A Model for Change Propagation Based on Graph Rewriting

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

Abstract

This paper presents a model of change propagation during software maintenance and evolution. Change propagation is modeled as a sequence of snapshots, where each snapshot represents one particular moment in the process, with some software dependencies being consistent and others being inconsistent. A snapshot is changed into the next one by a change in one software entity and the dependencies related to it. The formalism for this process is based on graph rewriting. The paper discusses two basic processes of change propagation: change-and-fix, and top-down propagation. It also describes a prototype tool "Ripples 2" which supports both processes, and an example of the use of the tool.

1. Introduction

Change propagation is one of the key parts of software maintenance and evolution. In order to explain change propagation, we have to understand that software consists of entities (classes, objects, functions, etc.) and their dependencies. The dependency between entities A and B means that entity B provides certain services, which A requires for its correct function. Function call is an example of a dependency among functions. Examples of dependencies among classes include composition, where one class is composed of instances of other classes, and inheritance, where one class inherits properties from another class. Different programming languages or software systems may consist of different entities and dependencies. The 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 entity of the software. After the change, the entity may no longer fit with the other entities of the software, because it may no longer provide what the other entities require, or it may now require different services from the entities 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].

For the software maintainer, change propagation is a key process. The maintainer 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. However, in order to develop effective tools, the problem must be correctly understood. This understanding is best formulated in terms of a formal model, which extracts the essential properties and separates them from accidental ones. Each model is based on a certain set of assumptions, both explicit and implicit. The model creates the foundation on which the tool developer bases the tools and techniques. The implementation of a prototype tool is a first validation of the model, and of the abstractions and assumptions built into it. In the final phase, validation of the tool must be made in a practical setting, where the usability, practical importance, effectiveness, etc., is assessed. Needless to say, all three phases are important for progress in the area of software tools, and there is no real conflict between them. This

paper presents both the model and a prototype tool based on it. Furthermore, it presents an example of the use of the tool.

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]. This paper does not deal with the first topic, and uses the results of the second topic as a starting point, i.e. assumes that the dependencies in the program are known. The process of change propagation has also been described, for example, in [11,19], but here we provide a more complete treatment.

In our 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 entity, which changes some inconsistent dependencies into consistent ones and vice versa. The formalism for this process is based on the graph rewriting of [14], and is presented in Sections 2, 3, and 4. As a step towards validation of the model, we implemented a prototype tool "Ripples 2" described in Section 5. Section 6 contains an example of the use of the tool. Section 7 contains conclusions.

2. Basic model

The basic definition of this section is a definition of a program with both consistent and inconsistent dependencies. Formally, this is expressed in the following way:

Let C be a set of *entities* of the program; for example, a set of classes. Then a *dependency* between two classes a,b ∈ 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> ∈ P, there is either D<a,b> ∈ P or D<b,a> ∈ P. Set of entities of a program is ent(P) = {a,b | D<a,b> ∈ P}, set of *marks* is the set mark(P) = {b | there exists I<a,b> ∈ P}. Dependencies of the program is the set depend(P) = {D<a,b> | D<a,b> ∈ P}, and inconsistencies of the program are inconsist(P) = {I<a,b> | ∈ P}. A *consistent program* is a program P for which inconsist(P) = mark(P) = ∅.

Graphically, we denote the programs as directed graphs, where directed arcs represent dependencies, and additional large arrows represent inconsistencies pointing in the direction of change, see Figures 2 and 3.

Examples of the entities are classes in C++, and functions and global variables in C. Examples of dependencies are inheritance, use, friendship, etc. in C++, or calls among the functions, and references of functions to global variables in C. Other languages have different entities and dependencies. In our model, we are assuming that all updates always change one specific entity at a time. This is captured in the following definitions:

Let a ∈ ent(P). Then

T(a) = {D<b,a>| D<b,a> ∈ P} (top dependencies of a)
B(a)= {D<a,c>| D<a,c> ∈ P}(bottom dependencies of a)
I(a)={I<b,a>| I<b,a>∈ P}(incoming inconsistencies of a)
O(a)={I<a,c>| I<a,c>∈ P}(outgoing inconsistencies of a)
P(a) = T(a) ∪ B(a) ∪ I(a) ∪ O(a) (neighborhood of a)

A step in change propagation is the replacement of an entity and its neighborhood by an updated one. This is formally defined in the following way: Let P be a program before a change, and P' be the same program after the change. Denote P(a) to be the neighborhood of an entity a before the change, and P'(a) to be the neighborhood of entity a after the change. Then

P' = (P-P(a)) ∪ P'(a).

