Building plan recognition systems on arbitrary applications : the spying technique. F. Pachet, S. Giroux , IJCAI'95 Workshop on New Generation of Plan Recognition Systems, Montréal (1996).

Building plan recognition systems on arbitrary applications: the spying technique

François Pachet
Laforia-IBP, Université Paris 6, Boîte 169
4, place Jussieu,
75252 Paris Cedex 05, France,
Email: pachet@laforia.ibp.fr

Sylvain Giroux
LICEF, Télé-Université,
1001, rue Sherbrooke est,
H2X3M4 Montréal, Canada,
Email: giroux@teluq.uquebec.ca

Abstract

There are today a lot of different techniques for performing plan recognition in various domains, such as Intelligent Tutoring Systems, Human-Computer Interaction, or multi agent Systems. In order to bring theory into practice, we claim that experiments should be carried out a larger scales than what is done today. To do so, we stress on the importance of building incomplete plan recognition systems that may run on arbitrary existing applications. We propose a technique for grafting plan recognition systems onto arbitrary object-oriented applications, without modifying their code. This technique is based on the notion of *spy* a particular object, that may be inserted in object-oriented systems in a non-intrusive manner, and may track incoming messages to arbitrary objects. We show how spies may be defined, installed automatically to produce low-level information about a system's behavior. Information produced by spies may then be fed to plan recognition systems that perform various tasks such as advice-production in tutorial systems or program introspection and analysis.

1. Introduction

The context of our work is the formalization of pedagogical expertise, and the development of frameworks and tools for the design of tutorial systems in the context of remote teaching (as practiced by the Télé-Université, Montréal). More particularly, one of our goals is to provide frameworks to specify and build "over the shoulder" advisor systems. In this scheme, we are faced with two problems:

I) The strongest constraint of our work is to develop advisor systems on top of *existing applications*, instead of having to rewrite applications from scratch. The benefit of reusing existing applications is enormous, and has short-term as well as long-term consequences: applications may be designed and implemented independently of their advisor component; modifications of advisor systems do not require intervention of the developers of the initial applications. For instance, [Desmarais & al. 93] compared different plan recognition techniques on the use of WordPerfect. Their study required the development of a WordPerfect emulator, "configured to monitor the user's actions". This study is a perfect illustration of the kind of work cannot afford. Instead, we want to reuse as much as possible work done by others.

II) We do not know yet which kind of plan recognizer is best suited to the need of the application. A variety of plan recognition systems have been designed, each one is more suited to certain tasks than the other. [Kautz & Allen 86] proposed a technique that is satisfying only when plan libraries are complete. [Carberry 90], [Konolige & Pollack 89] propose alternative theories of plan recognition in which beliefs and intentions are ascribed to the user by using a direct argumentation system, which does not require a complete library of plans, but with other drawbacks. Lots of authors have investigated grammar-based approaches: action grammars [Reisner 81], task-action grammars [Payne & Green 86], and various implementation for corresponding parsers have been proposed (e.g. [Hoppe 88]). [Quast 93] proposes a technique to recognize action plans based on a multi-layered symbolic nets, with a bottom-up spreading, exemplified on Excel abstract tasks. The list is long and growing daily. Although some efforts have been made to unify all existing theories of plan recognition, and to find arguments pros and cons each technique in general [Greer & al. 93], our position is pragmatic: each situation requires a specific technique, which is not necessarily known at application development time.

The technique we devised, called *spying* addresses the two preceding problems for the construction of plan recognizers. The technique, initially designed to build advisor systems, turned out to be directly applicable to any kind of plan recognition systems, regardless of the domain field. Our technique however, is based on the assumption that the application is written in an object-oriented language, satisfying the constraints listed below. The current implementation is realized in Smalltalk. Porting to other object-oriented languages (C++, CLOS) is in progress.

In this scheme, the application is called the *host system*. The advisor system - or any plan recognition system - is designed and built independently of the host system. Our contribution is to provide a scheme for linking both systems that preserves the independence of the two systems and does not require any modification of the host system.

