JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2003; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2002/09/23 v1.02]*

Research

Towards a taxonomy of software change

Jim Buckley*[,1], Tom Mens[2], Matthias Zenger[3], Awais Rashid[4], Günter Kniesel[5]

[1] *University of Limerick, Ireland*
[2] *Université de Mons-Hainaut, Belgium*
[3] *EPFL, Switzerland*
[4] *Lancaster University, UK*
[5] *University of Bonn, Germany*

SUMMARY

Previous taxonomies of software change have focused on the purpose of the change (i.e. the *why*) rather than the underlying mechanisms. This paper proposes a taxonomy of software change based on characterizing the mechanisms of change and the factors that influence these mechanisms.

The ultimate goal of this taxonomy is to provide a framework that positions concrete tools, formalisms and methods within the domain of software evolution. Such a framework would considerably ease comparison between the various mechanisms of change. It would also allow practitioners to identify and evaluate the relevant tools, methods and formalisms for a particular change scenario. As an initial step towards this taxonomy, the paper presents a framework that can be used to characterize software change support tools and to identify the factors that impact on the use of these tools. The framework is evaluated by applying it to three different change support tools and by comparing these tools based on this analysis.

1. Introduction

Evolution is critical in the life cycle of all software systems, particularly those serving highly volatile business domains such as banking, e-commerce and telecommunications. An increasing number of evolution mechanisms and tools are becoming available and many systems are being built with some change support in place. Because of this, there is a need for a common vocabulary and conceptual framework to categorize and compare the change support offered by these various tools and techniques.

*Correspondence to: Jim Buckley, University of Limerick, Castletroy, Limerick, Ireland

More than two decades ago, Lientz and Swanson [35] proposed a mutually exclusive and exhaustive software maintenance typology that distinguishes between *perfective*, *adaptive* and *corrective* maintenance activities. This typology was further refined by Chapin *et al.* [9] into an evidence-based classification of 12 different types of software evolution and software maintenance: evaluative, consultive, training, updative, reformative, adaptive, performance, preventive, groomative, enhancive, corrective and reductive. This earlier work is important and relevant, in that it categorizes software maintenance and evolution activities on the basis of their *purpose* (i.e., the *why* of software changes).

In this paper, we will take a complementary view of the domain, by focusing more on the technical aspects, i.e., the *how*, *when*, *what* and *where*, of software change. This focus is used to propose a taxonomy of *characteristics of software change mechanisms and the factors that influence these mechanisms.* By *taxonomy* we mean "A system for naming and organizing things . . . into groups which share similar qualities" [7]. By *change mechanisms* we refer to the software tools used to achieve software evolution and the algorithms underlying these tools (although it is intended that this taxonomy should be extended to consider the formalisms used and the methods employed to carry out software change).

The purpose of this taxonomy is manifold: (1) to position concrete software evolution tools and techniques within the domain; (2) to provide a framework for comparing and combining individual tools and techniques; (3) to identify relevant evolution tools given a specific maintenance or change context; (4) to allow evaluation of a software evolution tool or technique for a particular change context and thus, (5) to provide an overview of the research domain of software evolution. Each of these purposes is essential, given the proliferation of tools and techniques within the research field of software evolution.

2. Dimensions as Characterizing and Influencing Factors

Table I identifies the proposed dimensions of the taxonomy and defines, for each of these dimensions, whether it characterizes the mechanism of change, or whether it influences the mechanism of change (or both). We stress that the taxonomy should not be considered exhaustive or a finished work. In the first instance, we deliberately did not address all possible aspects of software change. For example, the *why* question is excluded since it has been treated in [35, 9]. Likewise, the *who* question, which identifies the stakeholders involved in software change, is not addressed as it would be impossible to develop comprehensive categories that cover the variety and skill-set of all the stakeholders involved in maintenance tasks. In the second instance, the taxonomy itself is subject to continuous evolution, since the elements that it classifies continue to evolve, due to scientific and technological advances in the field of software development and evolution. The following two subsections detail the rationale adhered to when deciding if the dimensions were classified as characterizing the mechanism or as influencing factors.

Dimension	Section	Characterizing Factor	Influencing Factor
time of change	3.1.1	•	•
change type	3.4.3	•	•
change history	3.1.2	•	
degree of automation	3.4.1	•	
activeness	3.3.2	•	
change frequency	3.1.3		•
anticipation	3.1.4		•
artifact	3.2.1		•
granularity	3.2.2		•
impact	3.2.3		•
change propagation	3.2.4		•
availability	3.3.1		•
openness	3.3.3		•
safety	3.3.4		•
degree of formality	3.4.2		•

Table I. Dimensions that characterize or influence the mechanism of change.

2.1. Dimensions as Characterizing Factors

In determining the dimensions that characterize the change mechanism, two simple heuristics were adhered to. The first was to review the literature to determine if the dimension had frequently been used to position software evolution tools. So, for example, in the literature, software tools have often been introduced as run-time or load-time [31, 33, 46] (with compile-time being implicitly accepted otherwise). Hence, 'time of change' was considered a dimension that characterized the mechanism of change.

The second heuristic, was to put the dimension in a simple sentence of the form: "The change mechanism is <*dimension*>". If such a sentence makes sense, then the dimension must reflect the essence of the change mechanism and is thus classified as a characterizing mechanism. For example, because we can say that "The change mechanism is compile-time/load-time/run-time", the 'time of change' dimension is a characterizing factor. In a similar way the dimensions of 'change history', 'activeness', 'degree of automation', and 'type of change' are classified as characterizing factors. These dimensions are discussed more fully in Section 3.

2.2. Dimensions as Influencing Factors

In determining if the dimension is an influencing factor we followed a third heuristic. For each dimension, the group tried to find an example of a way in which it could influence the change mechanism.

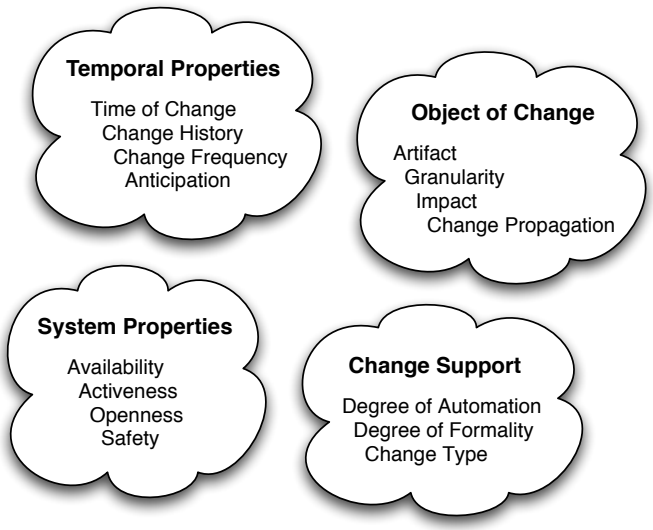

Figure 1. Themes and dimensions of software change.

