Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw.: Evol. and Proc. 2013; **25**:535–568
Published online 25 April 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1556

A survey of dynamic software updating

Habib Seifzadeh[1,*,†], Hassan Abolhassani[2] and Mohsen Sadighi Moshkenani[3]

[1]*Computer Engineering Department, Science and Research Branch, Islamic Azad University, Tehran, Iran*
[2]*Computer Engineering Department, Sharif University of Technology, Tehran, Iran*
[3]*School of Science and Engineering, Sharif University of Technology, International Campus, Kish Island, Iran*

ABSTRACT

Application update at run-time remains a challenging issue in software engineering. There are many techniques with different evaluation metrics, resulting in different behaviours in the application being updated. In this paper, we provide an extensive review of research work on dynamic software updating. A framework for the evaluation of dynamic updating features is developed, and the articles are categorized and discussed based on the provided framework. Areas of online software maintenance requiring further research are also identified and highlighted. This information is deemed to not only assist practitioners in selecting appropriate dynamic updating techniques for their systems, but also to facilitate the ongoing and continuous research in the field of dynamic software updating. Copyright © 2012 John Wiley & Sons, Ltd.

Received 19 April 2011; Revised 7 March 2012; Accepted 13 March 2012

KEY WORDS: dynamic software updating; long-lived and mission-critical software applications; availability

1. INTRODUCTION

Dynamic Software Updating (DSU) systems are software applications with which running programs can be updated without interrupting service [1]. DSU is also referred to as 'on-the-fly' program modification, online version change [2], hot-swapping [3], and/or dynamic software maintenance. In component-oriented or distributed systems, it is usually called 'software reconfiguration' [4–6].

Utilizing DSU techniques promotes system uptime and availability. Today, there exist many mission-critical and long-running software applications requiring round-the-clock service. Examples include air traffic or satellite control systems, enterprise applications and communication switching systems. Disrupting such systems even for short periods during updates can have significant undesired, and in some cases even catastrophic, consequences. The more popular these applications, the more important dynamic updating systems become.

Dynamic updating systems are not limited to the above applications. If an operating system patch is published over the Internet, applying it without restarting the system might be desired by users. Other examples of applications requiring DSU over their life-time include desktop applications running on uniprocessor computers, operating systems running on an embedded Mars rover controller, and a distributed application running on multiple nodes in a network. The number of published dynamic updating system is considerably comparable to the number of systems with demand for dynamic updating. Since the first paper introducing the concept (to the best of our knowledge) [7], hundreds of papers have been published in this area. In spite of this diversity, a survey to facilitate choosing the appropriate dynamic updating system for a specific application has been little discussed in the literature. This paper aims to fill that gap.

*Correspondence to: Habib Seifzadeh, Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.
†E-mail: seifzadeh@iaun.ac.ir

Papers reviewed in this survey were selected based on several criteria including year of publication, journal or conference reputation, and scope, a wide range of which are covered here. Hence, this survey's target audiences are those interested in the state-of-the-art of dynamic updating systems and their applications in other areas of software engineering. A name has been chosen for every DSU system reviewed to simplify their nomenclature in this survey. If a system has been named by its developers, we kept their designation. Otherwise, the last name of the first author has been chosen as the system's name.

In addition to DSU systems, we referenced some ideas and categorizations from other surveys in this paper [8–12] even though some of these surveys have become obsolete protect[8, 9] while others only tangentially relate to dynamic updating without actual focus on this topic [10–12]. We therefore chose to omit these from our discussion.

Reviewing a variety of different papers in an area can result in an inconsistent presentation, unless a set of predefined criteria is defined and applied to categorize the systems described. To this end, we have identified distinctive evaluation metrics of dynamic software updating and then examined techniques each paper has taken to classify DSU systems coherently. There are also other software systems, called *autonomous* or *self-adaptive* software systems, which are adapted dynamically to promote availability and simplify maintenance. Despite this similarity, there are several basic differences between them and dynamic software updating systems, forcing us to briefly describe them here to give the reader an overall insight about DSU related systems.

Autonomous systems dynamically monitor their behaviour by an embedded *control loop* and attempt to enhance this behaviour by a knowledge base that has been created by an expert or previous executions [13–15, 11, 16, 12]. Therefore, they are dynamically updated at run-time like applications that leverage DSU, but the difference is that the dynamic adaptation is performed by the running application in the former, while this adaptation is performed by the programmer in the latter. Moreover, because a control loop should be embedded in autonomous systems, the implementation cost and code complexity is sacrificed for maintenance cost in these systems. On the other hand, dynamic updating techniques do not impose any design or implementation cost. Evidently, there are DSU systems that can dynamically update running applications created without dynamic updating concepts in mind [17–20].

According to the last difference, a good idea for creating an autonomous system with both low design and implementation and maintenance costs may be to use dynamic updating to add autonomic behaviours to normal systems dynamically [21]. Delineating autonomous systems here, we do not describe them in this survey anymore and focus only on dynamic updating systems. The remaining parts of this paper are organized as follows: in Section 2, we provide evaluation metrics that can be used to evaluate techniques developed by DSU systems to respond or implement DSU features. In addition, we propose several subclasses and affect/depend relationships among evaluation metrics. In Section 3, we evaluate and classify DSU systems based on evaluation metrics described in Section 2. Finally in Section 4, we suggest some areas of dynamic updating to be focused on in future studies.

2. EVALUATION METRICS

To accurately stratify distinct evaluation metrics of DSU systems, we categorized them along two dimensions: (1) capabilities and constraints in Section 2.1 and (2) techniques for dealing with DSU issues in Section 2.2. In particular, we will focus on subclasses within each evaluation metric. We realize that, despite their distinctiveness, certain evaluation metrics depend on (or affect) each other. For example, certain performance requirements desire certain programming languages. Therefore, we will also focus on affect/depend interdependency among evaluation metrics in Subsection 2.3.

2.1. Capabilities and constraints

2.1.1. Scope. We specified the following scopes for dynamic updating systems based on the DSU literature: (1) server applications, (2) desktop applications, (3) operating systems, (4) real-time and embedded systems, (5) distributed systems, and (6) business applications and databases.

DOI: 10.1002/smr

We based these categories on actual research done on case studies, but a specific DSU system may not be limited in its scope to that implied by these case studies. There are also research works in the literature that although they are designated for a special scope, they have not been implemented yet. Therefore, we indicate whether a system is implemented when we categorize DSU systems based on their scope.

2.1.2. Level of abstraction. The level of abstraction via which the programmer specifies updates to his/her program plays a key role in simplifying usage of a DSU system. To date, almost all of DSU systems discover the difference between new and old program versions via source code that the programmer gives to the system. It is more desirable that the programmer can specify some program design updates. We therefore categorized existing DSU research works into: (1) code level and (2) design level of abstraction.

2.1.3. Ability to update previously started code. Most dynamic updating systems require their target programs to embed dynamic updating capabilities, while some can dynamically update applications without this requirement. Importantly, this evaluation metric determines the ability to dynamically update long-lived legacy systems that have been created and executed without the dynamic updating idea in mind. On the basis of this feature, we categorized DSU systems into: (1) can update previously started code and (2) cannot update previously started code.

A related feature of some dynamic updating systems is whether or not they require older version program source code to be present to discover differences between it and newer version source code. DSU systems bound to this requirement are useful only when the programmer of old and new versions is the same. This metric categorizes DSU systems into the following disjointed subclasses: (1) requires older source code version and (2) does not require older source code version.

2.1.4. Simplicity. Even though simplicity is an important evaluation metric of software systems, measuring it is not a trivial task, if one wants to avoid subjective results. To be as objective as possible in categorizing DSU systems with respect to simplicity, we specified a number of fixed criteria. These include popularity of the DSU's target programming language, the level of abstraction, and tools (or GUIs) the system provides for automating some usual routines. We categorized a dynamic updating system into the subclass 'very simple' if it provides more than two of the above criteria. In our subclassification scheme, a 'simple' DSU system provides only one of these criteria, and a 'hard' one provides none.

2.1.5. Consistency. In this survey, we propose a separation between 'Consistency' and 'Type-Safety' as distinct DSU evaluation metrics in contradistinction to dynamic updating literature. Here, we discuss the more important former and defer the less important latter to 'Type-Safety' paragraph of Section 2.2.

A program is consistent if it 'behaves' according to user expectation. For example, if the main loop of a program invokes an encode/decode function to encrypt send-messages over a connection at the beginning of each iteration and again invokes it to decrypt receive-messages at the end of the iteration, updating of such function during the main execution can cause the program to encrypt a message with one algorithm and decrypt its result with another one. This would cause the program to perform inconsistently and contrary to online updating, this cannot take place in the case of off-line updates. With this metric taken into account, we categorize DSU systems into three classes: (1) consistent, (2) consistent with remarks, and (3) inconsistent.

2.1.6. Wait time to update and predictability. Dynamic software updating systems that apply updates only at special program points should wait until these points are reached before updating, albeit systems that apply updates immediately. Reducing wait times is desirable in such DSU systems. In addition, the wait time for a proper update point should also be known in advance in some special, deterministic systems. This evaluation metric divides dynamic updating systems into three subclasses: (1) immediate, (2) predictable, and (3) unpredictable.

2.1.7. Update duration. Update duration defines the amount of time during which the update is applied to the running program. If a DSU system applies updates in an atomic way, 'Update Duration' translates into 'Disruption Duration' from the client's perspective, which means that shorter updates equal less disruption to clients using the program.

DOI: 10.1002/smr

Even DSU systems that inject program updates in the nonatomic manner should have short update durations because long update duration causes the execution of a long extra process in parallel with the program's normal execution. Figure 1 shows wait time to update and update duration times.

2.1.8. Code clean-up. If the old modules are not removed from memory after updating, a memory footprint occurs. The condition worsens with each update of the running application if the running application is updated regularly. Therefore, dynamic updating frameworks should be constructed with code clean-up in mind especially when programs are updated regularly.

2.1.9. Performance overhead. Dynamic software updating systems usually embed extra code in applications to simplify their updates. However, the extra code decreases the program's overall performance. This performance degradation becomes more significant, if dynamic updates are received only rarely. Thus, extending the applications with such code should be avoided in programs uncommonly updated.

2.1.10. Supported changes. When features are added to or bugs are removed from applications, their source code inevitably changes, sometimes drastically. If the DSU system in use cannot support some such changes, for example changes to module interfaces of the running program, then there is lower probability that the system can succeed in applying every kind of update. However, an excessive repertoire of supported changes also makes a dynamic updating system difficult to implement. Research discussed in this survey categorizes DSU systems into two subclasses based on this evaluation metric: (1) can change modules' interface and (2) cannot change modules' interface.

2.1.11. Programming language. We decided to categorize DSU research works based on two programming language criteria: Popularity of programming language, which divides DSU systems into three subclasses: (1) using popular programming languages unchanged, (2) using unpopular programming languages, and (3) modifying a popular programming language to add dynamic updating features; and type of programming language, which divides DSU systems into three other subclasses: (1) procedural, (2) functional, and (3) object-oriented. Several online sources were investigated to find programming languages popularity [22–24].

2.1.12. Multithreading support. Nowadays, multithreading support becomes an important evaluation metric because of the popularity of multicore systems. We categorize DSU system into two classes based on either (1) support or (2) no support for multithreading.

2.2. Techniques for dealing with dynamic software updating issues

2.2.1. Unit of update. In a software application with DSU capabilities, once the new version of a running program becomes ready, the execution should be taken from the old version and carried on with the new one. Therefore, the decision about what parts of the program should be reloaded into memory can be an issue. To this end, dynamic updating systems employ three different avenues. In the first, whole code of the updated program is loaded into memory. In the second technique, only modified modules of the updated program are loaded. Finally, in the third one, the program is divided into several update units. Then, whole code of an update unit is loaded into memory if any of its modules have been modified. On the basis of these evaluation metrics, we categorize DSU system into these three classes: (1) whole program replacement, (2) module update, and (3) update unit.

