Propagation of Change in Object Oriented Programs

Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202
rajlich@cs.wayne.edu

Abstract

This position statement presents a prototype tool "Ripples 2" which supports two processes of propagation of ripple effect in Object Oriented systems: change-and-fix, and top-down propagation (MSE). The paper also contains an example of the use of the tool.

1. Introduction

At Wayne State University, we are approaching Object Oriented Reengineering from three different perspectives: Object Oriented redocumentation, propagation of ripple effects in object oriented systems, and restructuring object oriented systems by behavior preserving transformation. In this position statement, we present propagation of ripple effects and tool Ripples 2 which we implemented.

Change propagation is one of the key parts of reengineering and evolution. In order to explain change propagation, we have to understand that object oriented software consists of classes and their dependencies. The dependency between classes A and B means that class B provides certain services, which A requires for its correct function. Examples include composition, where one class is composed of instances of other classes, and inheritance, where one class inherits properties from another class. There may be additional dependencies among the classes. A dependency is consistent if requirements of A are satisfied by what B provides.

When a programmer makes a change in software, he starts by changing a specific class of the software. After the change, the class may no longer fit with the other classes of the software, because it may no longer provide what the other classes require, or it may now require different services from the classes it depends on. The dependencies which no longer satisfy the require-provide relationships are called inconsistent dependencies (inconsistencies for short), and they may arise whenever a change is made in the software. In order to reintroduce the consistency into software, a change propagation process keeps track of the inconsistencies and the locations where the secondary changes are to be made. The secondary changes, however, may introduce new inconsistencies, etc. The process in which the change spreads through the software is sometimes called the ripple effect of the change [19].

Programmer must guarantee that the change is correctly propagated, and that no inconsistency is left in the software. An unforeseen and uncorrected inconsistency is one of the most common sources of errors in software.

Change propagation is made easier by supporting tools and techniques which improve both the efficiency and quality of the process. This position statement presents a prototype tool Ripples 2 supporting change propagation, and an example of the use of the tool.

Ripples 2 is based on the model of change propagation of [20]. In literature, several formal models of change propagation have been proposed; see the overview in [1,2]. Particular attention has been paid to the prediction of the size and location of a change [6,7,13], and to the analysis of the dependencies in the software [4,8,9,10,12,15,18]. The process of change propagation has also been described, for example, in [11,19], but the model of [20] is different.

In the model, the evolution of the dependency graph is modeled as a sequence of snapshots, where each snapshot represents one particular phase in change propagation. In each snapshot, the dependencies are either consistent or inconsistent. A snapshot is changed into the next one by a change in one class, which changes some inconsistent dependencies into consistent ones and vice versa. A prototype tool "Ripples 2" described in Sections 2 and 3. Section 4 contains an example, and Section 5 contains conclusions.

2. Basic model

Let us start with a definition of a program with both consistent and inconsistent dependencies:

Let C be a set of *classes* of the program; for example, a set of classes. Then a *dependency* between two classes $a,b \in C$ is a labeled couple $D<a,b>$. $I<a,b>$ denotes an *inconsistency* between class a and class b, where b is to be updated. Then a *program* P is a set of dependencies and inconsistencies such that for every $I<a,b> \in P$, there is either $D<a,b> \in P$ or $D<b,a> \in P$. Set of classes of a program is $\mathrm{ent}(P) = \{a,b \mid D<a,b> \in P\}$, set of *marks* is the set $\mathrm{mark}(P) = \{b \mid \text{there exists } I<a,b> \in P\}$. Dependencies of the program is the set $\mathrm{depend}(P) = \{D<a,b> \mid D<a,b> \in P\}$, and inconsistencies of the program are $\mathrm{inconsist}(P) = \{I<a,b> \mid \in P\}$. A *consistent program* is a program P for which $\mathrm{inconsist}(P) = \mathrm{mark}(P) = \emptyset$.

