

Automating Software Evolution

David Hearnden, Paul Bailes
School of ITEE
University of Queensland
Australia
hearnden@itee.uq.edu.au

Michael Lawley, Kerry Raymond
Distributed Systems Technology Centre (DSTC)
Australia

Abstract

Software maintenance and evolution are the most expensive activities in the software process, consuming 60% to 80% of the total time spent on a software system. However our understanding of maintenance activities has barely developed beyond arbitrary change to arbitrary things. The standard categories of maintenance are based on subjective characteristics (purpose), rather than objective attributes. Only by understanding the relationships and dependencies between entities in the software process (such as specification, design and implementation) can we begin to objectively categorise and potentially automate aspects of software evolution.

1. Introduction

1.1. Evolution

Many studies have shown that the maintenance phase constitutes 60% to 80% of the total effort spent on a software product [1, 5, 12]. Contemporary iterative methodologies place even more importance on maintenance and evolution, to the point where software engineering is a process of continual evolution [13, 6].

1.2. State of the problem

Given the crucial role of evolution and maintenance in contemporary software engineering, we need mechanisms to increase the speed and the reliability (which will ultimately decrease the cost) of maintenance. To do this, we must strive to automate as many maintenance tasks as possible.

Ground-breaking work has been conducted on studying the effects of evolution (such as Lehman's Laws of Software Evolution [10]), however there is still little coherency in

evolution *practice*. It is hard to objectively classify software evolution as anything more detailed than arbitrary changes to arbitrary things.

The problem with automating maintenance tasks is that in order to automate maintenance, we must first understand the mechanics of how maintenance affects the various artefacts of a software system. However, the standard classifications of maintenance types are based on human-centric semantics, such as *purpose*. The ISO/IEC Standard for Software Maintenance [8] and the IEEE Standard for Software Maintenance [7] define five types of maintenance activities between them: *adaptive*, *corrective*, *perfective*, *preventive* and *emergency*.

The problem with this classification is that it is not based on objective characteristics of the changes that are made, but rather the *purpose* of the change. This deficiency has been noted in [9].

One way we can add more useful information into our description of maintenance tasks is to realise that some changes are *caused* by other changes, and furthermore, some changes are *determined* by other changes. Our intuitive approach to automation is to specify *causal* or *root* information and to expect machinery to compute *consequent* information. If we are able to analyse a set of changes and identify what the causal changes are, we can begin to see that set of changes in a more structured way, allowing us to better understand those maintenance tasks. For example, we may classify implementation-level changes as being *caused* by design-level changes. Furthermore, if we are able to identify dependencies between software artefacts that we wish to maintain, then not only can we identify causal relationships between changes to those artefacts, but we may also *determine* one change from another. Thus we are interested in *dependency relationships* between artefacts in order to identify where a set of root changes may induce other changes. We are also interested in those dependencies which are *functional*, meaning that the induced changes are not only caused by *determined* by root changes.

What we see when we start to explore the world of au-

tomated evolution, is that the current classification of maintenance tasks is orthogonal to the information we require for automation. Our current classification of maintenance is based on the semantic attribute of *purpose*, and offers suggestions on neither how change should be expressed nor what process should be followed to make that change.

Before we can automate evolution, we must first be able to express what we mean by *evolution* such that a machine can interpret what we mean. This means we need an *objective* view of evolution. When a system evolves, there are certain relationships between software entities that must be maintained. The implementation must ‘conform’ to the design, it must ‘satisfy’ the requirements, it must ‘pass’ the test cases. When one of these entities changes, there are consequent changes to the other entities in order to maintain those relationships. Thus we could describe evolution as the process of maintaining the dependencies between software artefacts after some have been changed.

2. Objectivity

By *objective*[3], we mean ‘of or having to do with a material object’ or ‘having actual existence or reality’. Thus we can only talk about *objective* change when it is change to something *real*, so we can only reason objectively about changes to software artefacts. Software maintenance and evolution, in its essence, is about changing our *ideas* about software (since software is an abstract entity), and thus the correspondence between our changed ideas and how those changes are manifested on concrete artefacts relies very heavily on how we represent our ideas as software artefacts. Thus to understand objective change, we must first understand the relationship between *ideas* and *representation*.