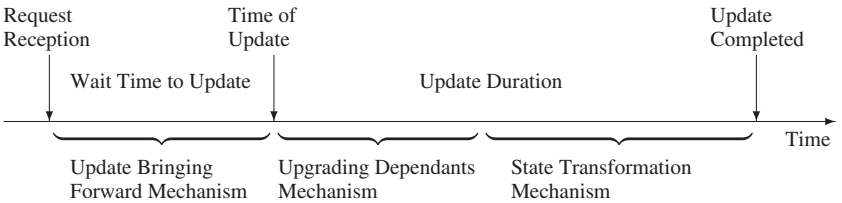


Figure 1. Time-line of dynamic updating.

DOI: 10.1002/smr

2.2.2. Upgrading dependants of an updated module. Upgrading dependants of an updated module specifies the way in which dependants are redirected to the updated module. This is an essential step of the dynamic updating process when the unit of update is 'Module Update' or 'Update Unit'. Of note, because modules can usually be loaded into memory without any particular issue, we know 'Loading Modules into Memory' as a mandatory part of 'Upgrading Dependants of an Updated Module' feature rather than a separate evaluation metric.

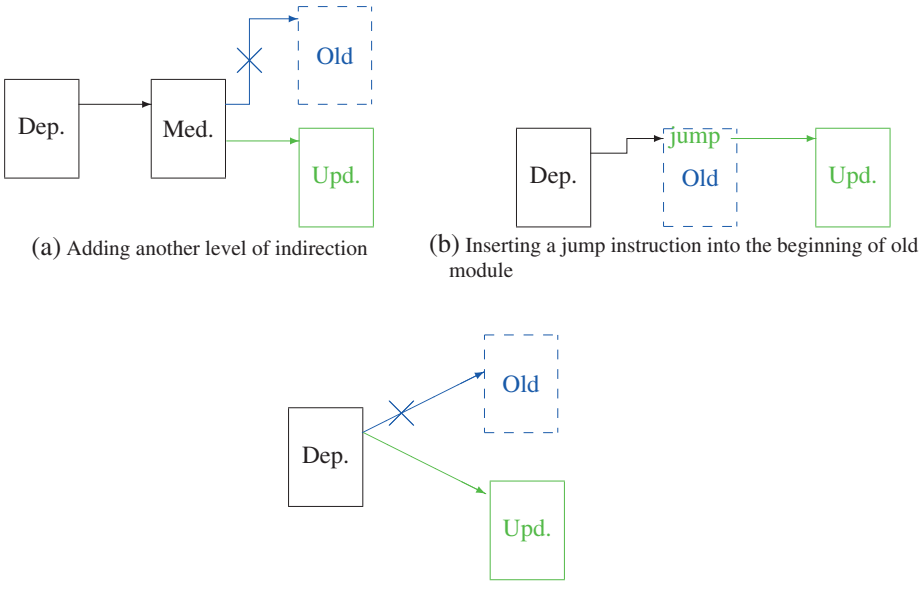
The first approach to this issue simplifies updating dependants by adding a level of indirection to all accesses in the program. Each function call or data access is performed through a data structure usually referred to as dynamic symbol table, wrapper class, pointer to function, etc. Then, it suffices to update that data structure to upgrade all dependants of the updated module.

Figure 2(a) shows this technique. A box represents a program's module and an arrow from box Dep to box Old/New means that module Dep is dependent on module Old/New. Examples of dependency include function calls and variable accesses.

In the second technique, a jump instruction is added into the beginning of an old module so that it redirects accesses into the new module. Like in the previous one, this technique obviates the need to upgrade every dependant of the updated module. This technique is illustrated in Figure 2(b). In the third technique, the old version of a module is replaced with its new version at the same location. Despite implementation issues especially because of the new code size exceeding its old version, this approach does not require address correction of changed modules within their dependants.

As the fourth technique, a number of DSU systems scan the whole program and correct the address of newly loaded module throughout its dependants. This technique is shown in Figure 2(c). Therefore, we divide techniques for upgrading dependants of an updated module into four categories: (1) adding another level of indirection, (2) proxification (we borrow the term *Proxification* from the *Dynamic Correspondence Framework* (DCF) [25]), (3) replacement, and (4) upgrading all dependants. We believe that any other technique for upgrading dependants will ultimately fall into one of these four categories.

2.2.3. Update atomicity. Atomicity refers to this fact that DSU system stops the whole program's execution during applying the requested dynamic update rather than applying it in parallel with the program's normal execution. The update atomicity plays a crucial role in maintaining the balance between consistency and service disruption duration. According to the papers reviewed in this survey, three proposed techniques deal with atomicity to date.



(a) Adding another level of indirection

(b) Inserting a jump instruction into the beginning of old module

(c) Updating references of all dependants

Figure 2. Various methods for upgrading dependants of an updated module.

DOI: 10.1002/smr

The first technique carries out the entire update process in a completely atomic step. This constitutes the most consistent way to update while making the system unavailable during the update. In the second technique, the entire update process is performed in parallel with the program's normal execution. Although the program continues to run during the update, the new version of an updated data structure may possibly be used by the old version of an updated function or vice versa, which can lead to inconsistencies during the program's execution if the DSU system does not synchronize state between versions.

In the third technique, the update process is separated into steps that either do not jeopardize consistency or that pose some risk. Risk-free steps are then run in parallel with the program's execution and risky steps are run in a separate, atomic action. Risk-free steps to consistency include loading updated modules into the memory or queuing incoming requests and risky steps include state transformation. The third technique balances consistency of being updated applications against the service disruption because of applying updates, and therefore, this is more powerful than other techniques in this area. Consequently, DSU systems fall into three categories based on the atomicity: (1) atomic, (2) nonatomic, and (3) partially atomic.

2.2.4. State transformation. State transformation technique of a DSU system transfers the state of an updated module's old to its new version. Without state transformation, a dynamic updating system becomes a stop/starting service, which adds no advantage compared with an off-line update method. State transformation composes six DSU categories, most are being described here for the first time in the dynamic updating literature.

The first category, which refers to the data structure used for transferring state, is divided to two subcategories: (1a) copy and (1b) shadow. In the copy method, the value of the updated data structure is transferred into a new location. Shadow, by contrast, continues to use the old data structure after the update, but adds some pointers to the added fields.

Concerning the method by which a new application reaches the proper state, the second category comprises two subcategories: (2a) programmer-specified state transformers and (2b) re-executing. In programmer-specified state transformers, some functions responsible for transferring the state at run-time are specified by the programmer and then embedded into the dynamic patch or the old application to be executed at update times. In re-executing, the execution is rolled back to the nearest checkpoint in which the state of the old application and its new version are the same. Then, the new version is executed until the current state in the new application is reached.

Regarding what is being transferred, the third category creates two subcategories: (3a) global state transformation in which the state of a program's global variables is transferred, and (3b) active function state transformation in which the local state of active updated functions is transferred. The local sate transformation is more difficult because the stack segment of the running program should also be transferred (in addition to the data segment and heap). For this reason, only few DSU systems support such transformations.

Other categorizations that is related to time of state transformation divides existing DSU systems into two subcategorizes: (4a) eager state transformation, which means that the state of the entire application is transferred immediately after upgrading dependants of the updated modules, and (2) lazy state transformation, which means that the state of a module is transferred before access of that module for the first time after applying the dynamic update. We borrowed the terms 'eager' and 'lazy' from HotSwap [3]. The lazy state transformation recently seen in DSU systems can shorten the update duration if it is used properly, not leading to program's inconsistencies.

With regard to the kind of state transformations, we have categorized the state transformation techniques into two subclasses: (5a) value state transformation in which only data values are transferred and (5b) type state transformation in which data types may be changed in addition to their values. As an example, if there is a structure with two fields x and y in a program and the programmer wants to update the structure so that it contains three fields x, y, and z, a type state transformation should be performed.

Finally, where state transformation is performed divides DSU systems into two subcategories: if we assume that the state transformation is a process of transferring state from a source module to a target one, then (6a) in the direct transformation, transferring is performed by the target module or the DSU

DOI: 10.1002/smr

system. (6b) In contrast, in the indirect transformation, state is first exported to a standard format by the source module and this standardized state is then imported by the target module.

Figure 3 shows the classification of DSU systems based on 'State Transformation' evaluation metric categories. Of note, we have gathered the first and the second categories to a more general category called way to state transformation in the figure.

2.2.5. Type-safety. A program is type-safe if every module in the program has a type that its dependants expect it to have. As an example, an integer data type should not be interpreted as a Boolean by its dependants. Type-safety violation may occur in a DSU system only when the system supports either change of modules' interface and nonatomicity or change of modules' interface and unit of updates smaller than the whole program.

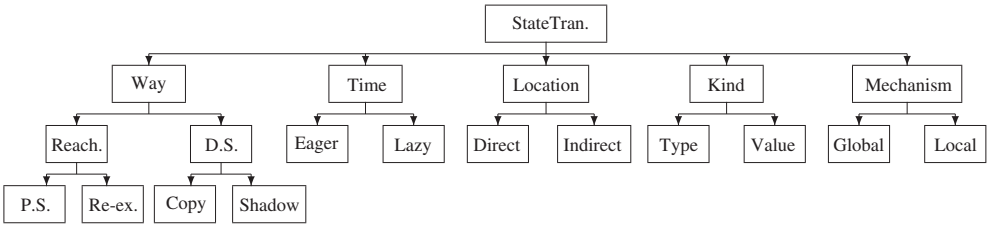
There are four different techniques for keeping a program type-safe when it is dynamically updated. In the first technique, the DSU system does not preserve type-safety. The idea behind this approach is that programmers usually update modules and their dependants simultaneously instead of changing a function's signature but not updating functions that call that modified function. In the second technique, the programmer is required to provide a stub function for each modified module. This stub function is responsible for getting the requests from old dependants and convert to intelligible format in the updated module.

In the third technique, the DSU system reorders received updates so that the type-safety is guaranteed. For instance, if an update for a module arrives, it postpones the update until updates for its dependants are received. In the fourth and last technique, DSU system places some constraints on update sets to ensure that the set is type-safe. As the most popular constraint, a dynamic updating system may require that if a module is in the update set, then all of its dependants should also be in the set. These four techniques divide DSU systems into one of the following categories: (1) ostrich, (2) stub modules, (3) update reordering, and (4) type-checked updates.

2.2.6. Time of update. Because time of update has a major impact on a number of other DSU evaluation metrics such as consistency, it is one of the most important metrics of a dynamic updating system. The greatest challenge of update timing is to establish a trade-off between applying updates quickly and consistently. DSU systems are generally divided into two categories based on this evaluation metric: (1) immediate DSU systems that apply updates immediately and (2) delayed DSU systems that postpone applying updates until a proper time at which they estimate the update does not violate the program's consistency. Each of these two categories has several subcategories to be discussed in the following paragraphs.

Immediate update systems are subdivided into three categories: (1a) *Strict* systems that check for a criterion such as nonactivity of updated functions each time an update request is received and reject the request immediately if the criterion is not fulfilled. (1b) *Multi-version* systems that update every modified module regardless of its activity. Each active function carries on with its old version, while new invocations cause the new version to run. (1c) *Updating active code* systems that can update active modules. Because this subcategory poses several challenges to DSU systems, it will be discussed in a separate paragraph in Section 2.2.7.

Delayed update systems also fall into two categories based on (2a) the technique for determining the appropriate update time and (2b) the technique for bringing forward the update. With respect to



Information: Tran.: Transformation, P. S.: Programmer Specified, Re-ex.: Re-executing, Reach.: Reaching the proper state, D. S.: Data Structure

Figure 3. Categorization of DSU systems based on 'State Transformation'.

DOI: 10.1002/smr

determining the appropriate update time, we observed four techniques in the literature. The first and most popular technique postpones the update until none of the updated modules remains active in the running program. The system either checks the condition periodically or checks it at each function return.

In the second technique, the programmer determines the update time by specifying *update point*s in his/her program. The DSU system then dynamically updates the program when the execution reaches the first update point occurrence after arrival of the update request. The third technique postpones applying updates until none of the modules to be updated participates in any active transaction.