In our model, we are assuming that all updates always change one specific class at a time. A step in change propagation is the replacement of an class and its neighborhood dependencies by an updated one. A process of change propagation is a sequence of such changes. We are assuming that all incoming inconsistencies were resolved by the change, i.e. after the change there are no longer any incoming inconsistencies.

For outgoing dependencies, we distinguish two cases: In the first case, the change does not propagate to any neighboring classes. In the other case, the change may propagate to all neighboring classes, which all have to be marked. We are not dealing with intermediate situations where some of the neighboring classes are marked and some are not. The new inconsistencies always point away from the class a which was changed, towards the neighbors. We also assume that the changes do not "bounce back", i.e. do not point to neighbors which were changed in the previous rounds and whose inconsistencies forced the change in class a..

In [16], we described a process called Methodology for Software Evolution (MSE). It is a top-down process where all changes always propagate from using classes to the providing classes. During the top-down process, a change always starts in the top classes; i.e., classes which do not support any other classes. This follows from the fact that all specifications are tied together in top classes, although parts may be delegated to supporting classes. Hence it is logical to look at the top as the first place where the change may be implemented. If the change is not needed there, then it can be localized in those supporting classes, which provide the functionality to be changed, etc.

A less restricted process is a change-and-fix process, where a change can start anywhere in the system, and propagates in all directions, i.e. both up and down.

The MSE process has more predictable properties than the change-and-fix process, and is therefore preferable whenever it is applicable. A more detailed description of the MSE process and an example can be found in [16], or a special case of MSE can be found in [17]. On the other hand, this process assumes that there are no loops in the dependencies of the classes. Since loops in dependencies among classes are present in some programs, this process is not always applicable.

3. Tool "Ripples 2"

"Ripples 2" is a prototype tool that supports both the Change-and-Fix and MSE processes. It is an updated version of an earlier tool "Ripples" [16], and a more detailed description can be found in [3].

The tool starts out with the user specifying the directory of the project. The external parser gen++ [5] parses the files of the directory and displays the dependencies between the classes. Then, depending on the mode and prior status of the project, Ripples 2 determines a set of marks.

If the user chooses not to modify a marked class because the code for the class is acceptable as is and the change is not propagating to the next classes, he can erase the mark. If on the other hand, the user chooses to change the code for a class, Ripples 2 brings up the text editor. Upon completion of changes to the class (including deletion), Ripples 2 invokes the scheduler to assess the impact of the change and derive a new set of marks.

Ripples 2 has three operating modes, listed and explained as follows:

Top-down process (MSE). The top down evolution process of MSE follows the process described in Section 2, and begins by the function main() being the first mark. When changes to this function are done, the classes that are used by function main are marked. After them, the classes they depend on are marked, etc. Among the marked classes, the scheduled classes are the only ones available for modification. The rest of the classes are not available for modification at any point. The change thus propagates from top to bottom, and ends when the last of the marked classes has been checked or modified.

Strict change-and-fix process. In this process, the change can begin anywhere in the system and propagates according to the process described in Section 2. The user is free to choose any of the classes for the first modification. After the first modification is performed, the user is forced to work only on the marked classes. When all the marked classes and their ripple effects have been taken care of, the system again returns to the consistent state. This approach differs from the top down approach in the following two assumptions:
- Ripple effects of changes propagate in all directions, not just downwards.
- The first change can begin anywhere, not just in the topmost class.

Random change-and-fix process. The random change-and-fix approach follows the same basic assumptions as the strict change-and-fix process. Changes begin anywhere in the system. Changes always propagate in all directions. The difference is in the fact that the user is free to modify any class at any time, even if it is not marked.

The strict process reduces the options which the programmer can use. Since errors are more likely when there isn't tight control, the strict mode would be the best in an unfamiliar project environment, and should always be used by a novice programmer. However the expert programmer may feel uncomfortable with the control exerted by the strict mode. For him, the random mode acts as a guide, helping him to be organized and not to forget any ripple effect, while allowing him the complete freedom to update any class.