For example, a system's 'availability' could typically affect the change mechanism applied to that system. If a system is required to be highly available, then this would suggest a run-time change mechanism. Low availability would allow run-time or compile-time changes.

Note that being a characterizing factor and being an influencing factor are not mutually exclusive. For example, the dimension 'time of change' described in Section 3.1.1, is employed as a characterizing mechanism in the literature, but also influences the change mechanism. If the time of change is 'run-time' additional activities like state management must be handled by the mechanism.

3. Detailed Description of the Proposed Taxonomy

Given the group's focus on the *when*, *where*, *what*, and *how* aspects of software changes, the paper will discuss each of the dimensions under the following logical themes : temporal properties (*when* is the change made), object of change (*where* is a change made), system properties (*what* is being changed), and change support (*how* is the change accomplished).

Figure 1 illustrates these themes and the dimensions that each contains. However, it should be noted that this represents only one of an infinite number of ways that software change mechanisms can be grouped. The themes and their dimensions are discussed in detail in the following subsections.

3.1. Temporal Properties (*when*)

The first logical theme in the taxonomy addresses the *when* question. Dimensions discussed in this grouping describe properties such as when evolution changes are performed and the frequency with which they occur.

3.1.1. Time of Change

Depending on the programming language, or the development environment being used, it is possible to distinguish between different phases of the software life-cycle, such as compile-time, load time and run-time. These phases have been indirectly used as a basis for categorizing software evolution tools in the literature [31, 33]. Using these phases, at least three categories become apparent, based on *when* the change specified is incorporated into the software system. Specifically these are:

- *Static.* The software change concerns the source code of the system. Consequently, the software needs to be recompiled for the changes to become available.
- *Load-time.* The software change occurs while software elements are loaded into an executable system.
- *Dynamic.* The software change occurs during execution of the software.

The traditional approach to software maintenance, where the programmer edits or extends the source code of a software system, and re-compiles (possible incrementally) the changes into a new executable system, is *static evolution*. Instead of static evolution, one often uses the term *compile-time* evolution. However this is slightly misleading since only the binary code evolves at compile-time, whereas other artifacts (source code, designs, etc.) evolve earlier.

In contrast, *dynamic* evolution considers the case where the changes are *made* or *activated* at run-time. Systems evolve dynamically, either by hot-swapping existing components or by integrating newly developed ones without the need for stopping the system. In most cases, the new or modified binary code has been evolved before run-time. Its integration into the running system is called *runtime activation*. In contrast, genuine *run-time evolution* directly modifies the executable image. This is typical in reflective software systems that can reason about, and modify, their own internal state and behaviour [38].

Please note that static and dynamic evolution neither exclude nor depend on each other. Statically modified binaries can be the basis of run-time activation. So static evolution is often an enabler for dynamic evolution. However, it is not required for dynamic evolution, since a running system can directly modify its own image.

Dynamic evolution has to be either planned ahead explicitly in the system or else the underlying platform has to provide means to effectuate software changes dynamically. Dynamic evolution is often referred to as *run-time evolution*.

Load-time evolution sits between these two extremes. It refers to changes that are incorporated as software elements become loaded into an executable system. In general, load-time evolution does not require access to the source code, but instead applies changes directly to the binaries of a system. Load-time evolution is especially well-suited for adapting statically

compiled components on demand, so that they fit into a particular deployment context. The most prominent example for a load-time evolution mechanism is Java's *ClassLoader* architecture [36]. It enables class file modifications at load-time [10, 31, 32].

Obviously, the time of change heavily influences the kind of change mechanism needed. For example, systems that allow dynamic evolution must ensure, at run-time, that the system's integrity is preserved and that there is an appropriate level of control over the change [46]. Otherwise, when the changes are implemented, the running system will crash or behave erroneously.

3.1.2. Change History

The *change history* of a software system refers to the history of all (sequential or parallel) changes that have been made to the software. Tools that make this change history explicitly available are called version control tools, and are used for a wide variety of purposes. For example, versioning mechanisms have been extensively used for schema evolution in object-oriented databases in order to support forward and backward compatibility of applications with schemas and existing objects [42, 47, 50].

In completely unversioned systems, changes are applied destructively so that new versions of a component overwrite old ones. In this scenario, old versions get lost in the evolution process. In systems that support *static versioning*, new and old versions can physically coexist at compile-time but they are identified at load- and run-time and therefore cannot be used simultaneously in the same context. In contrast to this, *dynamically versioned* systems allow two different versions of one component being deployed simultaneously side by side, within the same name space[†]. This is particularly relevant for the dynamic evolution of systems. Continued use of old components might be required for business reasons in a transition period in which the old service must be provided in parallel to the new one. It might also be temporarily required for technical reasons. Safe updates of existing components often require that new clients of the component use the new version whereas existing clients of the old component continue to use the old one. In such a context, two versions of a component coexist until the old version reaches a quiescent state [33], which allows the safe removal of the old version.

The previous discussion refers to the ability to create and to deploy different versions of a component. We will now classify the different kinds of versioning, and the mechanisms needed to support them.

Software changes may be carried out *sequentially* or *in parallel* (see Figure 2). With *sequential* software evolution, multiple persons cannot make changes to the same data at the same time. To enforce this constraint, we need some form of concurrency control (e.g., a locking or transaction mechanism) to avoid simultaneous access. With *parallel* evolution, multiple persons *can* make changes to the same data at the same time. For example, if parallel

[†]The class loading architecture of Java, for instance, does not qualify as a dynamic versioning mechanism in the above sense because different versions of a class can only coexist as part of different name spaces.

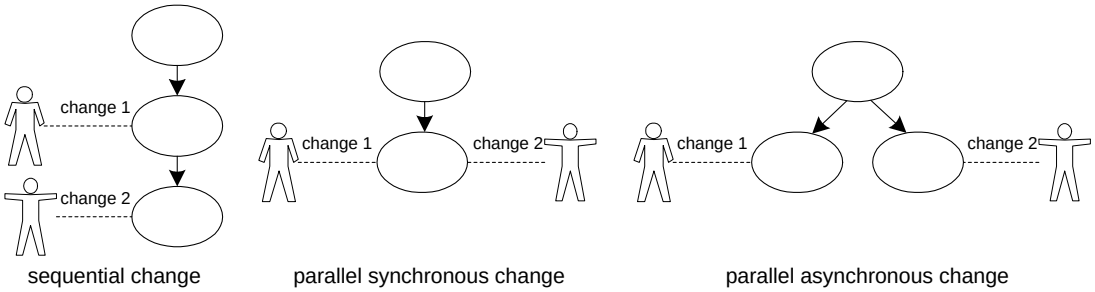

Figure 2. Sequential versus parallel evolution.

evolution exists then different software developers can simultaneously make changes to the same software component.