In the fourth technique, the DSU system periodically checks whether any update request has been received in the interim because of its last check and applies the update if there is any. Therefore, delayed DSU systems are divided into four subcategories in regard to determining the appropriate update time: (2a1) no active module, (2a2) programmer specified update points, (2a3) quiescence, and (2a4) periodic. Of note, most DSU systems that use programmer specified as their update time, suggest that programmers specify update points at the end of the application's infinite main loop. We believe that this special case of programmer specified approach is equal to quiescence technique, because no active transaction exists at the end of an infinite main loop.

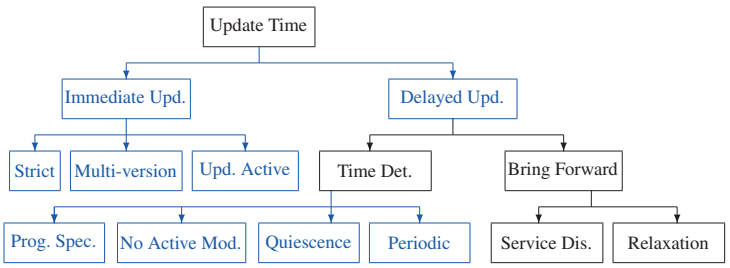
In regard to bringing forward the update, two techniques exist. In the first, the running program rejects incoming requests until the update point is reached causing different parts of the program to converge into the update point sooner. In the second technique, points of the program, which are equivalent to the programmer specified update points, are found and marked as further update points. Because the number of update points are increased in this approach, the wait time before reaching each is most probably decreased. In abstract, there are two methods to bring forward the update: (2b1) service disruption and (2b2) relaxation.

To avoid confusion, we provided a figure (Figure 4) that shows the classification of dynamic updating systems based on the categories that 'Time of Update' constructs. Moreover, Figure 1, which was used to show the wait time to update and update duration times, can also be used here to show the time at which the update is started and the times at which different DSU mechanisms are leveraged.

2.2.7. Updating active (nonquiescent) code. Updating active code refers to upgrading functions in the midst of their execution. DSU systems designed to apply updates upon request should support active code updates. The difficulty with updating active code arises from two issues. The first is to find a point in the updated code from which the execution of active functions should be continued. The second is to transfer local state of active functions so that they can continue their execution with the correct state. Here, only the techniques developed to deal with the former are provided (For more information about local state transformation techniques, the reader is referred to Section 2.2.4).

In the first technique, the programmer specifies functions that can be updated during their execution along with a correspondence table that contains a map from several points of old functions to their equivalents in the updated functions. In case of an update request, the DSU system lingers until the execution reaches one of the points specified in the correspondence table, and then switches the execution from the old function to its new version using the information provided in the table.

The second technique is useful for updating active functions that, although never inactive, are blocked regularly during their executions. To dynamically update such functions, the technique waits until the



Information: 1- Red boxes show disjointed categories 2- Dis.: Disruption, Upd.: Update, Mod.: Module, Prog. Spec.: Programmer Specified, Det.: Determination

Figure 4. Categorization of DSU systems based on 'Time of Update'.

DOI: 10.1002/smr

function is blocked rather than inactive. One example of this approach is updating an operating system's scheduler, which never exits but instead is blocked during an interval between two scheduling processes. Although both of these techniques update nonquiescence functions, neither can with a zero delay time.

In the third technique, the update is applied immediately after it arrives regardless of where in the active code it is received. DSU systems that take this approach usually reuse the state of the old active code to a point that the old and new codes are the same and execute the new code from that point until the correct state is reached. In summary, the three avenues are (1) programmer-specified update points, (2) block times, and (3) just-in-time updates.

2.2.8. Level of code differencing. Dynamic software updating systems, which update only modified parts of an application, should discover these upgrades before taking action. This process, which we call 'Code Differencing', is performed by comparing old and new application code. This feature has been subject to extensive work to date encompassing DSU systems that compare low level structures such as object code and those that compare high level structures such as abstract syntax trees.

The motivation claimed for using lower level structures is that such structures facilitate code difference discovery compared with code of higher structures. On the other hand, lower level structures make for needless updates because of negligible differences resulting in unnecessary code replacements at run-time. For instance, if two consecutive independent lines in a function are exchanged, the object code triggers a function update despite the function's overall behaviour remaining unchanged. Such unnecessary replacements cause longer update durations and are the main reason why some DSU systems target higher level code than the object [26].

In other DSU systems, the programmer explicitly specifies modified modules by a language. In these systems, therefore, the differencing tool should become a differencing language. With respect to this evaluation metric, we categorized dynamic updating systems into four subclasses: (1) DSU systems that use object code, (2) DSU systems that use source code, (3) DSU systems that use higher level structures than source code, and (4) DSU systems in which the programmer explicitly describes what parts of the program have been changed.

2.3. Dependencies amongst evaluation metrics

In this section, the affect/depend relationships among evaluation metrics are described. Because 'depend' and 'affect' are opposites (if A affects B, then B depends on A), indicating one relationship means indicating the other. Therefore, to avoid redundancy, we will consider only 'affect' relationships except for a few evaluation metrics such as type-safety and simplicity because of their more complex interdependency. In addition, we have constructed a pseudo-ontology (see Figure 5) diagram to graphically illustrate all depend/affect relationships between distinct evaluation metrics together in one place.

2.3.1. Level of abstraction. The level of update abstraction employed by a programmer can affect the 'Simplicity' of the dynamic updating system used; the higher the level of abstraction, the simpler the DSU system.

2.3.2. Wait time to update and predictability. Predictability weighs heavily in real-time or operating system applications; 'Wait Time to Update and Predictability' affects the 'Scope' evaluation metric of DSU systems. The other DSU evaluation metrics' impacts on 'Scope' have not been elaborated in this survey because of their much lower significance.

2.3.3. Supported changes. If a DSU system is able to change modules' interface (when combined with nonatomicity or when the unit of update is smaller than the whole program), the type-safety of updated application may be violated. The reason for this is that the dependants of an updated module may still expect the old interface from that module. Techniques that avoid such circumstances are mentioned in Section 2.2.5. 'Supported Changes' also impacts 'State Transformation'. If 'Supported Changes' of a DSU system is able to change modules' interface, then its state transformation mechanism should support transferring of types besides values.

2.3.4. Programming language. This feature can affect 'State Transformation', 'Ability to Update Previously Started Code', 'Simplicity', and 'Multi-Threading Support' evaluation metrics. Because a

DOI: 10.1002/smr

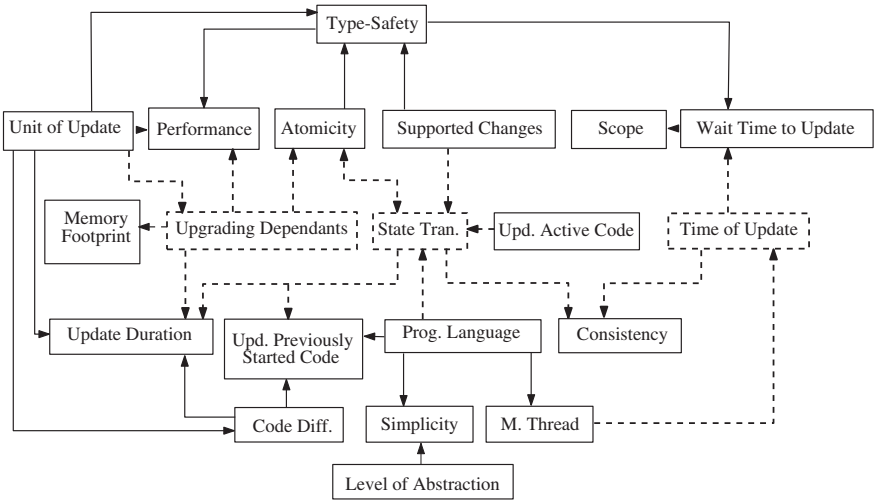


Figure 5. A pseudo-ontology of DSU evaluation metrics.

large amount of objects are usually initiated from a class in an object-oriented application and because all of these objects require state transformation when their class is updated, the object-orientation affects state transformation. In addition, establishing an unpopular DSU enabled programming language renders that language unable to dynamically update running programs compiled by other daily used languages that do not have dynamic updating capabilities.

According to 'Simplicity', using an unpopular programming language forces the programmer to learn a language that cannot be used in other areas. Similarly, using a modified popular programming language causes the programmer to learn the language's modified parts. These learned modifications degrade the DSU system's simplicity. With respect to the last dependent metric, only DSU systems that are implemented in multithreaded language may support multithreading.

2.3.5. Multithreading support. Delayed DSU systems that support multithreading must wait until all threads satisfy the delay constraint. This means that the multithreading support metric affects 'Time of Update', and consequently, 'Wait Time to Update'. Therefore, the best delayed DSU systems from a multithreading point of view are those systems that maximize the bring-forward time of update.

2.3.6. Unit of update. Unit of update can affect 'Upgrading Dependants of an Updated Module', 'Level of Code Differencing', 'Update Duration', 'Performance Overhead', and 'Type-Safety'. Concerning 'Upgrading Dependants of an Updated Module' and 'Level of Code Differencing', the systems do not require to adopt any approach to them if its unit of update is 'Whole Program Replacement'.

According to 'Update Duration' and 'Performance Overhead' evaluation metrics, the unit of update becomes larger, the update duration degrades, but the performance improves and vice versa. The reason for this is that with large update units the system should reload a large update unit because of small changes within it most of the time, resulting in longer update durations. On the other hand, because the majority of DSU systems place extra code between update units to simplify the update process, larger update units require fewer units and fewer code added to the application's original code. That translates into better dynamic updating performance.

'Type-Safety' is the last evaluation metric that depends on the unit of update. If the unit of update is smaller than the whole program in a DSU system, and if also the interface of modified module is allowed to be changed, type-safety may be violated, a situation that should never occur in running programs.

2.3.7. Upgrading dependants of an updated module. This evaluation metric can affect 'Performance Overhead', 'Update Time Duration', 'Code Clean-up', and 'Atomicity'. Systems that leverage additional level of indirection decrease the program's performance because they convert all variable

DOI: 10.1002/smr

and function accesses to the indirect access. On the other hand, systems that scan the whole program at update times require long update durations because whole program scanning consumes time. Regarding 'Code Clean-up', if a DSU system does not remove the updated modules after updating its dependants, memory leakage occur.

Finally, a DSU system becomes nonatomic if 'Upgrading Dependants of an Updated Module' is performed in parallel with the program's normal execution. The system becomes partially atomic if only updating steps that cannot violate program's consistency (e.g., loading updated modules into the memory) are performed in parallel with the program's execution.

2.3.8. Update atomicity. Atomicity evaluation metric affects 'State Transformation' and 'Type-Safety' metrics. As previously mentioned discussing atomicity, nonatomic updates can violate program's consistency. To avoid such conditions, the state transformation should cover program's accesses during the update so that each function accesses only the correct data version. In other words, if the update is applied in a nonatomic fashion, then the state transformation mechanism should become a state synchronization mechanism to prevent program inconsistencies. According to 'Type-Safety', nonatomicity, when combined with the 'Can Change Modules' Interface' approach of 'Supported Changes', can result in type-safety violations.

2.3.9. State transformation. This evaluation metric affects 'Consistency', 'Atomicity', 'Update Duration', and 'Ability to Update Previously Started Code'. According to 'Consistency', the transformation should be converted to the synchronization mechanism in the systems that apply updates nonatomically to avoid program inconsistencies. Choosing 'Lazy' technique as the time of state transformation makes the DSU system nonatomic. This explains why state transformation influences atomicity.

With respect to 'Update Duration', it has a direct relationship with state transformation. A short state transformation results in a shorter update duration and vice versa. Regarding the last evaluation metric, if the location of the state transformation is indirect in a DSU system, the old and the new versions of a component should negotiate for transferring state, and consequently the old version should be aware of dynamic updating. Therefore, previously started applications not having dynamic updating capabilities cannot be updated by such system.