An example of use of Ripples 2

In this example, we create an interactive TV guide from an existing program for an interactive calendar manager. The conversion was performed using Ripples 2 in the strict change-and-fix process mode, see a more detailed description in [3].

The source system. The source system is an interactive calendar manager, which keeps track of appointments. Each appointment has a start time and end time, and is entered into the system by the user. It is stored in a database to be retrieved when required. The system checks for overlaps and rejects overlapping appointments. User selections are made through the text based menus. There are no limitations on the number of events that the system can hold. Figure 1 displays the complete architecture of the source system. In it, all dependencies are depicted as edges, without distinction whether they are use, inheritance, or data flow dependencies. The source system consists of 19 classes and about 2,000 lines of code.

The target system. The target system is a TV guide system which keeps track of programs that will be aired on various TV channels. This TV guide is an interactive system into which the user enters data through menus. The user also can query the program for various channels at various times. In the target system, there will be overlapping programs, but in different channels. There cannot be any overlapping programs in the same channel.

Following is a detailed step-by-step explanation of the strict change-and-fix process that evolved the source system into the target system.

Step 1 : Adding class Channel. In the target domain, all television programs are classified according to the channel on which they are aired. So, the class Channel is very important as a distinguishing concept between the two systems. The class Channel will hold the channel number, the name of the channel, and the appropriate functions. The class was added using the Add option of Ripples 2.

Class Channel will be used in a currently nonexistent class ChannelList, which the user adds to the system with the help of Ripples 2. Ripples 2 generates the skeleton for ChannelList and marks it. Another class to be marked is the class Event, which will also use information from Channel.

Step 2 : Change in ChannelList. There can be multiple channels in the system, and there is no restriction on the number of channels that are allowed. Hence, class ChannelList represents a list of objects of type Channel. The code for this class contains functions to manipulate individual Channel objects and to perform collective operations such as sort, save, load, etc.

Class ChannelList is used by a currently nonexistent class ChannelMenu, which the user adds to the system with the help of Ripples 2. Ripples 2 generates the skeleton for ChannelMenu and marks it.

Step 3 : Change in ChannelMenu. This class manipulates the classes ChannelList using the add, delete, and modify functions, and allows the user to view all channels. Since the existing three menus are subclasses of AbstractSubmenu, ChannelMenu also is derived from AbstractSubmenu.

ChannelMenu is used by classes Choice and Cmenu and they will be marked. Another class which is marked is AbstractSubmenu. However the class AbstractSubmenu does not change, and hence after inspection the mark can be removed.

Step 4 : Change in CMenu. Class CMenu is one of the classes marked at this point, and it defines the main menu of both the source and target systems. This requires defining an object of type ChannelMenu as a data member of class CMenu.

Class CMenu is derived from AbstractMenu, which is therefore marked. Also Choice, YearMenu, DayMenu, and MonthMenu are marked, because class CMenu uses all these classes. None of these classes requires a change. Function main() is marked because it uses CMenu. There is a small change to be made to function main(). Choice is marked again as a result of that, but does not require a change.

Step 5 : Change in class Event. At this point, class Event is the only marked class in the program. It should now contain information regarding the program as well as the channel on which the program is aired. All functions of class Event have been modified to handle this change in the data members.

Event is one of the core classes of the source system. It supports Event_list, CFile, and Cbase. It uses CDate, Key, and CTime, and Channel. Of these, Channel, CDate, CTime, and Key do not change. But, Event_list, CFile, and CBase have to be changed in the following steps.

Step 6 : Changes in Event_list. Event_list creates a list of objects that belong to type Event. There are various places where Event_list makes references to functions that handle the newly added data members. Hence, these references have to be changed.

Classes Key, CFile, CBase, and CDay are marked after the change. Cday and Key are not affected and their marks are erased. CFile and CBase are the classes that handle the events in a disk file and database abstraction levels, respectively. They are affected and handled in the following steps.

Step 7 : Changes in CFile. Class CFile handles physical file i/o related to the events. The i/o functions are affected. They are modified to accommodate the changes made to Event_list.