In other words, a step is a replacement of just one entity and its neighborhood. Please note that the formula for the change follows the notation of [14]. It simplifies the algebraic complexities of the graph rewriting, and still captures all essential properties of graph rewriting which are needed for the model.

3. Change-and-fix process

In the previous section, we defined formally a change in one entity of a program. A process of change is a sequence of such changes. In this section, we are going to explore one such process, called the change-and-fix process. In this process, P'(a) = T(a) ∪ B'(a) ∪ O'(a). We are assuming that all incoming inconsistencies were resolved by the change, i.e. after the change there are no longer any incoming inconsistencies and I'(a) = ∅.

For outgoing dependencies, we distinguish two cases: In the first case, the change does not propagate to any neighboring entities. In that case, O'(a) = ∅, and P'(a) = P(a) - I(a). In the other case, the change may propagate to all neighboring entities, which all have to be marked. We are not dealing with intermediate situations where some of the neighboring entities are marked and some are not. The new inconsistencies always point away from the entity 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 entity a. Formally,

$O'(a) = \{I<b,a> \mid D<a,b> \in P'(a)$ or $D<b,a> \in P'(a)\}$ - $\{I<a,c> \mid I<c,a> \in P(a)\}$.

We also assume that $T'(a) \subseteq T(a)$, i.e. the change in entity a does not remove any top dependencies (i.e. the entities dependent on a will still be dependent on a even after the change). However the change in entity a may change bottom dependencies in an arbitrary manner, i.e. either $B'(a) = B(a)$ (a will be dependent on the same entities), or $B'(a) \neq B(a)$ (the general case where entity a will be dependent on a different set of entities after the change), with the special cases $B'(a) \subset B(a)$ (entity a will be dependent on fewer entities after the change), and $B'(a) \supset B(a)$ (entity a will be dependent on more entities after the change).

The process of change propagation is defined in the following way:

Change-and-fix process of change propagation

given a consistent P;
select a;
change(a);
$P = (P - P(a)) \cup P'(a)$;
do {
   select $a \in$ mark(P);
   change(a);
   $P = (P - P(a)) \cup P'(a)$;
} while (mark(P)!= $\varnothing$);

Example

Let us consider a process of modification for the telephone directory from [16]. The phone directory uses a file of names and phone numbers, which the program reads and then creates an internal data structure. When the user enters a name of a person, the program searches the data structure for the phone number. Figure 1 shows the entities of the program P_0, where "Main" controls the execution of the program, "Input" is the function through which the user inputs the name, "Data" is the data structure containing the names and phone numbers which is being initialized by "Init" and searched by "Search". The function "Search" implements binary search where function "C" implements alphabetical comparison of two names, needed for the binary search. Formally, ent(P_0) = {Main, Input, C, Search, Data, Init}, dependent(P_0) = {D<Main,Input>, D<Main, Search>, D<Main,Init>, D<Search,Input>, D<Search, C>, D<Search,Data>, D<Init,Data>}, inconsist(P_0) = mark(P_0) = $\varnothing$.
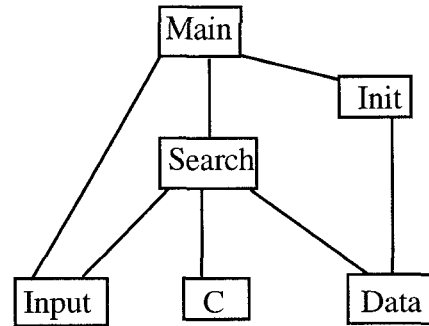