2.3.10. Type-safety. Because there is a possibility that stub modules are placed between each updated module and its dependants, the performance of running systems may be degraded when a DSU system wants to avoid type-safety violations. In addition, update reordering used here causes the 'Wait Time to Update' of updates to be altered. Therefore, type-safety impacts on 'Performance Overhead' and 'Wait Time to Update' evaluation metrics.

2.3.11. Time of update. 'Time of Update' metric can affect 'Consistency' and 'Wait Time to Update and Predictability' features. We refer the reader again to the example given in Section 2.1.5, which shows how bad timing can lead to program inconsistencies. Which technique for 'Time of Update' is superior is discussed in Section 4, but suffice it to say that the mentioned example shows that 'No Active Module' technique cannot preserve consistency under all circumstances, which unfortunately is the technique used by many DSU systems.

'Time of Update' also impacts on 'Wait Time to Update' evaluation metric which gets longer if the DSU system applies updates after longer delays. In the ideal case, the wait time to update becomes zero if the DSU system applies updates really immediately. Moreover, delayed DSU systems that do not bring forward the update apply updates at unpredictable times.

2.3.12. Updating active (nonquiescent) code. A DSU system with updating nonquiescence code capability should also support local state transformation, so that it can transfer the state of active code being updated. Therefore, updating active code affects 'State Transformation' evaluation metric.

2.3.13. Level of code differencing. 'Level of Code Differencing' affects 'Update Duration' and 'Ability to Update Previously Started Code' evaluation metrics. It impacts on the former because code differencing at lower levels of code forces DSU systems to apply unnecessary updates, resulting in longer update durations. Code differencing impacts on the latter because in the systems that leverage source code or

DOI: 10.1002/smr

higher level structures, the source code of the currently running application must be present. Therefore, an application cannot be dynamically upgraded unless its source code is at hand.

To graphically represent affect/depend relationships between evaluation metrics as discussed above, we have constructed a pseudo-ontology diagram (see Figure 5). In this figure, a box represents an evaluation metric while arrows between boxes indicate that one evaluation metric affects another or another depends on one. The purpose of this diagram is to assist those who wish to design dynamic updating systems focused on one or more evaluation metrics of interest. For example, if one wanted to design a DSU system with focus on performance, then type-safety, unit of update, and upgrading dependants of an updated module evaluation metrics should be taken into consideration and incorporated into the DSU composition.

We purposefully omitted certain relationships like reverse or indirect relationships from the figure. For instance, the performance and the update duration usually have a reverse relationship or the time of update indirectly impacts on the state transformation through the updating active code evaluation metric, but these relationships have not been shown in the diagram for an enhanced reading experience. In addition, because the three DSU evaluation metrics 'Time of Update', 'Upgrading Dependants of an Updated Module', and 'State Transformation' along with relationships among them weigh prominently in this survey, these metrics have been shown by a dashed line to provide a top–down view of evaluation metrics based on their importance.

These three features are important for us because without taking a proper approach to them in an application being either renders the application inconsistent or severely limits scope of the DSU in such a system. For example, without 'State Transformation' mechanism the DSU system becomes merely a stop/start service. It is noted that other features such as performance or level of abstraction are important, but do not play such a pivotal role in a DSU system.

3. REVIEW OF DYNAMIC SOFTWARE UPDATING SYSTEMS

In this section, we proceed to review and compare the most influential DSU research papers based on the evaluation metrics described in Section 2. Each evaluation metric has a separate paragraph in the section and DSU systems are categorized in that paragraph based on a technique for the related evaluation metric. We want to draw the reader's attention to the fact that if a study did not incorporate a technique for a given feature we omitted it from the paragraph, discussing only systems that have the most mature or state-of-the-art techniques for that evaluation metric. To illustrate, we present Table I indicating capabilities of DSU system and Table II indicating techniques it develops to deal with DSU issues. With these tables at hand, research areas of dynamic software updating ignored or to whom little attention has been paid can be more easily identified indicating subjects of future interest.

3.1. Capability-based review

3.1.1. Scope. To the best of our knowledge, *MultiCS* [7] was the first operating system with a limited set of dynamic updating features similar to the 'Edit and Continue' debugging technique recently added to Microsoft C# [27]. Appavoo *et al.* converted a running into an autonomous system at run-time [21]. They tested their method on the K42 operating system.[‡] Nevertheless, dynamic updating of the K42 operating system was originally established and supplemented by Soules *et al.* [28] and Baumann [2], respectively.

AutoPod updates the operating systems' kernel by allowing user-space processes to first migrate to other machines [29]. AutoPod updates the Linux operating system's kernel. Ksplice, a practical DSU system, has been designed to update the Linux operating system's kernel at run-time [30]. DynAMOS is another DSU system that updates active, nonexiting functions of the operating system (e.g., scheduler) [19]. The authors updated the Linux scheduler to show the system's applicability.

The *Buisson*, *Hashimoto*, and *ReCaml* formal frameworks focus on upgrading active functions in functional programming languages [31–33]. None of these systems have been implemented

[‡]http://www.research.ibm.com/K42/

DOI: 10.1002/smr

Table I. Evaluation of DSU research works based on capabilities and constraints.

	Scope*	Implemented	High level of Abs.	Upd. pre. started code	Simplicity	Consistency	Wait time and predict.†	Update duration	Code clean-up	Performance	Supported changes	Type of language‡	Popularity of language§	Multihreading Supp.
Appavoo	OS	✓	×	×	✓	✓✓	P	✓	×	✓	×	O	P	✓
Argus(A)	DS	✓	×	×	×	×	I	✓	×	×	×	P	U	✓
Argus(S)	DS	✓	×	✓	×	✓	P	✓	×	✓	×	P	U	✓
AutoPod	OS	✓	×	×	✓✓	✓✓	I	×	×	×	✓	P	P	✓
Bialek	D	✓	×	×	✓	✓	U	✓	×	✓	✓	O	P	✓
Boyapati	B	✓	×	×	×	✓	I	✓	×	✓	✓	O	U	×
Buisson	×	×	×	×	?	✓✓	I	?	?	?	?	F	U	×
DCF	D	✓	✓	×	✓✓	×	I	✓	×	×	✓	O	P	✓
DRACO	DS	✓	×	✓	✓✓	✓✓	U	✓	×	✓	✓	O	C	✓
Duggan	×	×	×	×	?	✓	I	✓✓	?	×	✓	F	U	×
DURTS	RT	✓	×	×	✓	×	I	✓✓	×	✓	×	P	P	✓
DUSC	D	✓	×	×	✓✓	✓	I	✓	×	✓	×	O	P	×
DVM	D	✓	×	×	✓	✓	I	×	×	✓✓	✓	O	C	✓
DYMOS	S	✓	×	×	✓	✓✓	P	✓	×	✓	✓	P	U	✓
DynAMOS	OS	✓	×	✓	✓	✓✓	I	✓	×	×	✓	P	P	✓
DynC++	S	✓	×	×	✓	×	I	✓	✓	✓	×	O	P	✓
Ekiden	S	✓	×	×	✓✓	✓	I	×	—	✓✓	✓	P	P	×
EmbedDSU	RT	✓	×	×	✓✓	✓	P	✓	×	✓	✓	O	C	×
Fabry	B	×	×	×	?	✓	I	✓	×	✓	✓	P	U	×
Ginseng	S	✓	×	×	✓✓	✓✓	U	✓	×	✓	✓	P	C	✓
Giuffrida	×	×	×	×	?	✓✓	P	?	?	?	×	P	?	×
Gracioli	RT	✓	×	×	✓✓	✓	I	✓	✓	✓	×	O	P	×
Gupta	S	✓	×	×	✓✓	✓	U	×	—	✓✓	×	P	P	×
Hashimoto	×	×	×	×	?	✓✓	I	?	?	?	?	F	U	×
Hicks	S	✓	×	×	✓	✓✓	U	✓	×	✓	✓	P	U	×
HotSwap	D	✓	×	×	✓	×	I	✓✓	×	✓	×	O	P	×
Jalili	S	✓	×	✓	✓	✓✓	U		×	✓	✓✓	P	P	✓

(Continues)

DOI: 10.1002/smr

Table I. *(Continued)*

	Scope*	Implemented	High level of Abs.	Upd. pre. started code	Simplicity	Consistency	Wait time and predict.†	Update duration	Code clean-up	Performance	Supported changes	Type of language‡	Popularity of language§	Multihreading Supp.
JVOLVE	S	✓	×	×	✓✓	✓✓	U	×	×	✓	✓	O	C	✓
K42	OS	✓	×	✓	✓	✓	U	✓	✓	×	✓	O	P	✓
Kim	S	✓	×	×	✓✓	×	I	✓	×	✓	✓	O	P	×
Ksplice	OS	✓	×	✓	✓✓	✓	P	×	×	✓	✓	P	P	✓
Lin	B	×	×	?	?	✓✓	U	×	?	?	✓	?	?	×
Lyu	S	✓	×	×	✓	✓	U	✓✓	×	✓	×	P	P	×
MultiCS	OS	✓	×	✓	×	✓	I	✓✓	×	✓✓	×	P	U	×
OPUS	M	✓	×	✓	✓✓	✓	U	✓✓	×	✓	×	P	P	✓
OSM	S	✓	✓	×	✓✓	×	P	✓	✓	✓	✓	O	P	×
POLUS	S	✓	×	✓	✓✓	✓	I	✓✓	×	×	✓	P	P	✓
Proteus	S	×	×	×	?	✓✓	U	✓	?	✓	✓	F	U	×
ReCaml	×	×	×	×	?	✓✓	I	?	?	?	✓	F	U	×
Seif	D	✓	✓	×	✓✓	×	I	✓✓	×	✓	×	O	P	×
Seif-Real	RT	×	×	?	?	✓	P	×	?	?	?	?	?	×
Sidiroglou	S	✓	×	✓	✓✓	×	I	✓✓	×	✓	×	P	P	×
UpgradeJ	D	×	×	×	?	✓✓	U	✓✓	?	✓	✓	O	C	×
UpStare	M	✓	×	×	✓✓	✓✓	I	✓	×	✓	✓	P	C	✓
UpStart	DS	✓	×	×	✓✓	✓	U	✓	✓	×	✓	—	P	×
Wahler	RT	✓	×	×	✓	✓	P	✓	×	✓	×	O	P	×
YAO	D	✓	×	✓	✓	×	I	✓	×	×	✓	O	P	×
Zhang	S	✓	×	×	✓	✓	U	×	×	✓	✓	O	P	✓

Systems are ordered alphabetically; —, not required; ?, not specified; ×, no approach; ✓, low; ✓✓, high.
*B, business and database; D, desktop; DS, distributed systems; M, multiple scope; OS, operating systems; RT, real-time; S, Server.
†I, immediate; P, predictable; U, unpredictable.
‡P, procedural; F, functional; O, object-oriented.
§P, popular; C, popular with change; U, unpopular.

DOI: 10.1002/smr

Table II. Evaluation of DSU research works based on techniques for issues.