Key is marked again after this change, but remains without a change. CBase is marked again and will be visited next.

Step 8 : Changes in CBase. CBase handles the events in a database abstraction level. It also handles the queries on the events. It is changed in response to the change in specifications.

Class Key is marked after the change, but remains without change.

Architecture of the target system. Please note that during the changes, three new classes were added, two existing classes were changed extensively, there were minor changes to several additional classes, and no existing class was deleted. Tool Ripples 2 was extensively used to monitor the propagation of the change.

4. Conclusions

In the paper, we presented a model of change propagation in software, together with a model of two different processes of change propagation: change-and-fix, and a more structured process MSE (methodology for software evolution). During the change, the software is represented as a graph of dependencies among the classes of the software, where some of the dependencies are inconsistent. Each change removes some inconsistencies, but it may create new ones. The modeling notation is based on graph rewriting.

As a partial validation of the model, we presented a tool "Ripples 2" which supports both the change-and-fix and the MSE processes of change propagation.

The tools which support change propagation are closely related to browsers. While browsers [4,15] deal with dependencies in software, they leave the particular query to the programmer. Change propagation tools differ from browsers in the fact that they maintain information about both dependencies and inconsistencies in software, and provide a specialized but important kind of query: Find all marked classes which have to be changed in order to make software consistent. These kinds of tools help the programmer to be organized during the process of software maintenance. We believe that they may play an important role in the future.

References

[1] R.S. Arnold, S.A. Bohner, Impact Analysis - Toward a Framework for Comparison, *Proc. Int. Conf. Software Maintenance*, 1993, 292-301.

[2] S.A. Bohner, R.S. Arnold, Software Change Impact Analysis, IEEE Computer Soc. Press, ISBN 0-8186-7384-2, 1996.

[3] S.S. Chandrasekaran, Change-and-Fix Software Evolution Using Ripples 2, M.S. thesis, Dept. of Computer Science, Wayne State University, Detroit, 1997.

[4] Y.F. Chen, M.Y. Nishimoto, C.V. Ramamoorthy, The C Information Abstractor System *IEEE Transactions on Software Engineering* Vol. 16, 1990, 325 - 334

[5] P. Devambu, "GENOA- A Language and Front-End independent source code analyzer generator", *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992, 307-317.

[6] K.B. Gallagher, Evaluating surgeon's assistant: Results of a pilot study. *Proceedings of the Conference on Software Maintenance* 1992, 236-255.

[7] K.B. Gallagher, J. Lyle; Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8), August 1991, 751-761.

[8] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12(1), January 1990, 35-56.

[9] J. Keables, K. Roberson, A. von Mayrhauser, Data Flow Analysis and Its Application to Maintenance, *IEEE Conference on Software Maintenance*, Phoenix, AZ, Oct. 1988, 335-347.
[10] J.P. Loyall, S.A. Mathisen, Using Dependence Analysis to Support the Software Maintenance Process, *Proc. Int. Conf. on Software Maintenance*, 1993, 282-291.
[11] Luqi, A Graph Model for Software Evolution, *IEEE Trans. on Software Engineering*, 1990, 917-927.