Within parallel evolution, a distinction must be made between *synchronous changes* and *asynchronous changes* (see Figure 2). In the *synchronous* case, the same data is shared by all persons. This is often the case in computer-supported collaborative work. It requires mechanisms such as a shared work surface and a permanent network connection to the server where the data resides, etc.

In the *asynchronous* case, all persons that change the data in parallel work on a different copy. Because the data is not shared anymore, we can have *convergent changes* or *divergent changes* (see Figure 3). With *convergent* changes, parallel versions can be merged or integrated together into a new combined version [40]. For *divergent* changes, different versions of the system co-exist indefinitely as part of the maintenance process. This is, for example, the case when invasive customisations are made to an open-source framework by different developers. As a result of these destructive changes directly applied to the framework itself, the different customisations may become incompatible, and need to evolve independently of one another.

Please note that an asynchronous convergent change process subsumes a completely synchronous process, if one only looks at the final product and not at the creation history, where in the asynchronous case different versions of some data may exist temporarily.

3.1.3. Change Frequency

Another important temporal property that influences the change support mechanisms is the *frequency of change*. Changes to a system may be performed *continuously*, *periodically*, or at *arbitrary* intervals. For example, in traditional management-information systems, users frequently request changes but these changes may only be incorporated into the system periodically, during scheduled downtimes. Other systems (for example interpreted systems), allied with less-formal change processes, may allow developers to incorporate changes continuously, as they are required.

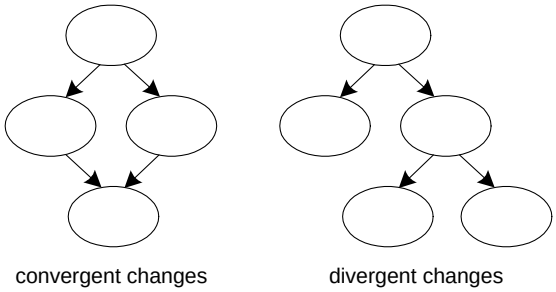

Figure 3. Convergent versus divergent parallel changes.

The frequency of change is important, because it influences the change mechanisms used. For example, if a system is being changed frequently, it suggests fine-grained changes are being performed and this, in turn, suggests an increased need for a fine-grained version control over the system. Otherwise, it would become very cumbersome to roll-back the system to specific earlier versions, when required.

3.1.4. Anticipation

This dimension refers to the time when the requirements for a software change are foreseen. Some software changes can be foreseen during the initial development of the system and, as such, can be accommodated in the design decisions taken. In contrast, *unanticipated* software changes are those that are not foreseen during the development phase. These may arise based on end-users exposure to the new system, changing business contexts or changing technical requirements. The current level of support for such changes is far from ideal even though they "account for most of the technical complications and related costs of evolving software" [30].

Obviously, this dimension is a strong influencing factor on the change mechanisms used. For example, anticipated change (that is accommodated during system design) typically involves much less effort to implement than the unanticipated changes [30].

3.2. Object of Change (*where*)

The second logical theme in our taxonomy addresses the *where* question. Dimensions discussed in this theme describe the location(s) in the system where changes are made, and the supporting mechanisms needed for this.

3.2.1. Artifact

Many kinds of software artifacts can be subject to change during static evolution. These can range from requirements through architecture and design, to source code, documentation and test suites. It can also be a combination of several or all of the above. Obviously, the different

types of software artifacts being changed influence the kind of change support mechanisms that will be required.

In dynamic evolution, as defined in Section 3.1.1, current research has focused on changes to the executing system itself. While this will probably continue to be the main research focus, there is the possibility that other artifacts of a system could be modified at run-time. For example, software could continually analyse, and dynamically extract, (design) patterns from the running source code. When this code is changed dynamically, the monitoring software could report these patterns back to the documentation. Hence, this is an example of dynamic evolution where the documentation artefact is changed.

3.2.2. Granularity

Another influencing factor on the mechanisms of software change is the *granularity* of the change. Granularity refers to the scale of the artifacts to be changed and can range from very coarse, through medium, to a very fine degree of granularity. For example, in object-oriented systems, coarse granularity might refer to changes at a system, subsystem or package level, medium granularity might refer to changes at class or object level and fine granularity might refer to changes at variable, method, or statement level. Traditionally,many researchers have distinguished only between coarse grained and fine grained artifacts with the boundary specified as being at file level. Anything smaller than a file was considered a fine-grained artifact.

3.2.3. Impact

The impact of a change can range from very *local* to *system-wide* changes. For instance, renaming a parameter in a procedure definition would only be a local change (restricted to that procedure definition), while renaming a global variable would have, in the worst case, an impact on the whole source code. Sometimes, even seemingly local changes in the software may have a global impact. For instance, deleting a procedure parameter might invalidate all call sites of the procedure.

The impact of a change can span different layers of abstraction, if we are dealing with artifacts of different kinds (see Section 3.2.1). For example, a source code change may require changes to the documentation, the design, the software architecture, and the requirements specification [48].

3.2.4. Change Propagation

Changes with non-local impact require follow up changes in other encapsulated entities (procedures, modules, classes, packages, etc). The process of performing such follow up changes is called *change propagation*. Current refactoring tools are a typical example of tools that perform automated change propagation.

Mechanisms and tools that help with change propagation often need to perform change impact analysis, traceability analysis or effort estimation. *Change impact analysis* [6] aims to assess or measure the exact impact of a change. *Traceability analysis* can help with change

Dimension	Value	Static Evolution	Dynamic Evolution
Availability	Always Available		•
	Not Always Available	•	•
Activeness	Reactive		•
	Proactive		•
Openness	Open	•	•
	Closed	•	

Table II. Dimension values that proscribe static or dynamic evolution exclusively.

impact analysis, since it establishes explicit relationships between two or more products of the software development process. Like impact, the traceability relationship can remain within the same level of abstraction (vertical traceability) or across different levels of abstraction (horizontal traceability). In many cases, changes with a high impact also require a significant *effort* to make the changes. This effort can be estimated using *effort estimation* techniques [49]. In some situations the effort can be reduced by automated tools. For example, renaming entities on the source code level is typically a global change with a high change impact, but the corresponding change effort is low because renaming can proceed in an automated way.

3.3. System Properties (*what*)

This logical theme concerns the software system that is undergoing change, and its attributes. It should be noted that the values of several of the dimensions in this grouping proscribe either static or dynamic evolution exclusively. This is illustrated in Table II.

3.3.1. Availability

Most software systems evolve continuously during their lifetime[‡]. *Availability* indicates whether the software system has to be permanently available or not. For most software systems, it is acceptable that the system is stopped occasionally to make changes (e.g., to enhance the functionality) by modifying or extending the source code. Alternatively, some software systems, for instance telephone switches, have to be permanently available. Therefore they cannot be stopped to incorporate changes. Such systems require dynamic evolution mechanisms such as dynamic loading of component updates and extensions into the running system.

[‡]Lehman [34] refers to these systems as *E-type systems*.