				State Trans						Time				
System	Unit of update*	Upgrading Deps†	Atomic updates‡	Way§	Type transform	Lazy transform	Local transform	Location¶	Type-Safety‖	Delayed: time det.**	Delayed: bring For.‡‡	Immediate update‡‡	Update active code***	Code Differencing
Appavoo	M	L	P	P	×	×	×	I	—	Q	D	—	×	×
Argus(A)	M	L	×	P	×	×	×	I	—	—	—	M	×	×
Argus(S)	M	L	✓	P	×	×	×	D	—	N	D	—	×	×
AutoPod	M	L	P	P	✓	×	✓	D	—	—	—	A	—	×
Bialek	U	L	P	P	✓	×	×	D	—	N	×	—	×	✓
Boyapati	M	L	×	P	✓	✓	×	D	R	—	—	M	×	×
Buisson	M	?	✓	R	✓	×	✓	D	×	—	—	A	J	?
DCF	M	P	×	P	✓	✓	×	D	T	—	—	M	×	×
DRACO	M	L	P	P	✓	×	×	I	—	Q	R	—	×	✓✓✓
Duggan	M	L	×	P	✓	✓	×	D	S	—	—	M	×	×
DURTS	M	L	×	×	×	×	×	×	—	—	—	M	×	×
DUSC	M	L	✓	P	×	×	×	D	—	—	—	S	×	×
DVM	M	A	✓	P	✓	×	×	D	T	—	—	S	×	✓✓
DYMOS	M	L	P	P	✓	×	✓	D	S	P	×	—	P	×
DynAMOS	M	P	P	P	✓	×	✓	D	T	—	—	A	B	×
DynC++	M	L	×	P	×	×	×	D	—	—	—	M	×	×
Ekiden	W	—	✓	P	✓	×	✓	I	×	—	—	A	P	—
EmbedDSU	M	L	✓	P	✓	×	×	D	R	N	D	—	—	✓✓✓
Fabry	M	L	×	P	✓	✓	×	D	×	—	—	M	×	×
Ginseng	M	L	✓	P	✓	✓	×	D	S	P	R	—	×	✓✓✓
Giuffrida	M	?	P	P	×	×	×	D	—	P	D	—	×	×
Gracioli	M	L	P	P	×	×	×	D	—	—	—	S	×	×
Gupta	W	—	✓	P	×	×	×	D	—	N	×	—	×	—
Hashimoto	?	?	✓	R	✓	×	✓	D	×	—	—	A	J	✓✓✓
Hicks	M	L	✓	P	✓	×	×	D	S	P	×	—	×	✓✓
HotSwap	M	L	×	×	×	×	×	×	—	—	—	M	×	×
Jalili	M	—	P	P	✓	×	×	D	S	P	×	—	×	✓✓✓
JVOLVE	M	L	✓	P	✓	×	×	D	×	P	×	—	×	✓✓
K42	M	L	×	P	✓	✓	×	D	S	N	D	—	×	×
Kim	M	L	×	P	✓	×	×	D	—	—	—	M	×	✓
Ksplice	M	P	✓	P	✓	×	×	D	×	N	×	—	×	✓

(Continues)

DOI: 10.1002/smr

Table II. *(Continued)*

System	Unit of update*	Upgrading Deps†	Atomic updates‡	State Trans: Way§	State Trans: Type transform	State Trans: Lazy transform	State Trans: Local transform	State Trans: Location¶	Type-Safety‖	Time: Delayed: time det.**	Time: Delayed: bring For.‡‡	Time: Immediate update‡‡	Update active code***	Code Differencing
Lin	?	?	✓	?	✓	?	?	?	?	P	?	×	?	?
Lyu	M	P	P	×	×	×	×	×	—	N	×	—	×	×
MultiCS	M	L	✓	×	×	×	×	×	—	—	—	S	×	—
OPUS	M	P	✓	×	×	×	×	×	—	N	×	—	×	✓✓
OSM	M	L	✓	P	✓	×	×	D	R	C	×	—	×	×
POLUS	M	P	×	P	✓	✓	×	D	S	—	—	M	×	✓✓✓
Proteus	M	P	✓	P	✓	✓	×	D	T	P	×	—	×	—
ReCaml	M	?	✓	P	✓	×	✓	D	×	—	—	A	J	?
Seif	M	L	×	×	×	×	×	×	—	—	—	M	×	×
Seif-Real	M	?	✓	?	?	?	?	?	?	C	×	—	×	?
Sidiroglou	M	P	×	×	×	×	×	×	—	—	—	M	×	✓✓
UpgradeJ	M	L	✓	×	✓	×	×	×	—	P	×	—	×	×
UpStare	M	L	✓	P	✓	×	✓	D	×	—	—	A	P	✓✓
UpStart	M	L	×	P	✓	×	×	D	S	P	×	—	×	×
Wahler	M	L	P	P	×	×	×	D	—	C	×	—	×	×
YAO	M	L	×	P	✓	×	×	D	R	—	—	M	×	✓✓
Zhang	M	A	×	P	✓	×	×	D	T	N	×	—	×	✓✓

Ordered alphabetically; —, not required; ?, not specified; ×, no approach; ✓, low; ✓✓, high.
*M, module; W, whole; U, update unit.
†L, another level; A, all; P, proxification.
‡P, partially atomic.
§P, Prog. Sp.; R, re-executing.
¶D, direct; I, indirect.
‖S, stub; T, type-checked; R, reordering.
**P, Prog. Sp.; N, no active; Q, quiescent; C, cycle.
††D, service disruption; R, relaxation.
‡‡S, strict; A, active code; M, multiversion.
***P, Prog. Sp.; B, Block time; J, JIT.

DOI: 10.1002/smr

and evaluated yet. Duggan's research, also not yet implemented, aims to preserve type-safety and consistency in systems that use the multiversion technique to update timing [34]. Similarly, unimplemented *Giuffrida* also focuses on update timing [35].

Among DSU systems that focus on server scope, *DYMOS* has been the pioneering system [36] featuring active functions upgrade and type-safety preservation. DYMOS has been implemented in banking system server updating. The *Gupta* DSU system has also been evaluated by a print server [37]. In a separate research, Gupta *et al*. determined that the exact time of update is generally undecidable [38]. A system similar to but more modern than Gupta is *Ekiden* [39], which is evaluated by updating the vsftpd server.[§]

Lyu is a DSU system designed to dynamically update programs in Solaris operating system environment [40]. Lyu has dynamically updated an Internet gateway server. The other system, *POLUS* [18] updates server applications such as vsftpd, OpenSSH,[¶] and Apache[‖] servers. POLUS has significant features such as binary compatibility, update roll-back, and recovery of tainted state before applying updates.

Hicks, *Proteus*, and *Ginseng* are the most practical and consistent dynamic updating systems in the server scope [1, 41, 42]. Hicks *et al*. evaluated their system by updating the FlashED web server. Proteus is a calculus capable of determining whether or not a point in a program is safe for the dynamic updating of a given set of data definitions. Ginseng dynamically updates vsftpd, OpenSSH, and Zebra** servers using ideas described by Proteus. *JVOLVE* [43] is a DSU system that uses the same ideas as Ginseng, but its implementation environment is Jikes *Research Virtual Machine* (RVM) [44] and its evaluation applications are Jetty,[††] JavaEmail,[‡‡] and CrossFTP[§§] servers.

Jalili is a Ginseng variant with the aim of reducing the performance overhead incurred by DSU [20]. Jalili updates the same servers as Ginseng, although the authors tested it on a sample list copying program. *Sidiroglou* [45] is another DSU system that executes both patched and unpatched codes at run-time to reduce the chance of system's failures caused by the new patch [46]. The authors did not specify their evaluation technique.

To our knowledge, *DynC++* was the first system to dynamically update object-oriented programs [47]. It has been used to dynamically update connection management services in telecommunication systems. *Online Software Maintenance* (OSM) System enhances system abstraction [48]. The developers applied OSM to a File Transfer Protocol server.

Kim supports all types of changes to the Java classes with little performance degradation [49]. Kim has been tested by updating the server portion of a *Remote Method Invocation* (RMI)-based application. *Zhang* updates programs safely at run-time without any additional framework or language requirement [50]. The authors indicated that the system dynamically updates a server.

OPUS and *DUSC* are two practical DSU systems, although they do not support all update types [51, 17]. DUSC has been evaluated using a regression test tool called DEJAVOO [52], and OPUS has been tested on various servers and desktop applications including MySQL server,[¶¶] Apache, and MPlayer.[‖‖] *HotSwap* enables programmers to update previously built Java applications. To this end, HotSwap's developers have embedded it into the standard Java HotSpot Virtual Machine. HotSwap has been used to add profiling to the running applications.

Dynamic classes-enabled Virtual Machine (DVM), a modified version of *Java Virtual Machine* (JVM), supports a large set of changes along with setting up a mechanism to preserve program type-safety after updates [53]. DVM has been tested on desktop applications especially compilers. *Bialek* improves the performance of updatable applications and has been used to update an instant messenger dynamically [54].

[§]Very secure ftp daemon, http://vsftpd.beasts.org/
[¶]OpenSSH secure shell server, http://www.openssh.com/
[‖]Apache web server, http://httpd.apache.org/
**GNU Zebra routing software, http://www.zebra.org/
[††]Jetty web server, http://jetty.codehaus.org/jetty/
[‡‡]JavaEmail mail server, http://javaemailserver.sourceforge.net/
[§§]CrossFTP ftp server, http://www.crossftp.com/
[¶¶]MySQL database server, http://www.mysql.com/
[‖‖]MPlayer Movie Player, http://www.mplayerhq.hu/

DOI: 10.1002/smr

Seif s purpose is to simplify update expressions used by programmers to specify those parts about to be updated [55]. The authors have tested the system using a simple desktop application. *DCF* dynamically updates programs with a high level of flexibility and programmer transparency [25]. It has been used to enhance *Integrated Development Environments* (IDEs) reload functionality. In addition, the authors suggested a JVM, called *Javeleon*, to reduce the performance degradation caused by DCF [56].

UpgradeJ is an unimplemented formal model for lightweight updating of applications [57]. The authors show its applicability [58] by investigating update repositories of several desktop applications including ant,*** ANTLR [59], Eclipse,††† and Weka [60]. *YAO* aims to preserve program type-safety when updates of different parts are not received simultaneously [61]. A drawing application has been dynamically updated to show the applicability of the system [62].

UpStare has an improved state transformation technique that is able to update active functions at run-time [63]. The developers evaluate their system by updating KissFFT library,‡‡‡ vsftpd, and PostgreSQL server.§§§

We noticed that *Fabry*, *Boyapati*, and *Lin* [64–66] focused on the business and database scope. Fabry and Boyapati preserve database consistency and type-safety by synchronizing data types with their dependants, while Lin, an empirical study, explores issues of and approaches to the dynamic updating of databases by investigating update repositories of popular database servers.

Bloom and Day adopted two similar approaches to dynamically update programs written in Argus, a language for building distributed systems [67]. The big difference between the two approaches is that one operates at the application level while the other operates at the language level. *UpStart* [68] is another powerful DSU system that updates *Remote Procedure Call* (RPC)-based distributed programs dynamically [69]. *DRACO*'s focus is distributed systems also [70]. It updates components with the so-called Live Update Module. DRACO also establishes a novel idea for update timing, *tranquility*, which is described more in Section 3.2.6.

DURTS updates real-time applications [71] by modeling updates as a transient fault and using slack times to apply them without missing a deadline. *Seif-Real* updates real-time functions that have modified timing properties by employing a schedulability test function [72]. Seif-Real is an as of yet unimplemented system.

Wahler et al. [73] dynamically updates real-time processes of Integrity operating system.¶¶¶ *Gracioli* [74] is a DSU system that remotely updates running components executed in the *Embedded Parallel Operating System* (EPOS) [75]. *EmbedDSU* [76], which updates Java-based smart cards, has been implemented in an embedded Java virtual machine called SimpleRTJ.‖‖‖ Among the significant features of this system we find support for update granularity and rolling back to the previous update versions in a system with limited resources to be outstanding features.

3.1.2. Level of abstraction. Dynamic software updating systems reviewed in this survey operate at the code level, except for Seif, DCF, and OSM. Seif *et al*. extracted new code of the updated class with an XML-based language specified by the programmer. An XSLT processor dynamically converts these requirements to Java source code. DCF supports changes in class inheritance in addition to normal class updates. A subclass in DCF dynamically distinguishes its super-class and directs nonoverridden method requests to it. Although implicitly supported by other systems, DCF explicitly supports this change.

OSM enables programmers to either merge or split C++ classes emphasizing the level of abstraction. Other updating tasks are automatically performed by OSM. For example, when a class is split into two parts, OSM corrects references of dependants automatically so that each of them refers to the correct part after the update. OSM does so by keeping track of the class's dependants in a container and a list of smart pointers.