2.1. Representation

As human beings, all our reasoning capabilities are based on thoughts and ideas. Our reasoning can only take ideas as input and produce ideas as output. There is no symbolic representation of our ideas, nor a symbolic representation of our reasoning processes.

Ideas must be codified in order for them to be transmitted to other people, or to be input into a machine. A piece of codified information we call a *sentence*. The transformation of ideas into sentences is the purpose of a *language*. A language has three aspects: *syntax*, *semantics*, and a *universe of discourse* (UoD).

For the purpose of this discussion, a language’s syntax defines the sentences that can be considered part of the language. Traditionally, this is mathematically defined as a set of symbol sequences, however to be slightly more general we treat the term to mean any concept that can distinguish sentences as being valid or invalid.

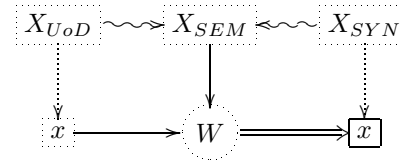


Figure 1. Transformation of an idea to its representation (writing).

The universe of discourse of a language is the conceptual domain of all ideas related to information expressed in that language. It includes all the ideas that are required for someone to be able to *understand* the meaning of a sentence in a particular language.

The semantics of a language relate the syntax to the universe of discourse. Language semantics are used both to ascribe meaning to a symbolic representation, but also to guide the transformation of ideas into a symbolic representation. Semantics are commonly described using a function from the syntactic domain to the universe of discourse.

Figures in this document are illustrated using *process graphs*, described in Table 1. The relationships between ideas, representation, syntax, semantics, and UoD are depicted in Figure 1.

Table 1. Process graphs

Construct	Meaning
x	The idea x
x	Representation of the idea x
$a \rightarrow f \Rightarrow y$	A real function / process / machine, $y = f(a, b)$
$b \rightarrow f \Rightarrow y$	An abstract function / thought-process, $y = f(a, b)$
$X \rightarrow x$	x is-of-type / element-of / is-an X . X describes x .
$X \rightsquigarrow Y$	Arbitrary dependency of Y on X . X is ‘used’ by Y .

In Figure 1, an idea x that belongs to some universe of discourse X_{UoD} is transformed into a representation of x that belongs to some syntactic domain X_{SYN} . This transformation relies on information that is part of the language semantics, X_{SEM} . Since language semantics relate a syn-

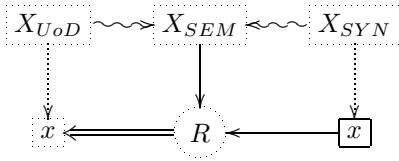


Figure 2. Interpretation of a representation (reading).

tax to a UoD, it is dependent on those concepts. The process of representation has been labelled W (for *Writing*) and interpretation has been labelled R (for *reading*), because they are commonly occurring processes and deserve special names. An example of this would be someone with a mental model of a Java program in their head who types out the symbolic representation of that idea. The idea belongs to the UoD of Java, and the representation belongs to the syntax (i.e. set of all sentences) of Java.

Because this transformation crosses the boundary of *real* and *abstract*, only a person can perform this action, not a machine, however it is simply the task of writing out an idea in one's head.

Although trivial, this process is not perfect, and by taking the process-graph perspective we have an objective way to distinguish between maintenance due to typographic errors and maintenance due to errors in understanding.

This understanding is important because objective characteristics only exists for objective things (i.e. representations), and thus the domain of objective change is representations (i.e. software artefacts). Furthermore, this is important because we can only automate (through machinery) tasks that operate on representations, we cannot deduce consequences from changes that occur to abstract entities. Thus our best hope for automated software evolution is by representing all the source information that was used to create a software system. This includes machinery that transforms source information (such as a system specification and design decisions) into an implementation.

Now that we have sophisticated techniques for building software in this manner (such as the Model-Driven Architecture, generative programming, domain-specific languages and aspect-oriented programming) we can begin to entertain the idea of an advanced level of automation in software evolution.

2.2. Process graphs

Table 1 describes *process graphs*, which are used here to model dependencies between software artefacts, ideas and processes that may include human involvement. One

of the arguments of this paper is that such dependencies (*functional* and *type*) are a sufficient metaphor for describing the relationships between software entities. By explicitly representing these relationships, we can objectively and generically identify *where* ripple effects can manifest themselves. We can also begin to use automated reasoning (such as incremental computation) to compute *what* changes are required for affected artefacts.

