the possibility of processing many small units, thus reducing the time of processing one unit, yet causing possibly redundant processing of the same unit when it is frequently affected by changes to other small units. On the other hand, processing larger units at less frequent intervals leads to the expense of longer delays when the users need the results. In some cases, this problem means exploration of separate processing techniques in order to avoid processing of the complete target system. Examples include linking in pieces through use of indirect references [8], and formatting in pieces by a document processor through maintenance of formatting context — as is supported by Scribe [22].

Feedback to the users can occur at several levels of granularity, where the grain size chosen may be different for different kinds of analysis. One form of feedback is enforcement of a particular kind of consistency. The most prominent example is enforcement of syntactic consistency, as done by syntax-directed or form editors. This is accomplished either by limiting the choices of entry to acceptable ones, *e.g.*, by providing a menu with the legal set of constructs, or by immediately checking and rejecting invalid entries. Another form of feedback is checking for consistency and reporting any violations, but accepting the input into the objectbase. In this case, MARVEL records whether or not objects have been checked for each kind of consistency and, if so, the results of each analysis.

During different phases of development and maintenance, a user may desire feedback for the same kind of consistency at different granularity levels. For example, while writing a new piece of code, a user does not want to be told repeatedly about the use of an undefined identifier until he has completed his activity with the procedure or module. However, when carrying out minor corrections, more immediate feedback is desirable. MARVEL offers such flexibility by separating checking from reporting. In this way, checking for a particular kind of consistency is always performed at the same level of granularity — the smallest level for which feedback is desired (as selected by the current user) — with one set of analysis processes. Reporting can be realized by querying the results of checking, and MARVEL does this by performing queries automatically at different times as determined by the reporting strategy.

This behavior is in contrast to SMILE, which initially performed compilation at the level of procedures and immediately informed the user of any errors. We found this behavior unacceptable. SMILE was modified to perform symbol resolution and type checking at the procedure level, but to apply compilation only to modules. Errors were no longer reported except in response to user queries, but their detection was indicated by an unintrusive visual change in the display that remained until the errors were corrected.

6. Conclusions

The fundamental tradeoffs regarding granularity of logical entities and of automatic processing demonstrate the impact of the choices of granularity on the apparent intelligence of an environment as well as on its responsiveness and performance. The most notable choices we made for PROFESSORMARVEL are as follows.

- A knowledge-based programming environment can more quickly answer more complex queries when it incrementally updates its analysis of the relationships among logical entities of the target software system and also maintains these entities refined to the level of relationships among individual software components rather than among modules;
- An environment is less intrusive and more informative when the granularity of automatic processing is separated from the granularity for automatic reporting of the results of processing, and it is not difficult to separate these behaviors for semantic analysis, compilation and other activities.

We believe that these choices are also appropriate for most other knowledge-based programming environments. While SMILE was targeted for C, we designed MARVEL to support either C, CommonLisp, or Ada. We also kept in mind document formatting languages such as Scribe and project management facilities such as those found in CMS [16] and DSEE [17]. Thus our conclusions cover a wide range of possible entities as well as tools.

Acknowledgements

We would like to thank Cecile Paris and Ursula Wolz for their useful criticisms and suggestions regarding this paper. Purvis Jackson assisted us with technical editing. Steve Popovich is working with us on the implementation of MARVEL.

References

[1] James E. Archer, Jr. and Michael T. Devlin.
Rational's Experience Using Ada for Very Large Systems.
In *First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages B.2.5.1-B.2.5.11. Houston, TX, June, 1986.

[2] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[3] Robert M. Balzer.
Living in the Next Generation Operating System.
In *10th World Computer Congress*. Dublin, Ireland, September, 1986.
Proceedings to appear as a book published by Springer-Verlag.

[4] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[5] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin.
Programming Expert Systems in OPS5.
Addison-Wesley Publishing Co., Reading, MA, 1985.

[6] *Diana Reference Manual*
Carnegie-Mellon University, Department of Computer Science, 1981.

[7] V.B. Erickson and J.F. Pellegrin.
Build — A Software Construction Tool.
AT&T Bell Laboratories Technical Journal 63(6):1049-1059, July-August, 1984.

[8] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.

[9] David Garlan.
Views for Tools in Integrated Environments.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments*. June, 1986.
Proceedings to appear as a book published by Springer-Verlag.

[10] Adele Goldberg.
The Influence of an Object-Oriented Language on the Programming Environment.
In *1983 ACM Computer Science Conference.* February, 1983.
Reprinted in [4].

[11] Allen T. Goldberg.
Knowledge-Based Programming: A Survey of Program Design and Construction Techniques.
IEEE Transactions on Software Engineering SE-12(7):752-768, July, 1986.

[12] James A. Gosling.
Unix Emacs
Carnegie-Mellon University, Department of Computer Science, 1982.

[13] C.A.R. Hoare.
An Axiomatic Approach to Computer Programming.
Communications of the ACM 12(10):576-580, 583, October, 1969.

[14] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
Technical Report SEI-86-TM-14, CMU Software Engineering Institute, September, 1986.
Submitted for publication.

[15] Gail E. Kaiser and Peter H. Feiler.
An Architecture for Intelligent Assistance in Software Development.
Technical Report SEI-86-TM-12, CMU Software Engineering Institute, September, 1986.
Submitted for publication.

[16] Beverly L. Kedzierski.
Knowledge-Based Project Management and Communication Support in a System Development Environment.
In *4th Jerusalem Conference on Information Technology.* Jerusalem, Israel, May, 1984.

[17] David B. Leblang and Robert P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[18] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[19] John R. Nestor.
Toward a Persistent Object Base.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments.* June, 1986.
Proceedings to appear as a book published by Springer-Verlag.

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

[21] M. J. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1:364-370, 1975.

[22] *SCRIBE Document Production Software User Manual*
Unilogic, Ltd., 1985.
The original Scribe manual was written by Brian K. Reid and Janet H. Walker.

[23] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[24] Barbara J. Staudt, Charles W. Krueger, A.N. Habermann and Vincenzo Ambriola.
The GANDALF System Reference Manuals.
Technical Report CMU-CS-86-130, Carnegie-Mellon University, Department of Computer Science, May, 1986.

[25] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[26] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Reprinted in [4].

[27] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[28] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8(3):273-291, July, 1986.

[29] Richard C. Waters.
KBEmacs: Where's the AI?
The AI Magazine VII(1):47-56, Spring, 1986.

[30] Terry Winograd.
Breaking the Complexity Barrier (Again).
In *SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, pages 13-30. Gaithersburg, MD, November, 1973.
Reprinted in [4].