***Ant java build tool, http://ant.apache.org/
†††Eclipse IDE, http://www.eclipse.org/
‡‡‡KissFFT fast Fourier transform library, http://sourceforge.net/projects/kissfft/
§§§PostgreSQL database server, http://www.postgresql.org/
¶¶¶Integrity real-time operating system, http://www.ghs.com/products/rtos/integrity.html
‖‖‖Simple Real-Time Java, http://www.rtjcom.com/

DOI: 10.1002/smr

3.1.3. Ability to update previously started code. Dynamic software updating systems that do not provide this feature either propose a new DSU-enabled language (e.g., Duggan, Buisson, Giuffrida, and ReCaml), meaning that programmers should consider dynamic updating into program writing, or create a new compiler and runtime environment (e.g., DVM, Ginseng, UpStare, and JVOLVE), meaning that previously started program should be stopped and executed in the new environment. Several other systems need the source code to be converted to a DSU-enabled form (e.g., DUSC and Ekiden) or require special command-line parameters passed to the compiler (e.g., HotSwap and Seif). Hence, programs should be recompiled after the conversion.

On the other hand, MultiCS and K42 are able to update previously started code, because they use only dynamic loading capabilities of the operating system that are available to all programs. Also, Ksplice and DynAMOS exploit *Linux dynamically-loadable kernel modules* that every program in Linux uses. Similarly, Argus(S) utilizes only the system's resources to update programs without the need for applications with dynamic updating features.

OPUS, POLUS, and Jalili use the *ptrace* system call of the Linux operating system to attach updated code to a normally running application. Sidiroglou leverages the *dyninst* runtime instrumentation tool to do a similar operation. YAO and DRACO execute Java programs in special middlewares and use dynamic loading features of those middlewares to reload updated components. Such Java middlewares usually use custom class loaders to do reloading.

3.1.4. Simplicity. MultiCS, Argus(A), Argus(S), and Boyapati satisfy none of our simplicity parameters: They have not been implemented in a popular language, do not operate at a high level of abstraction, and do not set up a tool to automate patch building/applying tasks.

Although Hicks' programming language is unpopular, it automatically generates dynamic patches. Therefore, it is a simple to use system. Similarly, DYMOS provides a tool with which programmers can write and apply updates, a feature we believe makes it user-friendly as well. By contrast, Lyu, Jalili, YAO, DVM, HotSwap, Zhang, Bialek, DynC++, Appavoo, K42, DynAMOS, DURTS, and Wahler do not provide a tool for programmers to describe or apply updates, although their programming languages are popular. Therefore, we place these designs also in the user friendly category.

Gupta, Ekiden, Ginseng, POLUS, OPUS, UpStare, Sidiroglou, DUSC, JVOLVE, Kim, DRACO, UpStart, Ksplice, AutoPod, Gracioli, and EmbedDSU are DSU systems that not only use a popular programming language, but also provide a tool to help the programmer build and apply updates. These systems are very simple in our opinion. Although Seif does not provide a tool for the programmer, we also mark this system very as simple because it satisfies the other two simplicity parameters. The best DSU systems from a simplicity point of view are DCF and OSM, which fulfill all of our simplicity parameters.

3.1.5. Consistency. Sidiroglou, YAO, HotSwap, Seif, DCF, Kim, OSM, DynC++, Argus(A), and DURTS fail to implement an adequate update timing technique and have an inadequate state synchronization mechanism when more than one version of a module exists in the memory. Consequently, these systems lack consistency in our assessment. Fabry, Duggan, and POLUS synchronize state between versions under these circumstances. We would rate these three systems as consistent with remark, because we believe that writing such state synchronization functions is not practical in real-life situations. Boyapati reorders updates in the existence of multiple versions of a data store in the memory. Hence, it is also consistent with remark DSU system. Similarly, UpStart is also consistent with remark, although it does not provide state synchronization functions. The reason for this is that each node's state in the distributed systems is usually not accessible by other nodes.

We mark MultiCS, Gupta, OPUS, Lyu, DVM, DUSC, Zhang, Bialek, Argus(S), K42, Ksplice, Seif-Real, Wahler, and Gracioli consistent with remark, because their technique for update timing is 'No Active Module', and as mentioned before, this does not completely guarantee consistency. DYMOS is a more consistent system, because it enables the programmer to specify function(s) s/he wants to be idle at the update time. Also, EmbedDSU is consistent with remark because its technique for update timing is similar to 'No Active Module'.

Because determining the exact update time is generally undecidable, DSU systems that allow the programmer to choose his/her desired update points can be considered consistent. These systems include Proteus, Hicks, Ginseng, Jalili, JVOLVE, UpgradeJ, and Lin. The programmer usually chooses

DOI: 10.1002/smr

the end of the main loop as the update point in these systems. In Giuffrida, which is more powerful than the above systems in this area, each module has a separate update point meaning that the update is applied only when all modules reach their update points.

DRACO and Appavoo update a module when it finishes its execution in all active transactions. The consistency of these systems has been proved in [77]. Buisson, Hashimoto, Ekiden, UpStare, and DynAMOS do not cause aversion inconsistency, because they discard the old version immediately in its entirety when the update is received. AutoPod does not require a consistency mechanism. The reason for this is that the processes only migrate from one node to another in this system.

3.1.6. Wait time to update and predictability. Proteus, Hicks, Gupta, Ginseng, OPUS, Lyu, Jalili, JVOLVE, Zhang, UpgradeJ, Bialek, DRACO, UpStart, K42, and Lin wait until a special condition such as quiescence is fulfilled before upgrading. Therefore, these are hard-to-predict DSU systems. The condition worsens in Bialek, Hicks, JVOLVE, OPUS, UpStart, and Zhang, because they also support multithreading, and therefore, should wait for all threads to converge. Ginseng, Giuffrida, DRACO, Argus(S), Appavoo, and K42 reduce the wait time to update by employing an update forward-bringing technique.

MultiCS lingers for the programmer to correct the faulty part of the program, and then compiles and reloads the corrected version into the memory. Fabry, Duggan, POLUS, Sidiroglou, YAO, HotSwap, Seif, DCF, Kim, DynC++, Argus(A), DURTS, and Boyapati immediately apply updates by allowing old requests to continue executing old functions while new requests call the functions' new version. Buisson, Hashimoto, ReCaml, Ekiden, UpStare, and DynAMOS immediately apply updates by upgrading active functions and DVM, DUSC, and Gracioli immediately update programs by checking if the desired conditions are satisfied upon update arrival and rejecting the request if they are not fulfilled. AutoPod also updates the operating system's kernel immediately by migrating user processes from one computer to another.

Predictable DSU systems, which include DYMOS, Giuffrida, OSM, Argus(S), Appavoo, Ksplice, Seif-Real, Wahler, and EmbedDSU, either use time-out timer or periodically check for new update requests.

3.1.7. Update duration. Gupta and Ekiden cause long update durations, because they replace the entire old version of the program, which results in unnecessary replacements at the upgrade time. Jalili's update duration is also long because it converts the running program to a nonupdatable version after replacing the outdated modules. Updates in DVM, JVOLVE, and Zhang also last long because these systems use 'Upgrading All Dependants' technique for the upgrading dependants of an updated module.

Ksplice's low level code differencing may lead to long update durations because of unnecessary update applications. AutoPod transfers user-space processes of an operating system to another computer to apply an update also prolonging its duration. Finally, Lin requires long update times because it suggests to update applications with their databases atomically.

MultiCS, OPUS, Lyu, and UpgradeJ result in short update durations because of the absence of a state transformation mechanism. Duggan and POLUS also lead to short update durations because they transfer state lazily, not in the update duration. Other systems result in reasonable update durations. Although they lengthen the update duration because of their eager state transformation, they compensate with the 'Adding Another Level of Indirection' technique used for upgrading dependants of an updated module. The best update durations are achieved in Sidiroglou, HotSwap, Seif, and DURTS, which only set references of the updated modules' dependents at the time of update.

3.1.8. Code clean-up. Most DSU systems either ignore code clean-up or do not measure the amount of memory they use. The exceptions are Gupta, Ekiden, DynC++, OSM, Gracioli, K42, UpStart, and Jalili, which are discussed next.

DynC++, OSM, and Gracioli measure only the amount of memory occupied by their system in the absence of any effort at code clean-up. UpStart and K42 remove stubs interposed between the updated module and its dependents when the dependents are also upgraded. Jalili converts the running application to the normal nonupdatable version when the update is finished to omit the memory footprint caused by the DSU-enabled code. Gupta and Ekiden are two prominent DSU systems with

DOI: 10.1002/smr

respect to code clean-up. They discard the program's old version entirely upon switching to its new version. In Ekiden, the only DSU-related code added to the main function is state transformation.

3.1.9. Performance overhead. Duggan and POLUS degrade the applications' performance because of their state synchronization mechanisms in addition to access indirections used for upgrading the updated module's dependents. Argus(A) and UpStart impose overhead on the running applications because they use multiple levels of indirection instead of just a single one.

AutoPod executes user-space processes in a virtual machine imposing considerable overhead on programs. YAO, DCF, K42, and DynAMOS degrade the running program's performance because of their access indirection techniques. For example, DCF executes a 19-line pseudocode for each method invocation to call the correct version of the desired function.

Gupta, Ekiden, DVM, and Jalili do not degrade performance. The reason for this is that the first three do not use 'Adding Another Level of Indirection' for upgrading the dependants of an updated module and the latter converts the new program after the update to the nonupdatable version. Other systems impose little performance overhead on the running program because they only include code for access indirections and/or stub functions.

3.1.10. Supported changes. In Kim and UpgradeJ, modules are only able to add fields and methods to their original version. Delete or modify operation are not permitted, because these two systems utilize inheritance properties of the Java language to evolve the running classes. Gupta merely suggests supporting the modules' interface change but has not yet implemented a technique to do so. Lyu claims no limitation on the modules' interface change, although it recommends not yet using the feature because of the lack of maturity.

Bialek and DRACO support all changes inside the update unit, but units must provide the same interface during the program's execution. Similarly, AutoPod allows interface change inside the operating system's kernel, but not between the kernel and user-space processes. Other systems that support unrestricted module interface changes include Fabry, DYMOS, Proteus, Duggan, ReCaml, Hicks, Ekiden, Ginseng, POLUS, Jalili, UpStare, YAO, DVM, JVOLVE, DCF, Zhang, OSM, UpStart, K42, Ksplice, DynAMOS, EmbedDSU, Boyapati, and Lin.

3.1.11. Programming languages. Giuffrida is designed to dynamically update C programs but has not been implemented yet. Therefore, the exact properties of its language are unknown to us. Fabry and DYMOS have been implemented in the SIMULA and StarMod**** programming languages, respectively. Therefore, we would rate their language as procedural and unpopular. Argus(A) and Argus(S) use Argus†††† as their programming environment. Their language is procedural and unpopular just the same. Hicks has been implemented in Popcorn, a type-safe variant of the C language. Hence, Hick's language is similarly procedural and unpopular.

Proteus, Buisson, Hashimoto, Duggan, and ReCaml establish new DSU-enabled functional languages.‡‡‡‡ Thus, we would rate their language as functional and unpopular. Boyapati is designed to dynamically update data objects of Thor DBMS [78] and is rated object-oriented and unpopular. Ginseng and UpStare use altered procedural and popular languages, because they have created special C compilers to update applications. DVM, textscJvolve, DRACO, and EmbedDSU are similar but they focus on the Java compiler.

With respect to programming languages, DSU systems that use unaltered popular languages are more powerful. Gupta, Ekiden, POLUS, OPUS, Lyu, Jalili, Sidiroglou, and DURTS update C programs without changing the language's syntax and semantics. Therefore, their languages are procedural and popular. They usually place dynamic updating into a library that can be loaded into memory alongside the normal application. Ksplice, DynAMOS, and AutoPod use an unchanged C programming language to dynamically update the Linux operating system, which places them in our procedural and popular category.