1.1. Typical examples of host applications

The typical application on top of which we need to build plan recognizers are the following:

- The Smalltalk tools themselves, such as the Browser or debugger. The Smalltalk environment is acknowledged to be one of the most powerful programming environments on the market. However, the learning phase is huge, and learner are often lost in the complexity of the interface. Simple advice may be produced based on rudimentary analysis of user's actions. We showed how this advice-giving modules can be designed and implemented without modifying the tools on which the module is grafted [Pachet & al. 95]. The design and implementation of advisers on more sophisticated tools such as the VisualWorks GUI [VisualWorks 94] is in progress.
- Smalltalk applications such as tutorial systems [Paquette & al. 94]. Classical plan recognition techniques are then used to produce advice to the user, according to its interaction with the tutorial system.
- Monitoring and introspecting systems. In order to optimize object-oriented systems, a fine analysis of their dynamic behavior is required. We claim that such analysis may be seen as a particular plan recognition task in which the activity observed is not a sequence of human actions, but a program execution. The technique we propose here is directly applicable to this class of problems as well.

We will now describe the spying mechanism, and describe the EpiTalk system, a dedicated advice production system that feeds from information gathered by spies.

2. From low-level tasks to messages

2.1. Object-oriented systems

In object-oriented languages such as Smalltalk, CLOS or C++, the basic unit of activity of a program is the *message*. An object-oriented program is designed and implemented as a set of objects. Procedures are represented as messages sent to objects. The activation of a message consists in turn in sending messages to other objects and so forth (until some primitive message is executed). Since every procedure is represented as a message, user's action (mouse click, item selection in a list, etc.) are eventually materialized as messages sent to particular objects. For instance, in Smalltalk, according to the MVC paradigm [Krasner & Pope 88], user's action are handled by a "controller" object, which interprets them and sends messages to a "view" object and to a "model" object to request information, and update its display on the screen.

In order to build plan recognition systems on such Object-Oriented programs, the first task is to identify relevant messages in the application that correspond to user's actions. Note that since we do not want to impose restrictions on the nature of actions we want to analyze, this tasks may not be delegated to some kind of "event handler" mechanism, because the list of "trackable" actions would then be pre-defined, and could not be suited the particular application.

3. Spies

Our technique relies on the notion of "spy". A spy is an object that may be inserted in a program in order to detect all incoming messages to a given object, without modifying the program's semantics.

In practice, this is realized by a combination of capsule programming [Pascoe 86] together with mechanisms for swapping objects' identities. Capsules are objects that "wrap" around arbitrary objects, and redefine some of their behavior in a non intrusive way. The main idea behind capsules is their ability to redefine message reception at the instance level. This has traditionally been implemented using a particularly popular mechanism of Smalltalk, the `doesNotUnderstand:`, which is considered the main reflective feature of Smalltalk [Foote & Johnson 89]. Thanks to this mechanism, capsules can easily intercept incoming messages to encapsulated objects, and redefine their semantics in various ways.

The capsule mechanism introduced by Pascoe consists in substituting capsule objects to spied objects. In Pascoe's view, however, this substitution is left to the responsibility of the tracing program, which is responsible for wrapping around the spied object when it is created. Practically, capsules require a *modification of methods* that actually create the objects to be spied. Encapsulation therefore may not be performed on existing code without modification. We propose to automate the creation of capsules, by a mechanism which automatically encapsulates objects. This mechanism is based on the systematic use of the Smalltalk primitive `become:`. This primitive message swaps the internal addresses of two arbitrary objects. Here, the idea is to encapsulate objects by making them physically "become" spies, which in turn point to the original object (Cf. Fig. 1). Thanks to this mechanism, we can encapsulate objects "from the outside", without redefining existing code.

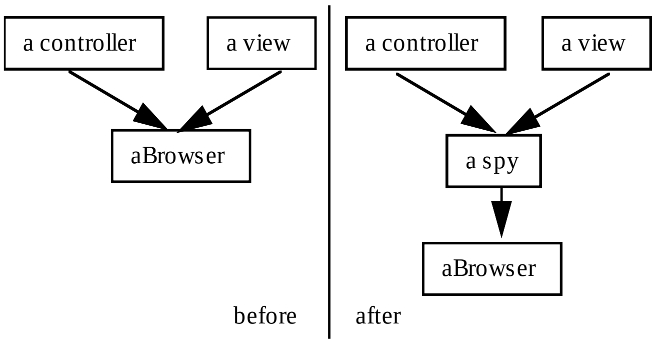

Figure 1. Installing a spy on a browser.