Figure 1

The change in the program is the change from the binary search to a hash search. As the first step, the algorithm of function "C" is changed into a hashing algorithm, and C returns a hash value for a given name. The resulting situation is pictured in Figure 2. In it, program P_1 has entities ent(P_1) = ent(P_0), dependent(P_1) = dependent(P_0), inconsist(P_1) = {I<Search,C>}, and mark(P_1) = {Search}. Please note that the inconsistency is denoted in Figure 2 by large arrow.
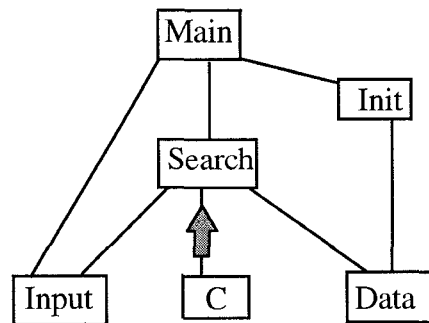

Figure 2

In the next step, the function Search must be changed, because it uses a new algorithm. The program after the change is in Figure 3. In it, ent(P_2)= ent(P_0), depend(P_2) = depend(P_0), inconsist(P_2) = {I<Search, Data>, I<Search,Main>, I<Search,Input>}, and mark(P_2) = {Input, Main, Data}.
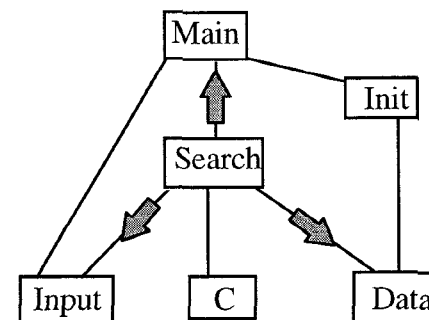

Figure 3

86

In the next step, the marked entity "Input" is checked and left without change because it still has the same implementation. Also, the interface of function "Search" remains the same, since the change involves only the internal algorithm and not the interface of the function, and therefore there is no change of "Main". However data structure "Data" must be changed, since the data structure for hash search is different from the data structure for binary search. The situation after the change is depicted in Figure 4, where $ent(P_3) = ent(P_0)$, $depend(P_3) = depend(P_0)$, $inconsist(P_3) = \{I<Data, Init>\}$.
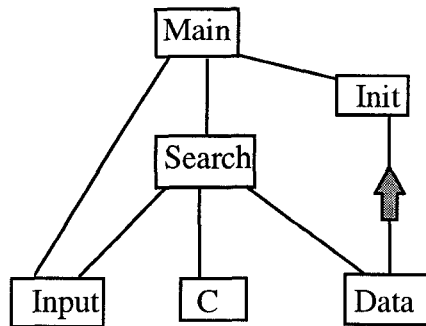

Figure 4

In the next step, entity "Init" is changed and "Main" is marked. Finally, "Main" is checked and left unchanged because the prototype of function "Init" remained the same, and the program becomes consistent again.

The Change-and-fix process is the most common process of software maintenance. However many of its properties are unclear. For example, it is conceivable that the entities will be visited several times or even that there may be an infinite process, where the change propagates in a circle and repeatedly revisits the previously changed entities. Therefore, in the next section we are going to model a more predictable process, where changes propagate only from top-down.

4. MSE: a top-down change propagation

The process was originally described in [16] and called Methodology for Software Evolution (MSE). A special case of this process was described in [17]. Let us start with the following definitions:

Let P be a program, where there are no loops among the dependencies. Then, define bottom slice in the following way: $B^*(a) = \{c \mid c = a$ or there exists $D<e,c> \in P$ such that $e \in B^*(a)\}$. Scheduled entities are the "highest" marks defined in the following way: $scheduled(P) = \{c \mid c \in mark(P)$ and if $d \in mark(P)$, $c \in B^*(d)$, then $c = d\}$. Also define top entities $top(P) = \{c \mid$ for no $b \in ent(P)$, $D<b,c> \in P\}$.