Likewise, YAO, DUSC, HotSwap, Seif, DCF, Zhang, Bialek, and Kim update Java programs without altering the language itself, and OSM, DynC++, Wahler, and Gracioli use the unchanged C++ language as

****StarMod is a distributed variant of Modula programming language.
††††Argus is a distributed systems' programming language.
‡‡‡‡Their languages are similar to OCaml and ML.

DOI: 10.1002/smr

their implementation environment. Their language is object-oriented and popular. Developers of K42 and Appavoo implement their ideas in C++ language to add DSU to the K42 operating system, making their system's language object-oriented and popular as well. UpStart is a powerful DSU system in the programming language area. It updates every distributed application provided that it communicates using RPC.

3.1.12. Multithreading support. Bialek, DYMOS, JVOLVE, Ksplice, OPUS, and Zhang are delayed DSU systems that support multithreading. Appavoo, Argus(S), DRACO, Ginseng, Jalili, and K42 are delayed multithreaded DSU systems that leverage update time bringing-forward techniques to reduce the threads wait time. Ginseng and DRACO are superior in this aspect because they use relaxation, causing minimum service disruptions at run-time. Immediate DSU systems that support multithreading include Argus(A), AutoPod, DCF, DURTS, DVM, DynAMOS, DynC++, POLUS, and UpStare. They do not require thread convergence as a prerequisite prior to engagement.

Hicks and Proteus provide ideas to support multithreading. However, we mark them nonmultithreaded because they do not implement the provided ideas. Although Boyapati, Duggan, DUSC, Giuffrida, Gracioli, HotSwap, Seif, and UpStart are not multithreaded, extending them to support multithreading is straightforward because their update timing approach is multiversion and/or they have been implemented in a multithreaded programming language. Other, unmentioned, DSU systems do not support multithreading, or at least, their developers do not describe the system's behaviour in such environments.

3.2. Technique-based review

3.2.1. Unit of update. Gupta and Ekiden replace the entire application when an update occurs. Other DSU systems take the 'Module' approach to the unit of update issue. Specifically, the update unit is a *class* in object-oriented systems, and *function*s in procedural ones. The exception is EmbedDSU, which supports variable granularity of changes, from a field in a class to the entire class itself. This prevents unnecessary upgrades at run-time.

DynC++ and Gracioli partition programs into two updatable and nonupdatable classes imposing no DSU overhead on nonupdatable classes. Also, Bialek partitions the program into update units specified by the programmer at the compilation time. Bialek, DynC++, and Gracioli employ the best techniques for dealing with the update unit provided that the partitions are determined correctly. They keep a balance between update duration and the program's performance overhead.

3.2.2. Upgrading dependants of an updated module. Gupta and Ekiden do not require this feature because they use the 'Whole Program Replacement' technique for the unit of update issue. Similarly, Jalili does not need this feature after it converts the program into the nonupdatable version.

POLUS, DCF, Sidiroglou, DynAMOS, OPUS, Lyu, and Ksplice use proxification to upgrade dependants of an updated module. DCF, a powerful DSU system in this area, instruments the program's classes using the ASM tool [79] to identify those not represented by the latest version and forward incoming requests to the correspondence framework in case they are. DCF is then responsible for directing requests to the latest version. Proteus also suggests using proxification but does not specify implementation details.

Dynamic classes-enabled Virtual Machine and Zhang upgrade all dependants within reach of an updated object by using the garbage collectors techniques. Other systems choose the 'Adding Another Level of Indirection', the simplest way to upgrade dependents. Systems developed in C language leverage *pointer-to-function*s or *dynamic shared libraries* to implement this technique while object-oriented systems leverage *wrapper classes* to this end. Other object-oriented systems, which include HotSwap, Seif, JVOLVE, and K42, use the platform's internal data structures to set up another level of indirection.

Deciding between the above techniques depends on what performance requirements a system should meet. Jalili, Gupta, Ekiden, DVM, and Zhang result in good performance but long update duration, while others lead to short update duration but degraded performance. Of note, DSU systems that use the platform's internal data structures to create another level of indirection degrade performance less than those that exploit wrappers.

To reload modified program parts, DSU systems either make use of *renaming* or *custom loaders*. *Java Reflection API* is also used in several Java systems.

DOI: 10.1002/smr

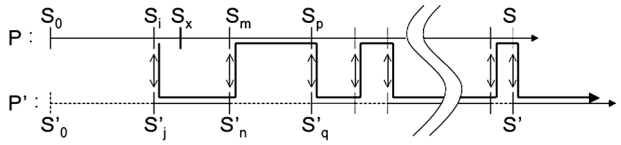


Figure 6. State transformation using re-executing the new version.

3.2.3. Update atomicity. Fabry, Duggan, POLUS, Sidiroglou, YAO, HotSwap, Seif, DCF, Kim, DynC++, Argus(A), DURTS, Boyapati, Zhang, UpStart, and K42 update applications nonatomically. Wahler *et al.* have recently proposed a nonatomic DSU system in which the state is synchronized between versions [80]. On the other hand, MultiCS, Buisson, Hashimoto, ReCaml, Ekiden, DVM, DUSC, Proteus, Hicks, Gupta, Ginseng, OPUS, JVOLVE, UpgradeJ, OSM, Argus(S), Ksplice, Seif-Real, EmbedDSU, and Lin apply updates atomically.

With respect to the more powerful partially atomic systems, Gracioli, Giuffrida, Appavoo, and K42 queue incoming requests such that no client request is lost during the update. UpStare also suggests a similar idea. DYMOS, Lyu, and Wahler reload updated program parts in parallel with the program's execution while transferring their state and upgrading their dependents in an atomic operation. Jalili updates the updatable version of a program atomically but switches between nonupdatable and updatable versions in parallel with the program's execution. Also, Bialek and DRACO update each update unit atomically while the other units continue with their execution. AutoPod updates the operating system's kernel while its user-processes run on other machines.

3.2.4. State transformation. MultiCS, Sidiroglou, HotSwap, Seif, DURTS, OPUS, Lyu, and UpgradeJ do not provide a state transformation mechanism and are not discussed in the following paragraphs. Most DSU systems choose the 'Programmer-specified' way with respect to the state transformation because of its simplicity and flexibility in the global state transformation. It is worth noting that DVM, DCF, Gupta, and EmbedDSU only copy data values or initialize the updated program's variables to default values, but they also suggest the programmer-specified technique for future work. Argus cites an interesting example to show that the programmer-specified technique cannot be applied in all circumstances. By the example, Argus demonstrates that a correct state transformation may even need an update history in some cases, even though it also makes use of the simple programmer-specified technique for implementation.

Object-oriented DSU systems that use the programmer-specified option should execute transformers for every object in the program. To this aim, several systems scan the heap and transfer the state of all objects found onto it (e.g., JVOLVE). Other systems employ wrappers[§§§§] to transfer objects' state (e.g., K42). In these systems, the wrapper performs like a *factory* to maintain a list of objects instantiated from the updated class [81].

In spite of the programmer-specified way, Buisson and Hashimoto use 'Re-executing' to transfer the state. Buisson executes the new version from where the program has been rolled back to the point corresponding to the last executed statement in the old version. Hashimoto, more powerful in this area, executes the new version from the point of first difference between old and new versions rather than the nearest roll-back point defined by the programmer. In addition, Hashimoto proposes to reuse the old version's state in parts of the program in which old and new versions are the same. Therefore, the updated state is achieved by a process of 'execute-and-reuse'.

Figure 6 illustrates this technique in which the horizontal axis shows state transitions occurring during execution of each program and the vertical axis shows transferring state from one program to its corresponding state in another program. Evidently, 'Re-executing' requires less programmer intervention than 'Programmer-specified', but is more difficult to implement in DSU systems.

With respect to the kind of state transformation, DSU systems that transfer the type in addition to the value are more powerful. These systems include Fabry, Buisson, Hashimoto, Duggan, ReCaml, Ekiden, POLUS, UpStare, YAO, DVM, DCF, Kim, DynAMOS, AutoPod, Boyapati, DYMOS, Proteus, Hicks, Ginseng, Jalili, JVOLVE, Zhang, UpgradeJ, Bialek, OSM, DRACO, UpStart, K42, Ksplice, EmbedDSU, and Lin.

[§§§§]Wrapper classes are described in 'Upgrading Dependants of an Updated Module' paragraph.

DOI: 10.1002/smr

Regarding the time of state transformation, DSU systems, which transfer state lazily, are more significant because of their prevention of unnecessary movements at run-time. We recognized state transformation of Fabry, Duggan, and POLUS as lazy because they install state synchronization functions executed after the update. DCF and K42 also utilize the lazy state transformation because they transfer the updated objects' state once accessed after upgrading. Every object in these systems has a Boolean variable showing whether or not it is up-to-date. Boyapati's state transformation is also lazy because it transfers the state of the updated object only after transferring its dependents' state.

Local state transformation used for transferring the state of the active functions is supported by DYMOS, Buisson, Hashimoto, ReCaml, Ekiden, UpStare, DynAMOS, and AutoPod. In AutoPod, the local state migrates from one machine to other ones. Buisson and Hashimoto adopt 'Re-executing' to the local state transformation while the other systems employ the 'Programmer-specified' approach. Bazzi *et al*. also suggested possible improvements regarding this issue [82]. Of note, upgrading active functions usually requires stack reconstruction because their state is on top. The exception is Ekiden, which discards the old program entirely and starts its new version.

With respect to the location of state transformation, DSU systems often utilize the direct technique, because no assumption is made about previous program versions. The benefit of the indirect technique, on the other hand, is that the state transformation is divided into two more simplified parts, only for one of which the updated program is responsible. Ekiden, Argus(A), and Appavoo employ the indirect technique also recommended by DRACO.

3.2.5. Type-safety. MultiCS, Sidiroglou, DUSC, HotSwap, Seif, Kim, DynC++, Argus(A), AutoPod, DURTS, Gupta,[¶¶¶¶] Gracioli, OPUS, Lyu, Giuffrida, UpgradeJ, Bialek, DRACO, Argus(S), Appavoo, and Wahler do not require a type-safety technique because modules' interfaces do not change in these systems. Similarly, Bialek, DRACO, Kim, and UpgradeJ do not require a type-safety preservation technique because of their limited support for modules' interface change. On the other hand, Fabry, Buisson, Hashimoto, ReCaml, Ekiden, UpStare, JVOLVE, and Ksplice require a type-safety approach but they do not adopt any. This is based on their belief that it is the programmer's responsibility to check type-safety and write state transformers that transform types in addition to values.

Dynamic classes-enabled Virtual Machine, DCF, DynAMOS, Proteus, and Zhang exploit the type-checked technique by checking whether the updated code violates the type-safety before the update is applied. For this, they use compilers or static code analyzers. Proteus dominates these systems. It confirms that no part of one program version accesses another version's part. To this end, Proteus inserts annotations (or coercions) into every program point indicating which data definitions are used there. It then unionizes coercions following an update point (which is named the *capability* of that update point) to find data definitions accessed after the point. Upon update arrival the update is applied, if the intersection of updated data definitions and capability of the next nearest update point is empty. In the case the update is not accepted, several techniques such as deferring it to the next update point or rejecting it can be chosen.

Dynamic software updating systems, which reorder updates to preserve type-safety, include EmbedDSU, YAO, Boyapati, and OSM. EmbedDSU and YAO defer updating of the modified part until its dependents finish execution. YAO provides an *update-dependency* graph in which vertices are parts of the program and an edge from vertex A to B refers to the update-dependency of part A on part B. Part A update-depends on part B if updating of part B results in type violations in part A (e.g., upon the function call or inheritance relationships). Strongly connected parts of the graph show those parts that should be updated simultaneously.

Boyapati enforces the correct order of updates by leveraging encapsulation. Every object in Boyapati should encapsulate objects on which it depends. This means that the dependent object is accessed and therefore updated before objects on which it depends. OSM uses the update-reordering technique by keeping both old and new versions of the object until its dependents are updated.