3.3.2. Activeness

The software system can be *reactive* (changes are driven externally) or, *proactive* (the system autonomously drives changes to itself). Typically, for a system to be proactive, it must contain some monitors that record external and internal state. It must also contain some logic that allows self-change based on the information received from those monitors [45]. A system is reactive if changes must be driven by an external agent, typically using some sort of user interface. In this way, the system can respond to external events initiated by the user.

This dimension applies only to dynamic evolution as, for a system to be proactive, the system must be running and thus the time of change is run-time. If instead the system is reactive, this implies a level of 'activity' in the system and thus, also suggests a run-time change.

The X-Adapt prototype system developed by the ACI group at the University of Limerick [39] allows for both proactive and reactive changes. It provides a GUI for system analysts to drive dynamic system reconfigurations. Additionally, it contains a number of simple monitors that assess characteristics of the system's operating environment. Using the data from these monitors, the X-Adapt system can reconfigure itself.

3.3.3. Openness

Software systems are *open* if they are specifically built to allow for extensions. Open systems usually come with a framework that is supposed to facilitate the inclusion of extensions. While they support unanticipated future extensions (statically or dynamically), it is difficult to come up with a framework that allows for flexible extensions without being too restrictive concerning all possible evolution scenarios. In general, a system cannot be open to every possible change.

Closed systems on the other hand do not provide a framework for possible extensions. Such systems are self contained, having their complete functionality fixed at build time. This does not imply that closed systems are not extensible, simply that they were not specifically designed for it. So it is possible to evolve closed systems but only statically, and usually with more effort.

Operating systems are probably the most prominent open systems. For these systems, the ability to create and run user programs that extend the functionality of the underlying operating system is essential. A second example of open systems are extensible programming languages. Extensibility in languages is either supported with explicit reflective capabilities (e.g., Smalltalk, Lisp) or with static meta-programming (e.g., OpenJava [58]).

Similarly, some database systems e.g., KIDS [23], Navajo [5] and SADES [51] support incorporation of new functionality or customization of existing functionality by using component-based and aspect-oriented techniques.

An example of a partially open system is a system that allows for plug-ins at runtime. While the plug-in modules may be unknown in advance, the ability to add them to the system at runtime is explicitly provided. A plug-in based system is not fully open since it exposes limited capacity for "extensions". An open system would allow you to do subtractions and modifications too in a clearly defined framework.

3.3.4. Safety

Note that there are many different notions of safety. One of them is *security*, for example to protect the software from viruses (in the case of dynamic evolution), or to prevent unauthorized access to certain parts of the software or to certain resources. A good example for such a mechanism is Java's concept of *security managers*, mainly exploited in web browsers for restricting access of dynamically loaded applets to the local machine.

Another is *behavioral safety*, in the sense that no crashes, unpredictable or meaningless behavior will arise at runtime due to undetected errors. Yet another notion is *backward compatibility* which guarantees that former versions of a software component can safely be replaced by newer versions without the need for global coherence checks during or after load-time. Directly related to this is the well-known *fragile base class* problem in class-based object-oriented programming, where independently developed subclasses of a given base class can be broken whenever the base class evolves.

- The *structural* variant of this problem is dealt with in IBM's SOM approach [13], by allowing (in some cases) a base class interface to be modified without needing to recompile clients and derived classes dependent on that class. This is clearly a *static* form of safety.
- The *semantic* variant of the problem is more complex and requires a dynamic approach, because the implementation of the base class can be changed as well. This gives rise to the question how a superclass can be safely replaced by a new version while remaining behaviorally compatible with all of its subclasses. This research question has been addressed in a number of research papers, including [57].

In the context of continuous evolution, *safety* becomes an essential system property. We distinguish between *static* and *dynamic safety*. The system features static safety if we are able to ensure, at compile-time, that specific safety aspects are preserved. The system provides dynamic safety if there are built-in provisions for preventing or restricting undesired behavior at runtime. To statically verify safety during dynamic evolution is a particularly hard problem. A pioneering approach in this domain is the work of Plasil and Adamek reported in this special issue [1].

Obviously, the kind and degree of safety that is required has a direct influence on the change support mechanisms that need to be provided. For example, a certain degree of static safety can be achieved by a programming language's type system at compile-time, while dynamic type tests can be used for those cases that are not covered by the static type system.

Moreover, systems that support dynamic loading need additional coherence checks to ensure that new components "fit" the rest of the system. Such checks are even necessary for systems that guarantee certain aspects of safety statically because of components' separate compilation. As a final example, systems where two versions of a single component can coexist together during a transition phase [15] not only need dynamic checks to ensure consistency: They also need some form of monitoring which is capable of mediating between the two versions actively. Object database systems, for example, provide mechanisms for adapting instances across historical schema changes, e.g., [56, 42, 52, 18].

3.4. Change Support (*how*)

During a software change, various support mechanisms can be provided. These mechanisms help us to analyze, manage, control, implement or measure software changes. The proposed mechanisms can be very diverse: automated solutions, informal techniques, formal representations and many more.

This logical theme describes some orthogonal dimensions that influence the change support mechanisms or that can be used to classify these mechanisms.

3.4.1. Degree of Automation

We propose to distinguish between *automated*, *partially automated*, and *manual* change support. In the domain of software re-engineering, numerous attempts have been made to automate, or partially automate, software maintenance tasks [54, 12, 24, 59, 44]. Typically, these are semantics-preserving transformations of the software system. In reality, however, these automated evolutions incorporate some form of manual verification and thus, can only be considered partially automated.

Within the specific domain of refactoring (i.e., restructuring of object-oriented source code), tool support also ranges from entirely manual to fully automated. Tools such as the *Refactoring Browser* support a partially automatic approach [53] while other researchers have demonstrated the feasibility of fully automated tools [8].

3.4.2. Degree of Formality

A change support mechanism can either be implemented in an ad-hoc way, or based on some underlying mathematical formalism. For example, the formalism of graph rewriting has been used to deal with change propagation [48] and refactoring [41]. In the context of re-engineering, [59] is an example of a fully automated restructuring approach that is based on graph theory. While formalisms are often used in automating software maintenance, the degree of formality is orthogonal to the degree of automation in an approach.

potential errors can be reduced significantly.

the degree of automation of a change process can range from fully manual to automatic. Second, we can have a formal change process that relies on an underlying mathematical formalism by resorting to *formal methods* [37]. Their mathematical basis makes it possible to define and prove notions like consistency, completeness and correctness.

3.4.3. Change Type

The characteristics of the change itself can influence the manner in which that change is performed. Because an extensive typology of software changes was already presented in [9], we will restrict ourselves here to the distinction between *structural* and *semantic* changes only. This distinction is an important influencing factor on the change support mechanisms that can be defined and used.

Structural changes are changes that alter the structure of the software. In many cases, these changes will alter the software behavior as well.