3.1. Issues and practical solutions

Based on our experience with the spying paradigm, we identified three main issues related to the use of spies, and propose practical solutions for them. The issues are the following (they are discussed in more details in [Pachet & al. 95]):

1) The *self problem*. Spies may not intercept messages sent internally by an object to itself [Lieberman 86]. There is no

solution to this problem, but we make the - natural - assumption that only external messages are interesting to spy in the context of plan recognition systems. A second related, and more subtle, problem is the "reverse-self" problem: certain messages may be intercepted by spies when in fact they should not, because they are mere "echoes" of already intercepted messages. For instance, the dependent mechanism in MVC ensures that each time a model object changes its state, it warns its dependents (usually view objects) of the change so that they can update their display, if needed. Dependents in turn query the model to get the information. If the model is spied, then each time a view object will query its model, the spy will intercept all the query message when only the message that caused the original change should have been intercepted. To solve this problem, we designed spies so that they intercept only one message at a time, using stack introspection.

2) *Classes* can not be spied, for technical reasons. This prevents us from easily detecting the *creation* of new objects. The need for detecting object's creation is very natural. For instance, when spying a browser object in the Smalltalk environment, it is interesting to detect when the user decides to open new browsers (e.g. hierarchy browsers) from the initial one. The newly created objects should then themselves be spied to record and analyze the user actions accordingly. We designed a scheme to circumvent this problem, by defining specialized spies whose task is to install new spies on newly created objects. Note that the reverse problem, i.e. detecting object destruction is not relevant in an object-oriented setting, since this work is usually performed by the garbage collector.

3) Spies sometimes understand too much. By definition of the capsule mechanism, there are basic messages (such as the vital message `class` that yields the class of an object) that are directly interpreted by spies, thereby modifying locally the semantics of the spied system. We solved this problem practically by providing tools to detect such cases, and ask the designer of the spying module to slightly modify the code and avoid such messages. This is the only (small) case when the application's code has to be modified.

4) The processing of spied information by spies (or by specialized advisers, as described below) may be a burden for the computer actually running the host application. In some case, this may alter the performance of the host system in a significant manner, and can even modify its semantics (typically if the host application expects real time reactions). To guaranty that spies do not perturbate the host application, we coupled the spying mechanism with the RPC (remote-procedure control) scheme [Pachet & al. 95]. Thanks to this coupling, spies transmit intercepted messages to objects that are physically located on a different machine, so the burden of the spying mechanism is really marginal.

3.2. Replayer systems: a first layer of plan recognition

The spying mechanism allows to *collect* information circulating within an object-oriented program. Plan recognition techniques are then used to analyze this information in order to perform various tasks. The first level of plan recognition systems is the recorder. This objects does nothing but record information given by spies, and is able to replay it, i.e. send the messages back to the spied objects in the order it received them. Spying allows to build "generic recorder" objects, without modifying the code of the classes involved. For instance, this recorder may be used to replay a sequence of user's action in a browser, to put it back into a previous state. Of course, building recorders is not a particularly difficult task in itself. Our contribution so far is to allow the construction of recorders for any kind of system without any modification of the system being recorded.

4. Epiphyte systems

4.1. Definition

For systems requiring a more complex analysis of user actions, such as advisor systems, the main problem is to organize information collected by spies. This problem concerns a whole class of systems, that we call *epiphyte systems*, after a botanical metaphor: epiphytes plants grow on host plants without causing them any damage (as opposed to *parasites,* which cause damage to their host, and - worst in the hierarchy - *predators*, who kill their host). Ivy, and most orchid flowers are typical examples of epiphytes plants. They live a life of their own, but need the presence of an existing living organism to grow on, with which they entertain a special kind of symbiosis. By analogy, our solution is to consider advisor systems as "epiphyte" systems, i.e. as systems growing onto other systems without perturbing them whatsoever.

4.2. Viewpoints as task trees

We designed EpiTalk [Paquette & al. 95], a framework and a system that proposes to organize spied information according to several viewpoints on the activity of the spied system.

In this scheme, the action analyzer is based on the exploitation of a pre-determined set of *task trees*. Here, a task trees is a hierarchical decomposition of a task into sub - tasks. Terminal tasks correspond to actual actions performed by the host system (or by a user interacting with the host system). These actions themselves correspond to a set of *messages* sent to particular *objects*.