This section shows how a number of different software construction techniques can be represented using process graphs, and how these graphs can be used to objectively identify consequent change.

2.2.1 Straight implementation

The most primitive software process (and also the process least conducive to maintenance) is where a software engineer simply writes software in order to satisfy requirements in his/her head. This process is depicted in Figure 3. The initial idea, s , would typically belong to some abstract UoD of requirements, specifications and constraints. Thus s would have come from analysis of the business problem being solved (not shown for brevity). Using knowledge of that UoD, S_{UoD} , knowledge of an implementation language UoD, L_{UoD} , and some pragmatic choices, C , the software engineer transforms (I) the original idea s into a mental model of a program, p . This mental model is then written into a representation of p .

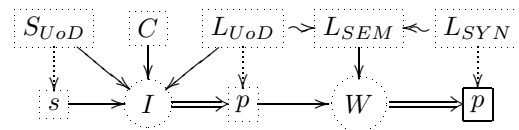


Figure 3. The process of implementing an idea.

The obvious problem with this process is that the UoDs of S and L are typically quite far removed, for example S_{UoD} might be *Behavioural-and-Functional-Specifications* and L_{UoD} might be *C-programs*. Thus the pragmatics, C , are a vital piece of the puzzle of explaining how p (which may be, for example, the conceptual model of a payroll program) relates to s (the conceptual specification of what that payroll program should do). However since s and C are concepts in the engineer's head, they have no objective characteristics, thus we cannot objectively model changes made to s or C . However we *can* objectively model the fact that p depends on s and C using this graph, even though we can not objectively reason in any way about how changes to those concepts may manifest as changes to p .

2.2.2 Documentation

The next level in sophistication is to concretise the idea of what the software is intended to be, i.e. *requirements* or a *specification* (note these terms are used in their general meaning, without any specific software-engineering connotations). This choice does present an inherent dilemma, since it is sometimes unclear whether the interpretation of the specification *artefact* is the true specification or the *idea* of the specification that was manifested as an artefact is the true specification (the letter of the law vs. the spirit of the law). Figures 4 illustrates the former of these processes. In Figure 4, the specification s initially exists as an artefact. The language in which s has been written reflects the UoD of s , and may be a special purpose specification language (such as B or Z) or may simply be natural language, since the UoD of natural language subsumes most of human thought. s is interpreted by a person and then input into the complex human-reasoning process, I , of transforming the idea s into an idea p in the universe-of-discourse of an implementation language L . This idea is then concretised (W) into a software implementation artefact p .

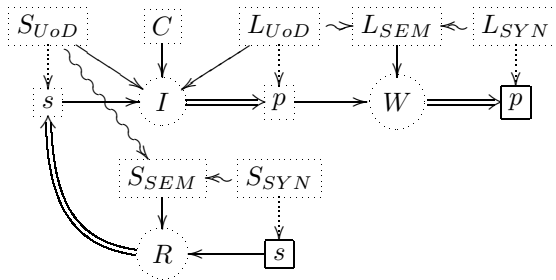


Figure 4. Documented specification.

The process graphs show us that even though the process by which s was transformed into p relies on conceptual information and processes, we can objectively identify a dependency of p on s that should be maintained. Changes to s may cause consequent changes to p . This process is more maintainable than that of Figure 3, however the missing link required to automate aspects of evolution in this process is an explicit representation of how s was transformed into p .

2.2.3 Compilation and Interpretation

The next step in maturing the process of software construction is to utilise machinery to perform the transformation of an element of one UoD to an element of another UoD. This *compiler* will often embody fixed design decisions and implementation patterns. Here we use the term *compiler* in its general sense to mean a transformer from one UoD to another UoD. This process is illustrated in Figure 5.