A distinction can be made between *addition* (adding new elements to the software), *subtraction* (removing elements from the software), and *alteration* (modifying an existing element in the software, e.g., renaming). Intuitively, it seems likely that addition is better suited to late, runtime preparation than subtraction and alteration. Subtraction and alteration suggest that changes will occur within the existing system, whereas addition suggests that extra functionality can be hooked onto the existing system.

Next to structural changes, a distinction should be made between *semantics-modifying* and *semantics-preserving* changes. In object-oriented systems, for example, relevant semantic aspects are the type hierarchy, scoping, visibility, accessibility, and overriding relationships, to name a few. In this context, semantics-preserving changes correspond to the well-known concept of software refactoring [53, 20]. In the wider context of re-engineering, semantics-preserving changes are accommodated by restructuring activities [11], such as the replacement of a `for` loop by a `while` loop, or the removal of `goto` statements in spaghetti code [59].

Note that a change may only be semantics-preserving with respect to a particular aspect of the software semantics, while it is semantics-modifying when taking other aspects of the semantics into account. For example, a typical refactoring operation will preserve the overall input-output semantics of the software, but may modify the efficiency or memory usage, which are other aspects of the software semantics that may be equally important.

The type of change is clearly orthogonal to the previous dimensions. First, some semantics-preserving changes can be fully automated [59, 43], while semantics-modifying changes typically require a lot of manual intervention. Second, semantics-preserving changes can be supported by a formal foundation [41] or not. Third, semantics-preserving changes can be a crucial part in the change process, as in the case of extreme programming [3].

4. Applying the Taxonomy

In this section, we apply the taxonomy to position some concrete tools within the software evolution domain. The taxonomy can also be used to compare formalisms or processes for software evolution in a similar manner, but this is outside the scope of the current paper.

The taxonomy will be applied to the following three tools: the *Refactoring Browser* [53], *CVS* [14, 19] and *eLiza* self-managing servers [26]. These tools have been selected because of their very different nature. As such, this should also be reflected in their comparison based on the taxonomy (see Table III). Next, this comparison will be used to identify to which extent the tools complement each other.

Another typical use of the taxonomy is to compare tools that share the same or a similar purpose. This allows us to identify the differences, strengths and weaknesses of each tool. Such a comparison has been carried out on four different refactorings tools in [55], using an earlier version of our taxonomy.

4.1. Refactoring Browser

The *Refactoring Browser* is an advanced browser for the *Smalltalk* IDEs *VisualWorks*, *VisualWorks/ENVY*, and *IBM Smalltalk*. It includes all the features of the standard browsers plus several enhancements. One notable enhancement is its ability to perform several behavior-preserving, refactoring transformations.

4.1.1. Temporal Properties

Time of change. The refactoring transformations and any other changes that the programmer wishes to make are prepared in the source code. It is then compiled into a executable system. As such the tool provides compile-time change support.

Change history. Although the Refactoring Browser has a basic undo mechanism, it does not provide facilities for managing different versions. It can be used in an unversioned environment (e.g., VisualWorks) or a versioned environment (e.g., VisualWorks/ENVY).

Change frequency. Refactorings can be applied at *arbitrary* moments in time. Typically, small refactorings (e.g. renaming of a local variable or method inlining) are applied more frequently than big refactorings.

4.1.2. Object of Change

Artifact. The Refactoring Browser applies changes directly (and only) to *source code* entities.

Granularity. The granularity of the changes that can be done with the Refactoring Browser depend on which refactoring is being applied. Typically, refactorings involve a *limited number of classes and methods*. Some of the supported refactorings have a finer level of granularity. For example, the *Inline Method* refactoring occurs at method level.

Impact. A refactoring typically has a low change impact, in the sense that the number of global changes to the source code is limited. However some of the refactorings can perform global changes.

Change propagation. The Refactoring Browser will make all the necessary changes in the entire source code. However, the tool provides no support for propagating changes to other layers, such as design models or requirements.

4.1.3. System Properties

Availability. During refactoring, the software system being refactored is only partially available for execution, since it is incrementally recompiled.

Activeness. The Refactoring Browser is used in a *reactive* way to refactor an existing software system. Changes to the system are triggered by the user of the Refactoring Browser who has the responsibility of deciding *which* refactoring to apply, *when* to apply it, and *where* to apply it to.

Openness. The Refactoring Browser can help to build open systems since it provides support for refactorings. Refactorings restructure object-oriented code to make it more evolvable, to accomodate future changes, to introduce design patterns or to turn an application into an application framework. Each of these help to increase or maintain the openness of a software system.

Safety. Refactorings guarantee a certain degree of behavioral safety, since the change is behavior-preserving with respect to the original behavior (although there is no formal proof of this).

4.1.4. Change Support

Degree of automation. The Refactoring Browser can be considered a *semi-automated* tool. Indeed, the refactorings themselves can be applied automatically, but it is the responsibility of the user to decide where and when a certain refactoring should be applied.

Degree of formality. Although formalisms for refactorings exist [41], the Refactoring Browser is not based on such an underlying mathematical model.

partial support for the extreme software development process [3], in which refactoring is advocated as an important change activity.

Change type. The changes performed by a refactoring are by definition *semantics-preserving*, since they are behavior-preserving, and hence only make changes to the structure of the source code. The changes fall under the category of *alterations* to existing code with associated additions and subtractions. For example the 'extract method' refactoring adds a new method and modifies the code of an existing one.

4.1.5. Discussion

Using the taxonomy, the Refactoring Browser can be positioned as a compile-time, semi-automatic change mechanism that supports semantics-preserving changes.

By applying the taxonomy to the Refactoring Browser, we were able to identify some of its weaknesses[§]. To overcome these current limitations, the tool could be complemented with a variety of other tools:

[§]Its strengths are well-known so we will not discuss those here.

- To overcome its *reactiveness* and *semi-automated nature*, the Refactoring Browser could be complemented with a tool that detects where and when refactorings should be applied [27].
- To improve its *change propagation* support, the Refactoring Browser could be complemented with a tool that ensures that source code refactorings are propagated to the design documents so that these can be kept consistent.
- To cope with the lack of formality of the Refactoring Browser, and to improve the *safety* of the software systems it acts upon, one might complement the tool with formal approaches that check whether a certain refactoring preserves certain aspects of the software behavior [41].
- To raise the level of granularity of the refactorings provided by the tool, there is a need for incorporating composite refactorings.
- To enable roll-backs to previous versions of the software, the refactoring tool should be complemented by a version control tool.

4.2. Concurrent Versions System

CVS [14] is the Concurrent Versions System, the dominant open-source network-transparent version control system. CVS is useful for everyone from individual developers to large, distributed teams.

4.2.1. Temporal Properties

Time of change. CVS only supports compile-time evolution. It typically (but not exclusively) stores source code files, which have to be compiled before they can be executed.

Change history. The main objective of CVS is to maintain a history of all source-level changes. CVS supports sequential as well as parallel changes. The changes can be divergent as well as convergent, since parallel changes can be merged. Commits are always performed in sequence.