The strong assumption of the EpiTalk architecture is materialize viewpoints on the activity of the host system by the task trees themselves. More precisely, task trees are used to generate automatically an isomorphic structure, called the *adviser tree*. This structure is in charge of analyzing user's actions and produce advice.

Figure 2 shows a task tree for the viewpoint on a tutorial system aimed at guiding students to discover scientific laws such as "PV-nRT". The host application is a set of tools such as tracers, simulators, graphers and spread sheets. The tutorial systems impose very few constraints on the order in which the user may use the tools. In this tutorial system, an advisor could focus either on the user's reasoning process or on the validity and structure of the law proposed. Each point of view is represented by distinct task tree.

The task tree in Figure 2 represents the viewpoint on the "reasoning process" of the student. At the first level, the root task is "T1.-Induce a Law". This task is decomposed at the second level into five sub tasks: "T2.1-Generation of Observation Sets", "T2.2-Analysis of Observation Sets", "T2.3-Conjecture Formulation", "T2.4-Conjecture Revision", "T2.5-Generalization". The sub task "T2.2-Analysis of Observation Sets" is in turn decomposed into "T3.1-Plot Data", "T3.2-Sort Data", "T3.3-Identify Tendency" etc. The precedence order of sub tasks is also specified in the task tree, for each task. Terminal tasks (here at the third level) identify tools provided by the host system, e.g. plotters and simulators. Intermediate tasks represent abstract tasks (levels 1 and 2) with no direct reference to a particular tool.

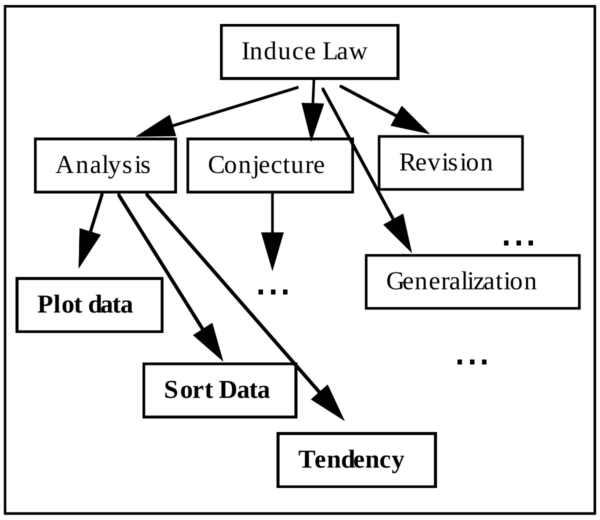

Figure 2. A Task tree on a tutorial system for scientific law discovery. Terminal tasks are in boldface, and contain descriptions used to generate spies automatically.

4.3. Connection with the spying system

The connection with the spying system described above is performed at the action (i.e. terminal task) level. To each terminal task of the task tree is associated the set of messages deemed interesting to spy. When a spy spies a message, it sends it to the terminal advisers, corresponding to the terminal tasks that are interested in this message. Spies are then used to feed the spying system, from the interaction of a user with the host system.

In our example the terminal task "Plot data" specifies which objects should be spied (here an instance of class `Plotter`"), and which messages should be intercepted (here messages `accept`, `undo` and `openTool`). At run time, spies are automatically generated from these descriptions and grafted on the host application.

4.4. Action analysis and advice production

The adviser module is in charge of observing the user action and provide advice such as the following ones:

1- Your proposed law has not been validated; before that, you should try to produce and analyze more observations.

2- Your analysis is not complete since there is an observation set that you have not looked at, in graph or table form.

3- Your selection will create an observation set that will be difficult to analyze because you have too many variables.

4- A law has to be expressed as an equation. Your law expression should therefore contain the symbol "=".

As these examples show, there are several "hierarchical levels of abstraction" of advice. Advice do not necessarily address the same conceptual level, within a given viewpoint. This organization of knowledge into levels is hierarchical by nature. Some advice are issued according to local and ephemeral information (#3-4). Others require more information, only available from a higher (or more global) level on the user's activity (#2). Others manipulate abstract information (#1) which are itself the result of lower levels inferences.