During the top-down process, a change always starts in the top entities; i.e., entities which do not support any other entities. This follows from the fact that all specifications are tied together in top entities, although parts may be delegated to supporting entities. 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 entities, which provide the functionality to be changed. The change propagates top-down to supporting entities, i.e. $O_T'(a) \subseteq B'(a)$, $P_T' = O_T'(a) \cup T(a) \cup B'(a)$. In order to avoid multiple visits to entities, we always visit the scheduled entities of set scheduled(P) only. The process is described in the following way:

Top-down process of change propagation (MSE)

given a consistent P;
select $a \in top(P)$;
change(a);
$P = (P - P(a)) \cup P_T'(a)$
do {
   select $a \in scheduled(P)$;
   change(a);
   $P = (P - P(a)) \cup P_T'(a)$
} while $(mark(P) != \varnothing)$;

This process always terminates. For proof, consider the number of classes in the union of all bottom slices for all marks (i.e. $|\cup_{a \in mark(P)} B^*(a)|$.) This number is always greater or equal to 0. This number decreases by one in each loop. Since the set of integers greater or equal to 0 is well founded and does not contain an infinite decreasing sequence, the loop terminates.

The MSE process has more predictable properties than the change-and-fix process of the previous section, 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 entities. Since loops in dependencies among entities are present in some programs, this process is not always applicable.

5. 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]

87

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 4, 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 (see section 4) 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 3. 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 the use of the tool Ripples 2 is in the next section.

6. 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 5 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.
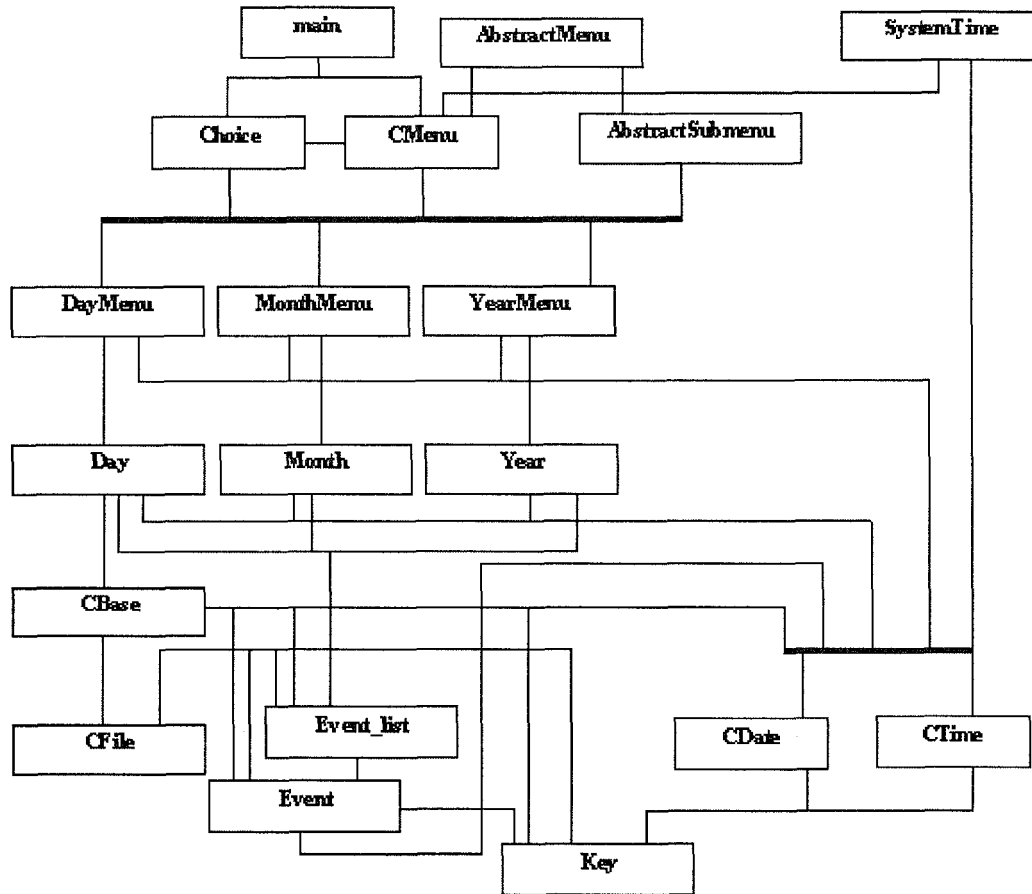
Figure 6. Architecture of the source program.

7. 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 entities 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 entities 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 Appplication 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.

[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.

[15] 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.