Change frequency. The frequency of changes is *arbitrary*, since changes are triggered by the user.

4.2.2. Object of Change

Artifact. CVS is file-based, so the artifacts that are versioned are basically files. However, the contents of these files can be virtually anything, so CVS is applicable to any kind of software artifact that can be stored in a file.

Granularity. Changes are carried out on the level of files. If a change requires simultaneous modifications of several files, it is difficult to express this explicitly via CVS. Similarly, several changes in one file have to be split up in several stages to express multiple independent changes.

Impact. Since CVS can be used to version any software artifact, the impact of changes can be arbitrary. For instance, changes in libraries that are under the control of CVS can have an impact on clients even outside of the local computer system.

Change propagation. Since CVS cannot reason about the contents of the files that are stored, it has no support for change propagation.

4.2.3. System Properties

Availability. The CVS is not very suited to evolve software systems that need to be continuously available.

Activeness. CVS is a tool applied to other systems. In general, it is used in a reactive way, i.e., triggered by the software developer rather than by the evolving system itself.

Openness. CVS does not provide any support to make a software system more open to future changes.

Safety. CVS provides support for network security and access control. As such, it contributes to a more robust and reliable software development and versioning process. On the other hand, tools as general as CVS have no knowledge about the semantics of the administered system, therefore they cannot provide any means to make the software evolution process more safe.

4.2.4. Change Support

Degree of automation. CVS is not automatic in that the user has to be aware of the presence of the versioning system. Whenever a change is being made, this change has to be committed explicitly to the version repository.

Degree of formality. CVS has no underlying mathematical foundation.

Change type. CVS puts no constraints on the types of change that can be made to the software system. It can be a semantics-preserving or semantics-changing change. It can be an addition, subtraction, or alteration.

4.2.5. Discussion

Using the taxonomy, the CVS system can be positioned as a compile-time, manual change mechanism that supports semantics-preserving and semantic changing evolutions. The file-based approach is one of the main strengths of CVS, but at the same time it is also the most important weakness. From the positive side, the file-based approach makes CVS general purpose, since it can be used in a platform-independent way, and any kind of software artifact can be versioned as long as it is stored in file format. From the negative side, the granularity of changes is always restricted to files, which makes it very difficult to deal with relationships at a

higher or lower level of granularity. As a result, CVS has poor support for change propagation or behavioral safety, because this requires information about the software that is not directly accessible in the CVS file format.

4.3. eLiza Self-Managing Servers

The eLiza project was set up by IBM to provide systems that would adapt to changes in their operational environment. eLiza technology has been incorporated into the MVS Mainframe Operating System since 1994 and works there to reallocate resources and to control the CPU configuration dynamically. In a distributed context, eLiza has been incorporated into IBM's Heterogeneous Workload Management software.

4.3.1. Temporal Properties

Time of change. In servers with eLiza technology, monitors take snapshots of the systems performance at runtime. Using this information the system may decide to adapt. Thus the tool provides run-time change support.

Change history. Versioning of the changes is sequential. That is, only one configuration of the system can be active at any given time. However, over time, the eLiza based servers can return to previously used configurations based on further changes in their operational environment.

Change frequency. eLiza technology allows systems to change whenever the monitors provide data that suggests a better CPU configuration or a better strategy for allocation of resources.

4.3.2. Object of change

Artifact. The artifact changed by eLiza technology is the executable code of the system.

Granularity. The changes are made to the system configuration. Hence they can be considered coarse grained.

Impact. The adaptations range from local, to slightly more global. Moreover, the changes only affect the running system.

Change propagation. There is no support for propagating changes in the running system to other artifacts such as the source code, documentation, design, and so on.

4.3.3. System Properties

Availability. eLiza technology is incorporated into highly available operating systems, allowing them to adapt without down-time.

Activeness. This system is proactive in that it relies on its own monitors to assess the health of the processes running on its servers. Based on this information, it actively re-configures itself without help from external agents.

Openness. The systems that are being changed by eLiza technology are open, because they can be modified at runtime to incorporate changes. However, this is a limited kind of openness because the adaptation logic and monitors are incorporated into the system at build time.

Safety. As the adaptations are built into the system at design time, it is anticipated that their safety level is high. Indeed, eLiza technology has been incorporated into several servers and these servers have been in operation for many years. This provides empirical evidence for the safety level of the adaptations.

4.3.4. Change Support

Degree of automation. The system changes are fully automated. They are specified at build time by the system developers and performed by the system as it executes.

Degree of formality. eLiza technology uses learning algorithms with a mathematical basis. These learning algorithms determine when the pattern of input from the monitors suggests that a change in the system configuration is beneficial.

Change type. The changes supported by eLiza are semantics-modifying in that the behavior of the system is changed by evolution. From a structural point of view, the systems are changed by adding new elements to the system, replacing elements with others, or by removing existing system elements.

4.3.5. Discussion

Again, the proposed taxonomy can be used to evaluate and position this technology. In terms of positioning the mechanism, the taxonomy states it is an automatic, run-time, change mechanism that affects the semantics of the changed system by means of addition, replacement and subtraction.

In terms of evaluating the system, the taxonomy focuses attention on its degree of formality, its openness and its change propagation dimensions. While the technology relies on the underlying formalism of learning algorithms, these algorithms have traditionally been associated with a lack of accountability. That is, it is difficult to determine why neural networks (for example) provide a specific output for a given input. This lack of accountability, means that an underlying rationale for the configuration changes is missing.

In terms of openness, all modifications to the system are built into the system at design time. So, while it does allow for run-time adaptation, eLiza technology makes little allowance for non-anticipated changes. For example, new monitors cannot be added at run-time. However, it must be acknowledged that the learning algorithm can determine new adaptation 'logic' in

Dimension	RefactoringBrowser	CVS	eLiza
time of change	compile-time	compile-time	run-time
change history	no	any	sequential
frequency	arbitrary	arbitrary	continuously
artifact	source code	file	executable code
granularity	several classes and methods	file	system configuration
impact	mostly local changes to source code	global changes to any artifact	mostly local changes to executable code
activeness	reactive	reactive	proactive
openness	by refactoring	no	limited (built-in)
availability	incremental compilation	no	no down-time
safety	behavior-preserving	network security, access control	high safety (empirical evidence)
automation	semi-automatic	manual	fully automated
formality	no	no	yes
change type	semantics-preserving	any	semantic-changing

Table III. Comparison of software evolution tools based on proposed taxonomy.

response to the existing monitored information. No change propagation is supported by the technology, as it only works at the code level.

4.4. Tool comparison

Table III illustrates how the taxonomy can be used to distinguish between different change mechanisms. That is, the three tools presented differ markedly in their characterizing factors. eLiza technology acts at run-time whereas the other two act at compile-time. The three range from semantics-preserving to semantics changing. And finally, the mechanism range from automatic, through semi-automatic, to fully manual.