4.5. The plan recognition scheme

The plan recognizer we use is built in the spirit of [Quast 93], as a bottom-up spreading of spied information. The main characteristics of this scheme is that the plan recognition and the production of advice are combined into a single walk through the adviser tree. The principle is simple: each time a spy intercepts a message, it sends it to the corresponding terminal advisers. Then a bottom-up spreading is activated as follows:

1) Each adviser (terminal or non-terminal) processes the information, either to issue local advice or to update a local model of the activity being observed,
2) The adviser transmits to its direct father any information it considers relevant.

This scheme is applied recursively for all advisers of the tree, terminal or non-terminal, until the root adviser is reached. Terminal advisers receive information directly from the host system, whereas non-terminal advisers receive information from advisers below them in the hierarchy.

Each adviser manages a local model of the activity. This model consists mainly in a management of *states* for the task being performed. In simple cases, a state having three values (e.g. #inactive, #pending, #finished) suffices to produce advice. However, since each adviser is responsible for managing its own model, more sophisticated representation of the task activity may be introduced.

4.6. Example of advice production

In our advisor module for scientific discovery, the following situations can occur:
1) At the lower level, when the user groups sorted observations into a table, the adviser associated to the task "T3.2-Sort Data" is informed that the user is building an observation set and the adviser also knows the variables governing this set (such as the number of variables used, their names, etc.). From this local information, the adviser can produce advice #3.

2) Suppose that the user first tries to identify a tendency using the tendency tool. The adviser associated to the terminal task "T3.3-Identify Tendency" is then notified of the corresponding user action, and processes this information, either to issue a "local" advice such as advice #3, or to update its local model. Then the adviser sends a signal to its father - the adviser associated to the task "T2.2-Observation Set Analysis" - that the task "T3.3-Identify Tendency" has begun. The father accepts this information, and may update its local model to deduce that this is the first sub task accomplished by the user. He may consequently generate advice #2. In turn, this adviser will also transmit a signal to its father (here, the root adviser).

3) In a similar fashion, the root adviser associated to the task "T1.-Induce a Law" can monitor the whole induction process and generate advice such as advice #1.

This bottom-up interpretation of the adviser tree naturally reflects the various levels of abstraction: as information sifts up in the tree, it is processed by advisers that interpret it according to their own local vision of the host system and user's actions. When a user action is detected in the host system, it is sent to the terminal advisers that are interested in this action. These terminal advisers process the information and transmit information to their fathers, eventually reaching the root adviser. Each terminal adviser has a local vision of the system. Intermediate advisers represent intermediate aspects of the system's activity according to the given viewpoint. Only the root adviser of the tree has a global vision of the system. This hierarchical structure of advisers together with the communication scheme is the backbone of the advisor system, for a given viewpoint.

Since the specification of the advising module is entirely represented in the (decorated) task tree, several viewpoints for a given host system will correspond to several task trees, which will generate several different multi-agent advisor systems.

4.7. Applications of EpiTalk

EpiTalk is being used in a number of tutorial systems such as the scientific law discovery system described here, and the DEW system [Paquette & al. 94], a didactic engineering workbench. A third interesting extension of EpiTalk in progress is a system that automates the production of advice that specifically address violations of precedence constraints between sub-tasks [Pachet & al. 95b]. In this system, specific advice such as "this sub-task is prematurely performed" or "finish this sub-task before starting this one" are automatically generated from the precedence constraints specified in the task tree.
Other applications of EpiTalk include an environment for debugging actor-like languages [Giroux & al. 94], and explanation-modules for expert systems. Extensions for dynamic typing of Smalltalk programs are also considered.

5. Conclusion

We extend the notion of plan recognition to the monitoring of arbitrary systems, not necessarily involving human actions. The main issue we address is the construction of plan recognizers on top of existing object-oriented applications which do not require modification of the application's code. The spying technique we propose allows to graft plan recognition systems on top of arbitrary Smalltalk applications, without modifying their code. The spying technique provides a safe and practical mechanism for the "raw" first layer of an observation system. We described briefly the EpiTalk architecture, in which spied information is organized along task trees to generate relevant advice at various levels of abstraction.

However, the spying mechanism presented here is clearly independent of the actual advising module used to analyze spied information. Other plan recognition systems and advising strategies are currently being developed that feed from the same spying machinery, such as the "hieractor" model of [Kosbie & Myers 94], and a more classical approach based on attribute grammars.