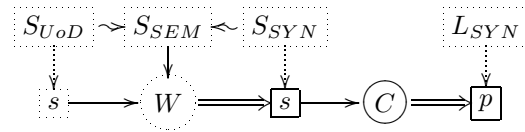


Figure 5. Compilation from one UoD to another

The compilation story is not quite complete, since it does not show how the compiler, C , comes into existence. Another approach to transforming from one UoD to another is illustrated in Figure 6. In this approach, a mapping m is written that describes the correspondence between the two UoDs, S_{UoD} and L_{UoD} . The mapping is written using knowledge of the two UoDs, knowledge of how conceptual elements map to syntactic constructs (S_{SEM} and L_{SEM}), plus some pragmatic choices, C . The mapping is codified as a syntactic translator that converts syntactic constructs in S_{SYN} to syntactic constructs in L_{SYN} . The mapping itself is written in a language for describing such inter-language mappings, M . The mapping m is run by process T that interprets it and enacts the rules contained within it to transform s to p . This is very similar to the model transformation approach in the Model-Driven Architecture, and demonstrates a high level of automation for the development process.

This process has a complex network of relationships between ideas and artefacts, however because we can *model* this complexity we can *reason objectively* about how maintenance tasks can affect the various artefacts in the system. For example, due to the functional dependency of p on s , changes to artefact s may require changes to the artefact p that can be objectively classified as being *consequent*. Also, changes to the mapping, m , may require changes to p that can also be classified as being consequent. More importantly, due to the high degree of machine involvement, we can automate a number of maintenance activities. Consequent changes to p due to changes in m or s can be found automatically.

3. Consequent Change

We have seen in the previous section that process graphs appear to be a sufficient metaphor for describing relationships between software ideas, artefacts and processes. It is therefore logical for our categorisation of objective change to be based on the concepts of these process graphs, namely ideas, representations, and dependencies (*functional*, *type* and *arbitrary*).

The arrows in a process graph are specifically con-

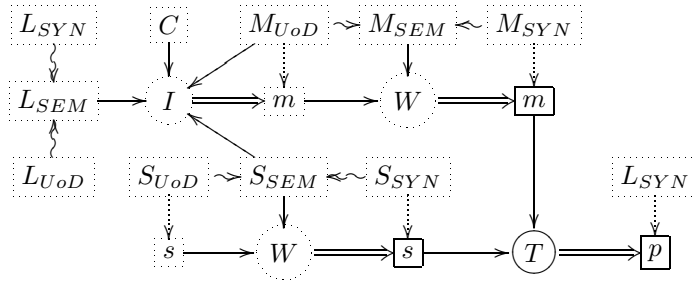


Figure 6. Using a generic interpreter, T , to enact the mapping in m on s .

structured to point in the direction of influence. Thus when anything in the graph changes, only things that are reachable by a path of the graph may be affected. We can thus objectively reason about consequent change.

We can also reason about the *kind* of consequent change by analysing characteristics of paths in the graph. This is important because the different kinds of dependency relationships have different mechanisms for maintaining them, some of which can be automated.

If changing artefact x causes a change to artefact y (i.e. y is reachable from x), then we can objectively classify the cause according to the relationship in the graph that is being maintained:

Functional Dependency If there exists a path of straight solid arrows from x to y , then y has a *functional dependency* on x . If all the processes that connect x to y are *real processes* (i.e. solid circles), then y has a *real functional dependency* on x . Otherwise, we say y has an *abstract functional dependency* on x .

Type dependency If there exists a dotted arrow from x to y , then y has a *type dependency* on x .

Arbitrary dependency For any other path from x to y , we say that y has an *arbitrary dependency* on x .

Functional and type dependencies are the most interesting kinds of dependency in software evolution, and an arbitrary dependency is simply the catch-all classification for all others.

3.1. Functional Dependency

A functional dependency asserts a deterministic relationship between source artefacts and target artefacts. If the source artefacts are changed, this may require changes to target artefacts in order to maintain the functional relationship. For example, we may have generated a web interface for an application using an interface generator that transformed a description of desired visual and interactive aspects of our system. If we update that description, then

the functional relationship is broken unless we make corresponding consequent changes to the (generated) web interface. Also, if we modify or replace the interface generator, then we must also make corresponding consequent changes to the web interface.

3.2. Type dependency

The second kind of dependency that can cause consequent change is when changes to ‘types’ affect changes to their elements. In software engineering we encounter two kinds of type dependency, *syntactic type dependency* and *semantic type dependency*. A syntactic type dependency exists between a symbolic representation and the syntax of the language that was used to represent it. A semantic type dependency exists between an idea and the universe of discourse that it belongs to.