From the table, we can also conclude that the Refactoring Browser and CVS are more similar to each other than to the eLiza technology. Both of these are reactive compile-time mechanisms which could be complementary, since they both work at the same level (source code) but emphasize different aspects (refactoring and version control). Unfortunately, it is not straightforward to integrate both tools, because the Smalltalk development environment does not make use of a file system to store its software artifacts. Instead, the entire software application is stored together with the environment itself as a single *Smalltalk image*. A similar refactoring tool for Java would be much easier to integrate with CVS, because of the fact that each Java class is stored as a separate file.¶

¶The Eclipse development environment for Java proves this point, since it features a refactoring tool similar to the Refactoring Browser, as well as built-in support for CVS.

5. Related Work

As mentioned in the introduction, Lientz and Swanson[35] produced a mutually exclusive typology of software maintenance. Chapin *et al.* [9] built on this typology to derive an evidence-based classification of maintenance activities. However, both these works categorized evolution activities based on their purpose. Other categorizations of software maintenance, as reviewed by Kemerer and Slaughter [28], have focused on the impact that maintenance has had on the software system [4], the trends that become apparent in evolution activity over time [25] and the relationship between maintenance types and maintenance activities [2].

More closely related to our work is the work of Kitchenham et al. [29]. This work attempted to identify the factors that influence maintenance activity. However, their work was directed at characterizing maintenance contexts that may exist in empirical studies, thus allowing researchers in that field compare their work, and explain differing results. Consequently, the group use rich descriptive dimensions that were not always relevant to the change mechanism. For example, in their grouping; 'Process Organisation', they had dimensions like 'group organization', which would seem to only have a limited impact on the change mechanism adopted.

In addition, the rich dimensions they used, sometimes seem difficult to categorize. For example, in their 'peopleware' grouping, dimensions such as skill and attitude would be hard to quantify for all the shareholders in a maintenance process. This is not a weakness in their work. Rather, theirs is an ontology directed at comparison across empirical studies where a multitude of varying independent variables have to be accounted for. Our taxonomy, in contrast, focuses more on one aspect of evolution: the change mechanism.

6. Conclusion

In this paper we proposed a taxonomy of software evolution based on a large number of dimensions characterizing the mechanisms of change and the factors that influence these mechanisms. Our discussion of these dimensions subdivides them into four logical themes: temporal properties (*when*), object of change (*where*), system properties (*what*) and change support (*how*).

We demonstrated the use of the taxonomy by applying it to three different software evolution tools: the Refactoring Browser, the Concurrent Version System, and eLiza self-managing servers. This allowed us to: (1) position each of the analyzed tools within the taxonomy; (2) identify the strengths and weaknesses of each tool; (3) suggest how each of the tools could be improved by or complemented with other tools; (4) compare the properties of the three analyzed tools.

In the future we intend to use and extend the taxonomy to be able to compare change support formalisms and processes as well. The method we used to obtain our taxonomy, based on a number of simple heuristics (see Section 2), might also be useful to develop taxonomies in other domains.

ACKNOWLEDGEMENTS

This work was carried out as part of the FWO research network on *Foundations of Software Evolution* [22], the ESF RELEASE network [17], the EPSRC project *AspOEv* [16] and the QAD/EI funded ACI project. The taxonomy is an elaboration and extension of the results of a working group (consisting of the authors of this paper, Salah Sadou, and Stefan Van Baelen) that discussed this topic during the *ECOOP 2002 Workshop on Unanticipated Software Evolution* [30]. We thank Kim Mens, Mehdi Jazayeri, Finbar McGurren for their discussions about this topic. We also thank the anonymous referees of the USE 2003 workshop for their "use"-ful reviews.

AUTHORS' BIOGRAPHIES

Jim Buckley obtained a honors BSc degree in Biochemistry from the University of Galway in 1989. In 1994 he was awarded an MSc degree in Computer Science from the University of Limerick and he followed this with a PhD in Computer Science from the same University in 2002. He currently works as a lecturer in the Computer Science and Information Systems Department at the University of Limerick, Ireland. His main research interest is in theories of software comprehension, software reengineering and software maintenance. In this context, he has published actively at many peer-reviewed conferences and workshops. He has also co-organized a number of international conferences and workshops in this domain. He currently coordinates 2 research projects at the University: one in the area of software visualization and the other in architecture-centric evolution.

Tom Mens obtained the degrees of Licentiate in Mathematics in 1992, Advanced Master in Computer Science in 1993 and PhD in Science in 1999 at the Vrije Universiteit Brussel, Belgium. In October 2000 he became a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO) associated to the Programming Technology Lab of the Vrije Universiteit Brussel. Since October 2003 he is a lecturer at the Université de Mons-Hainaut, Belgium. His main research interest lies in the use of formal techniques for improving support for software evolution. In this context, he published several peer-reviewed articles in international journals and conferences, he co-organised numerous international workshops, and he coordinates two European research networks on software evolution [22, 17]. He also co-promotes an interuniversity research project on software refactoring [21].

Matthias Zenger is a research assistant in the Programming Methods Laboratory of the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland. His main research interests are the design and the implementation of programming languages with a focus on software extensibility and evolution. In 1998, he obtained a diploma degree in computer science from the University of Karlsruhe, Germany.

Awais Rashid is a Lecturer in the Computing Department at Lancaster University. He holds a BSc in Electronics Engineering with Honours from the University of Engineering and Technology, Lahore, Pakistan, an MSc in Software Engineering Methods with Distinction from the University of Essex, UK and a PhD in Computer Science from Lancaster University, UK. His principal research interests are in aspect-oriented software engineering, and extensible and evolvable object and XML data management systems. He has published actively on these topics, edited special issues of IEE Proceedings Software and The Computer Journal and has been involved with the organising and program committees of a number of relevant events. Recently he also co-edited a book on XML Data Management with Akmal Chaudhri and Roberto Zicari. He is a member of the IEEE and the IEEE Computer Society.

Günter Kniesel is currently a substitute professor of Applied Computer Science at the university of Osnabrück and a lecturer at the the University of Bonn. His has worked and published on support for unanticipated software composition in object object-oriented programming languages and component technologies. This includes dynamic object models (object-based inheritance, roles), aspect-oriented programming, dynamic software evolution, program transformations, refactoring, and analysis of interferences between transformations. Other research interests include encapsulation, aliasing, reflection and knowledge representation. Dr. Kniesel co-organized a number of international conferences and workshops in this domain (WEAR 2003, AOSD-GI 2003, AOSD-GI 2002, AOSD 2002, USE 2003, USE 2002) and served on the program committees of AOSD 2003, AOSD 2004, ELISA 2003, ASWA 2002, SOFSEM 2002. He obtained a honours Diploma in Computer Science from the University of Dortmund in 1989 and a honours Ph.D. in Computer Science from the University of Bonn in 2000.