References

[Carberry 90] Carberry, S. Incorporating Default Inferences into Plan Recognition. *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI)*, Boston, July 1990. pp. 471-478, (1990).

[Desmarais & al 93] Desmarais, M. C. Giroux, L. Larochelle, S. An Advice-Giving Interface Based on Plan-Recognition and User-Knowledge Assessment. *Int. Journal of Man-Machine Studies* (1993) 39, pp. 901-924. (1993).

[Foote & Johnson 89] Foote, B. Johnson, R.-E. Reflective facilities in Smalltalk-80. *Proceedings of OOPSLA'89*, pp. 327-336, New Orleans, Louisiana, (1989).

[Giroux & al. 94] Giroux, S. Pachet, F & Desbiens, J. Debugging multi-agent systems: a distributed approach to events collection and analysis. *Canadian Workshop on Distributed Artificial Intelligence - CWDAI '94*. Banff, Alberta, Canada, mai 1994.

[Greer & al. 93] Greer, J.E. Koehn, G. M. Rodriguez-Gomez, J. A System for Exploring Plan Recognition. *Proceedings of Artificial Intelligence in Education - 93*, Edinburgh, Scotland, pp. 465-472 (1993).

[Hoppe, 88] Hoppe, H.U. Task-Oriented Parsing - A Diagnosis Method to be used by Adaptative Systems. Proceedings of CHI '88, Washington, D.C., ACM Eds, pp. 241-247 (1988).

[Kosbie & Myers 94] Kosbie, D. Myers, B. Extending Programming By Demonstration With Hierarchical Event Histories. *Proceedings of East-West Human Computer Interaction Conference 94,* pp. 128-139, Springer-Verlag, Lecture Notes in Computer Science, n. 876 (1994).

[Krasner & Pope 88] Krasner, G. Pope, S. A Cookbook for Using the Model-View-Controller Paradigm in Smalltalk-80. *ParcPlace systems* (1988).

[Kautz & Allen 86] Kautz, H.A. Allen, J. F. Generalized Plan Recognition. *Proceedings of AAAI '86*, Philadelphia, Pa. pp, 32-37, (1986).

[Konolige & Pollack 89] Konolige, K. Pollack, M. Ascribing plans to agents. *Proceedings of the 11th IJCAI* , Detroit, pp. 924-930, (1989).

[Lieberman 86] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-

Oriented Systems. *Proceedings of OOPSLA '86*, Portland, Oregon, pp. 214-223, (1986).

[Pachet & al. 95] Pachet, F. Wolinski, F. Giroux, S. Spying as a novel object-oriented programming paradigm. *Proceedings of TOOLS Europe '95*, Versailles, France, pp. 109-118, (1995).

[Pachet & al. 95b] Pachet, F. Djamen, J.-Y. Frasson, C. Kaltenbach, M. Production de conseils pertinents exploitant les relations de composition et de précédence dans un arbre de tâches. *Submitted to Technique et Sciences Educatives* (1995b).

[Paquette & al 94] Paquette, G., Crevier, F. Aubin, C. Frasson, C. Design of a Knowledge-based Didactic and Generic Workbench. *Computer Aided Learning in Science and Engineering*, Paris, France, Sept. 1994.

[Paquette & al. 95] Paquette, G. Pachet, F. Giroux, S. EpiTalk: a Generic Tool for the Development of Advisor Systems. *AI in Education*. To appear (1995).

[Pascoe 86] Pascoe, G. Encapsulators: A New Software Paradigm in Smalltalk-80. *Proceedings of OOPSLA'86*, pp. 341-346, Portland, Oregon, (1986).

[Payne & Green 86] Payne, S. J. Green, T.R.G. Task-Action grammars - A model of the mental representation of task languages. Human-Computer Interaction, Vol. 2, pp. 93-133.

[Quast 93] Quast, Klaus-Jürgen. Plan recognition for context-sensitive help. *Proceedings of the International Workshop on Intelligent User Interfaces*, Orlando, Florida, Jan. 1993. ACM Press. pp. 89-96, (1993).

[Reisner 81] Reisner, P. Formal Grammar and Human Factors Design of an Interactive Graphic System. IEEE Transactions on Software Engineering, Vol. SE-7/2, pp. 229-240.

[Visual Works 94] VisualWorks Cookbook, version 2.0. ParcPlace Systems, (1994).