For example, if we have an implementation of a payroll system, `payroll.c`, then we make use of the implicit knowledge that the implementation is a C program, thus the information in `payroll.c` is loosely dependent on what we mean by a C program. Imagine we change our C compiler. If our old C compiler accepted C++ style comments (`// . . .`) but the new compiler only accepts C-style comments (`/* . . . */`), then we feel compelled to *modify* any C++ style comments in `payroll.c`. Effectively, we have changed our definition of C syntax (by no longer accepting certain syntactic constructs), and we wish to *maintain* the syntactic type dependency, where `payroll.c` *conforms* to our version of the C syntax.

To illustrate semantic type dependency, consider the `payroll.c` example again. If the previous C compiler guaranteed left-to-right evaluation of function arguments but the new one does not, then we have effectively changed our universe of discourse of C, since our old version of C guaranteed certain semantics but the new version does not. Again, we feel compelled to modify our conceptual model of the program represented in `payroll.c` in order that it conforms to the new universe of discourse. This is because we wish to maintain the semantic type dependency.

Thus we can use the type dependencies in a process graph to objectively classify these kinds of consequent change.

4 Automation

By analysing a dependency relationship being maintained, we may be able to automate the maintenance of that relationship. In this section we discuss possible procedures for maintaining functional and type dependencies. Arbitrary dependencies, by their very nature, cannot in general be automatically maintained.

Functional Dependencies Maintaining real functional dependencies has a trivial brute-force solution: simply recompute the function. This has been the strategy of choice in similar domains such as build tools, where a change to source files results in various build steps being re-run to regenerate target files. If a representation of the changes made is required (perhaps for further propagation of change), then the new artefacts can be compared against the old artefacts, thus given *patch* and *diff* processes, we can trivially propagate change. However, we can be more efficient by utilising incremental computation ([2][4][11]) tailored to the computation paradigm used for the machine processes.

Type Dependencies In general, because the relationship between ‘type’ and ‘element’ is not a functional relationship (one is not determined by the other), then there is no general solution to knowing when and how to modify an element if its type changes. To solve the general problem, we need information that maps the elements of the old type to the corresponding elements of the new type where appropriate. In general, this mapping can not be derived automatically, however there are common subcases of the general problem where the process may be partially or completely derived based on common-sense policies, such as laziness. If the type relationship still exists with the new type, then we may simply leave elements unchanged. We can therefore reason about the effects of type change on an element-by-element basis.

Arbitrary Dependency Because there is no information on the nature of an arbitrary dependency, there is no scope for automating any consequent changes across an arbitrary dependency. These changes must be derived manually, however they may still be objectively classified.

5 Conclusion

Process graphs can be used as an objective basis for classifying change, and for deciding how to go about making consequent change to software artefacts. Systems such as Figure 3 and Figure 4 have vital processes that are complex and that only people can enact. Compare this to a system such as Figure 6, where even though there are still complex abstract processes, much of what was previously abstract is now concrete. Thus combining domain-specific and DRY (Don’t Repeat Yourself) principles with an evolution methodology based on maintaining dependencies is a very promising roadmap towards automated software evolution.

6 Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government’s CRC Programme (Department of Education, Science and Training).

References

- [1] P. B. Carroll. Computer glitch: patching up software occupies programmers and disables systems. *Wall Street Journal*, Jan 1988.
- [2] S. Ceri and J. Widom. Deriving Incremental Production Rules for Deductive Data. *Information Systems*, 19(6):467–490, 1994.
- [3] The American Heritage Dictionary of the English Language, 2000.
- [4] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD*, pages 157–166, 1993.
- [5] M. Hanna. Maintenance burden begging for a remedy. *Data-mation*, pages 53–63, April 1993.
- [6] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- [7] IEEE 1219 Standard for Software Maintenance, 1998.
- [8] ISO/IEC 14764 Standard for Software Maintenance, 1995.
- [9] B. Kitchenham, H. Travassos, A. von Maryhauser, F. Niessink, N. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an Ontology of Software Maintenance. *Journal of Software Maintenance: Research and Practice*, 11(6):365–389, 1999.
- [10] M. M. Lehman. Laws of Software Evolution Revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.
- [11] Y. Liu. Efficient Computation via Incremental Computation, 1998.
- [12] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 1998.
- [13] I. Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2000.