REFERENCES

1. J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. *Journal of Software Maintenance and Evolution (JSME)*, 2003. Other paper in this issue.
2. V. Basili, L. Briand, S. K. Y.-M. Condon, W. Melo, and J. Valett. Understanding and predicting the process of software maintenance releases. In *Proceedings of the 18th International Conference on Software Engineering*, 1996.
3. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
4. L. Belady and M. Lehman. A model of large program development. *IBM Systems Journal*, 15(1):225–252, 1976.
5. H. Bobzin. The architecture of a database system for mobile and embedded devices. In *Component Database Systems*, pages 237–251, 2000.
6. S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.
7. Cambridge. Dictionaries online. Webpage http://dictionary.cambridge.org, 2003. [25 July 2003].
8. E. Casais. Automatic reorganization of object-oriented hierarchies: a case study. *Object Oriented Systems*, 1:95–115, 1994.
9. N. Chapin, J. Hale, K. Khan, J. Ramil, and W.-G. Than. Types of software evolution and software maintenance. *Journal of software maintenance and evolution*, 13:3–30, 2001.
10. S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *Proc. European Conf. Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer-Verlag, 2000.
11. E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
12. A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28:117–127, 1995.
13. I. Corporation. The System Object Model (SOM) and the Component Object Model (COM): A comparison of technologies from a developers perspective. White paper, 1994.
14. CVS. Concurrent versions systems. Webpage http://www.cvshome.org/, 2003. [25 July 2003].
15. D. Duggan and Z. Wu. Adaptable objects for dynamic updating of software libraries. In *USE 2002 Workshop*, 2002.
16. EPSRC. AspOEv: An aspect-oriented evolution framework for object-oriented databases (EPSRC GR/R08612). Webpage http://www.comp.lancs.ac.uk/computing/aod/, 2003. [25 July 2003].
17. European Science Foundation. Scientific research network "REsearch Links to Explore and Advance Software Evolution (RELEASE)". Webpage http://labmol.di.fc.ul.pt/projects/release, 2003. [25 July 2003].
18. F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the O2 object database system. In *Proc. 21st Int'l Conf. Very Large Databases*, pages 170–181, Zurich, Switzerland, September 1995. Morgan Kaufmann.
19. K. Fogel and M. Bar. *Open Source Development With CVS*. Paraglyph Publishing, 2nd edition, 2002.
20. M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
21. Fund for Scientific Research – Flanders (Belgium). Research project "Foundations of Software Refactoring". Webpage http://win-www.uia.ac.be/u/lore/refactoringProject, 2003. [25 July 2003].

22. Fund for Scientific Research – Flanders (Belgium). Scientific research network "Foundations of Software Evolution". Webpage http://progwww.vub.ac.be/FFSE/network.html, 2003. [25 July 2003].
23. A. Geppert and K. Dittrich. Strategies and techniques: Reusable artifacts for the construction of database management systems. *Lecture Notes in Computer Science*, 932:297–310, June 1995.
24. J. Girard and R. Koschke. Finding components in a hierarchy of modules: A step towards architectural understanding. In *Proc. Int'l Conf. Software Maintenance*, pages 58–65. IEEE Computer Society Press, 1997.
25. Y. C. C. Hok. On analyzing maintenance process data at the global and detailed levels: A case study. In *Proceedings of the 4th International Conference on Software Maintenance*, 1988.
26. IBM. Ibm @server and autonomic computing. Webpage http://www-1.ibm.com/servers/autonomic/, 2003. [25 July 2003].
27. Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. Int'l Conf. Software Maintenance*, pages 736–743. IEEE Computer Society Press, 2001.
28. C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, 1999.
29. B. Kitchenham, G. Travassos, A. V. Mayrhauser, F. Niessink, N. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *International Journal of Software Maintenance: Research and Practice*, 11(6):365–391, 1999.
30. G. Kniesel. Unanticipated software evolution. Webpage http://joint.org/use/, 2003. [25 July 2003].
31. G. Kniesel, P. Costanza, and M. Austermann. JMangler – a framework for load-time transformation of Java class files. In *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, November 2001.
32. G. Kniesel, P. Costanza, and M. Austermann. JMangler – A Powerful Back-End for Aspect-Oriented Programming. In R. Filman, T. Elrad, and M. Aksit, editors, *Aspect-oriented Programming*. Prentice Hall, 2003.
33. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.
34. M. M. Lehman, J. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution — the nineties view. In *Proc. 4th Int'l Symp. Software Metrics*, pages 20–32, Albuquerque, New Mexico, November 1997. IEEE Computer Society Press.
35. B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.
36. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Ed)*. Java Series. Addison Wesley, 1999.
37. Luqi and J. A. Goguen. Formal methods: Promises and problems. *IEEE Software*, pages 73–85, January 1997.
38. P. Maes. *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.
39. F. McGurren and D. Conroy. X-adapt: An architecture for dynamic systems. In *Unanticipated Software Evolution Workshop*, 2002.
40. T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.
41. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.
42. S. Monk and I. Sommerville. Schema evolution in OODBs using class versioning. *ACM SIGMOD Record*, 22(3):16–22, 1993.
43. I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 235–250. ACM Press, 1996.
44. W. R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. PhD thesis, University of Texas, 1986.
45. P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, pages 54–62, May/June 1999.
46. P. Oreizy and R. Taylor. On the role of software architectures in runtime system reconfiguration. *IEE Proc. Software Engineering*, 145(5):137–145, October 1998.

47. Y.-G. Ra and E. A. Rundensteiner. A transparent schema-evolution system based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, 1997.
48. V. Rajlich. A model for change propagation based on graph rewriting. In *Proc. Int'l Conf. Software Maintenance*, pages 84–91. IEEE Computer Society Press, 1997.
49. J. F. Ramil and M. M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proc. Int'l Conf. Software Maintenance*, pages 163–172. IEEE Computer Society Press, 2000.
50. A. Rashid. A database evolution approach for object-oriented databases. In *Proc. Int'l Conf. Software Maintenance*, pages 561–564. IEEE Computer Society Press, 2001.
51. A. Rashid and P. Sawyer. Aspect-orientation and database systems: An effective customisation approach. *IEE Proceedings Software*, 148(5):156–164, 2001.
52. A. Rashid, P. Sawyer, and E. Pulvermueller. A flexible approach for instance adaptation during class versioning. In *ECOOP 2000 Symposium on Objects and Databases*, pages 101–113, 2000.
53. D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
54. R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings Int'l Conf. Software Engineering*, pages 83–92. IEEE Computer Society Press, 1991.
55. J. Simmonds and T. Mens. A comparison of software refactoring tools. Technical Report vub-prog-tr-02-15, Programming Technology Lab, November 2002.
56. A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *1st OOPSLA Conference*, pages 483–495, 1986.
57. P. Steyaert, C. Lucas, K. Mens, and T. DHondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–286, 1996.
58. M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Lecture Notes in Computer Science*, 1826:119–135, 2000.
59. G. Urshler. Automatic structuring of programs. *IBM Journal of Research and Development*, 19(2), March 1975.