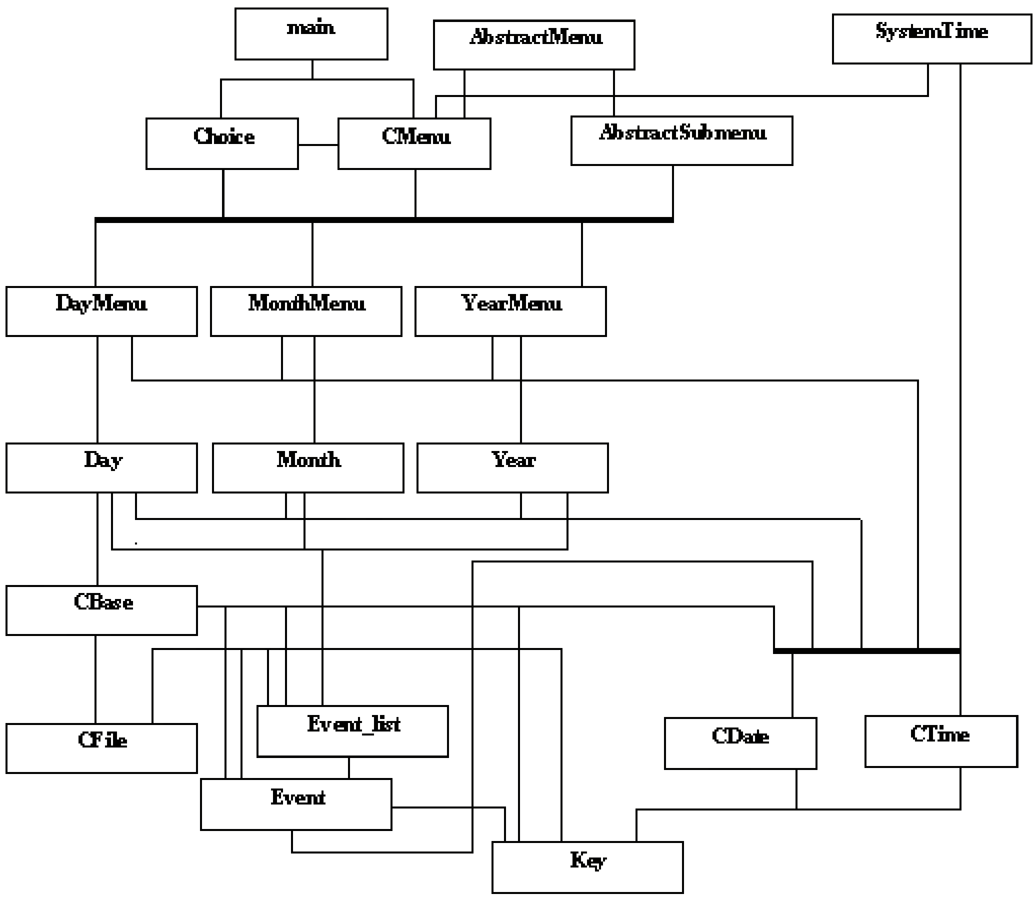

Figure 1. Architecture of the source program.

[12] K. Ottenstein and L. Ottenstein, The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1985, 177-185.
[13] P.P. Queille, J.F. Vodroit, N. Wilde, M. Munro, The Impact Analysis Task in Software Maintenance: A Case Study. *Proc. Int. Conf. on Software Maintenance*, 1995, 235-252.
[14] V. Rajlich, Theory of Data Structures by Relational and Graph Grammars, In Automata, Languages, and Programming, *Lecture Notes in Computer Science 52*, Springer Verlag, 1977, 391-511.
[15] V. Rajlich, N. Damaskinos, P. Linos, W. Khorshid, "VIFOR: A Tool for Software Maintenance," *Software Practice and Experience*, 20(1), January 1990, 67-77.
[16] V. Rajlich, MSE: A Methodology for Software Evolution, *Journal of Software Maintenance*, Vol. 9, 1997, 103-125.

[17] V. Rajlich, J. Silva,, Evolution and Reuse of Orthogonal Architectures, *IEEE Trans. On Software Engineering*, 22(2), 1996, 153-157.

[18] N. Wilde, R. Huitt, Maintenace Support for Object-Oriented Programs, *Proc. Conf. on Software Maintenance*, 1991, 162-170.

[19] S.S. Yau, R.A. Nicholl, J.J. Tsai, S. Liu, 'An Integrated Life-Cycle Model for Software Maintenance', *IEEE Trans. Software Engineering*, 15(7), 1988, 58-95.

[20] V. Rajlich, A Model for Change propagation based on Graph Rewriting, to be published in Proc. 1997 IEEE International Conference on Software Maintenance, see also http://www.cs.wayne.edu/~vtr/Propagation.ps