Duggan, POLUS, DYMOS, Hicks, Ginseng, Jalili, UpStart, and K42 use stub functions to preserve type-safety. The programmer provides these functions as part of the dynamic patch. UpStart is the most powerful system in this area. It maintains a list of stubs to be able to apply multiple updates

[¶¶¶¶] Gupta recommends using stub functions or type-checked technique in systems which support modules' interface change.

DOI: 10.1002/smr

simultaneously. When a client accesses a node in UpStart, the system determines the proper stub according to the client's version.

3.2.6. Time of update. Dynamic software updating systems that update applications without a proper technique for update timing include Sidiroglou, YAO, HotSwap, Seif, DCF, Kim, DynC++, Argus(A), and DURTS. When an update is received by these systems, each old function continues its execution while new invocations cause the new version to run. This technique, which is referred to as the multiversion time of update, is the simplest technique to implement. Fabry, Duggan, POLUS, and Boyapati leverage the same approach, although they avoid inconsistencies by using more powerful state transformation mechanisms.

The immediate dynamic updating systems that use the strict technique are DVM, DUSC, and Gracioli. The only condition examined by these systems to date is whether the updated part contains active functions or not. Usually, a counter in the wrapper classes maintains the number of active methods of the updatable unit. The last group of immediate DSU systems are those that update active functions. These systems, which are the best immediate systems in terms of consistency and wait time, include Buisson, Hashimoto, Ekiden, UpStare, DynAMOS, and AutoPod.

OSM, Seif-Real, and Wahler apply updates periodically and therefore are deterministically delayed DSU systems. Short periods in these systems have the benefit of applying updates sooner while long periods result in less performance overhead. For this reason, OSM provides a mechanism with which the programmer can set the desired period. By contrast, Seif-Real and Wahler update applications only at the end of each hyper-period when no process awaits execution.

Gupta, OPUS, Lyu, Zhang, Bialek, Argus(S), K42, and Ksplice leverage the 'No Active Module' technique for the time of update. Before applying an update, these systems examine the stack at each function call for an active updated function and, if so, defer the update to the next call. DSU systems using this technique are not compelled to deal with difficulties arising from the active functions upgrade.

To become more deterministic, Ksplice rejects an update if it fails to apply it in a predetermined number of attempts. Likewise, Argus(S) provides a mechanism by which the programmer can force an active function to exit. A more restrictive technique, EmbedDSU updates applications when both the updated function and its direct dependents are inactive.

DYMOS, Proteus, Hicks, Ginseng, Jalili, Giuffrida, JVOLVE, UpgradeJ, UpStart, and Lin leverage the practical programmer-specified technique for update timing. In DYMOS, which is a limited system in this area, the programmer can only determine functions that should be idle when the system is updated. UpStart and Giuffrida, the most powerful systems, allow programmers to specify a separate update point for each program part. The difference between these two systems is that UpStart updates the node when it reaches its update point regardless of other nodes' execution, while in Giuffrida all parts must reach their update points before the update can be applied. The other consistent technique for update timing is quiescence, first suggested by Kramer *et al*. [77] and utilized by Appavoo and DRACO.

Giuffrida, Appavoo, K42, and EmbedDSU block incoming requests to bring forward the update and DRACO and Ginseng, the most modern, use the relaxation techniques to do so. To relax the quiescence technique, DRACO leverages the following three facts: (1) updating a component that is used in an active transaction but has not yet been executed has the same effect as updating it when the transaction is inactive. (2) Updating a component that is used in an active transaction but will no longer be executed has the same effect as updating it when the transaction is inactive. (3) If the system is designed modular enough to divide a transaction into independent subtransactions, a component that belongs to one subtransaction can be updated when only its subtransaction is inactive. Therefore, DRACO lingers until at least one of the following conditions are fulfilled with respect to all active transactions that use the updated component when an update request is received: (1) the transaction becomes inactive, (2) the transaction has not yet started to use the component, (3) the transaction has finished using the component, and (4) the component belongs to subtransactions of the transaction and the related subtransactions become inactive.

Ginseng reduces the wait time to update caused by programmer-specified update points by (1) allowing a thread to continue its execution even if it reaches the update point using a concept called *relaxed synchronization* and (2) increasing the number of update points in the code with the addition of so-called *induced* update points. To check whether an induced update point is valid, Ginseng proposes the following approach: Under the assumption that an induced update point called *i* has been inserted

DOI: 10.1002/smr

between update point l_1 and l_2 (l_1 is prior to l_2), if no variable changed by update point l_1 is accessed by statements between l_1 and i (i.e., intersection of changes of l_1 and prior effects of i is empty), then i is valid and equivalent to l_1; similarly, if no variable changed by update point l_2 is accessed by statements between i and l_2 (i.e., intersection of changes of l_2 and future effects of i is empty), then i is valid and equivalent to l_2; otherwise, i is invalid and cannot be inserted in the current place.

An update point l and all of its equivalent induced update points create a block of code such that the update has the same effect regardless of the place it is applied in the block. Consequently, in the case of arrival of an update request, thread execution can be continued without interruption at the update point until it does not reach the end of the block. Ginseng developers call this idea 'relaxed synchronization'.

3.2.7. Updating active (nonquiescent) code. DynAMOS is able to update active nonexiting functions that block during their life time. The operating system's scheduler is one example of such functions. DynAMOS modifies the kernel's blocking routines to examine if an update request sits idle for the being blocked function. If so, the routines save the function's state, force it to exit, and continue with its new version.

DYMOS, Ekiden, and UpStare take the 'Programmer-specified Update Points' approach to update active functions. In UpStare, the programmer specifies a set of points in the old function at which the function can be updated and a set of corresponding points in its new version from which it must be continued. Ekiden switches execution to the new version's main function when the nearest update point is reached. The main function is responsible for transferring the state and running the appropriate new function.

Hashimoto, Buisson, and ReCaml update the active functions just-in-time. To do so, Hashimoto suggests executing the new version from a point at which the first difference between old and new versions occurs. In Buisson and ReCaml, the programmer tags the code[‖‖‖‖] showing points of the program at which the program can be dynamically updated. When an update is received, the execution is rolled back to the nearest tag and the update is applied. In the more modern ReCaml, the programmer may also provide a state transformer to be run at the time of update.

3.2.8. Level of code differencing. Kim and Bialek operate at the object level of code differencing, because they leverage Java byte-code to compare two code versions. Ksplice also chooses the object code to be able to extract differences that may be out of sight at the source code level. Method in-lining optimization performed by compilers is one example of such differences. To find functions to be replaced, Ksplice compares old and new version object code (pre–post differencing) and compares the old and currently running code to ensure that the old version is equal to the currently running version (run-prematching). Ksplice also reduces the number of unnecessary discovered updates by techniques such as ignoring the located addresses in the comparison process.

UpStare, Sidiroglou, YAO, DVM, Hicks, OPUS, JVOLVE, and Zhang compare the source code of two versions for differences, and Hashimoto, POLUS, Ginseng, Jalili, and DRACO use higher levels to achieve this aim. Specifically, POLUS uses CIL analysis and transformation tool [83] to simplify code differencing, and Hashimoto, Ginseng, and Jalili exploit the Abstract Syntax Trees (ASTs) to minimize the discovery of unnecessary changes. DRACO implements a powerful heuristic-based tool that generates a correspondence map between two code versions. The tool uses heuristics like counting the number of identical fields to find similar program parts. EmbedDSU utilizes Control Flow Graphs (CFGs) to find instruction blocks modified between two versions.

UpgradeJ and K42 build such code analyzers only for their empirical studies. These tools operate at higher levels than the source code. Other systems not mentioned do not specify the exact approach or require the programmer to exactly determine the updated parts of the application.

4. FUTURE RESEARCH DIRECTIONS

In this section, we will discuss the best techniques with respect to listed DSU evaluation metrics in the same order as described in Section 2. We also provide evaluation metrics that have not been considered

[‖‖‖‖]Buisson suggests using continuation operator which is available in functional languages to implement the roll back and forward in the running program.

DOI: 10.1002/smr

yet or techniques that should be supplemented to or enhanced by a feature. At the end, we mention the software development and engineering current trends, the holes that can be filled with the current DSU systems, and the holes that are yet to be filled in the software engineering trends.

4.1. Discussion of evaluation metrics

With respect to Scope evaluation metric, most DSU systems have focused on servers and desktop applications, but real-time and database applications have gained less attention requiring more research. For instance, we have not seen a DSU system that can update an application and its database simultaneously and consistently in DSU literature. Bhattacharya *et al*. has recently adapted the DSU techniques for dynamic updating of web and cloud applications, indicating a new and popular area that might get more attention in the future [84]. Remote updating, applying dynamic updates to a remote application, is another scope that has gained less attention to date. For instance, we do not notice a DSU system that remotely updates an application and takes an appropriate approach to the timing. By contrast, there is ample research with the aim of updating distributed systems in the literature. Another important feature especially in real-time, embedded, smart-phone, tablet, and cloud systems is 'Power Consumption'. No DSU system has yet focuses on this feature.

Regarding 'Level of Abstraction' evaluation metric, most DSU systems have concentrated on the source code level, and thus, we recommend systems that work on higher levels. As an example, a system that updates design or requirement modifications on-the-fly is missed, nowadays.

Several research studies into DSU systems that can update previously started code have recently emerged. Most use *ptrace* system call to this end and perform flawlessly. In addition (see Tables I and II), many DSU systems available today do not require the old version of program's source code for dynamic updating. Therefore, we feel that this space with the DSU field may have reached saturation in terms of further innovation.

Most DSU systems strive to prominently incorporate 'Simplicity'. Although many researchers have tried to preserve programming language syntax and semantics (which is one of our criteria for simplicity), a few have created a GUI or other kinds of interface to simplify user interactions. Hence, implementing DSU systems with such interfaces should be given more priority.

'Time of Update', 'Consistency', and 'Wait Time to Update and Predictability' are so interwoven that we cannot discuss one without discussing another. DSU systems that employ consistent techniques for 'Time of Update' frequently suffer from long wait time to update and unpredictability. Examples include systems that use 'Quiescence' and 'Programmer Specified Update Points' avenues. Therefore, research on bringing forward update in these systems is appealing. Some research has focused on doing so by introducing new concepts such as relax synchronization or tranquility [85, 86].

On the other hand, DSU systems that update applications shortly often suffer from program inconsistencies. Examples include 'No Active Module' and 'Multi-Version' avenues. Because, in our opinion, consistency is more important than wait time in the absence of special constraints, the latter approaches should be avoided in DSU systems. One exception is 'Updating Active Code', which update the running program immediately while it can also preserves consistency. However, Updating active code is one of the most difficult issues in the field of DSU, because of the difficulty in the local state transformation of updated functions that are active during an update. Some good research in this area exists (e.g., UpStare [63]), but active code upgrades should be more automated.

According to the update timing of real-time systems, 'No Active Module' technique can be used, although we mentioned it does not preserve consistency in all circumstances. The reason for this is that real-time functions are not so dependent on each other as functions of a normal application. For example, suppose a real-time flight control program that has two functions, engine temperature control and vertical speed control. Upgrade of a function in this application does not interfere in behaviour of the other function, regardless of update timing. Moreover, because real-time functions are usually short-lived, these systems can dispense with the 'Updating Active Code' technique and its difficulties. If one is forced to use a more consistent technique such as quiescence for time of update, then she/he should combine it with ideas in the strict systems to make the system deterministic. For instance, the system should reject the update if the quiescence constraint is not fulfilled in a determined number of attempts. This combination has not been seen in the literature yet.

DOI: 10.1002/smr

With regard to 'Update Duration' and 'Performance Overhead' evaluation metrics, they have been opposites to date. In other words, systems that have short update durations usually degrade program's performance and vice versa. Therefore, a system that can maintain a balance between update duration and performance is desired. Although some research has emerged in this area [87], no DSU system has practically implemented such ideas yet.

With respect to 'Code Clean-up', sufficient attention has been lacking and more work is necessary. Evidently, authors have not seen a DSU system whose main focus is code clean-up after the update. In addition, one should note that the memory footprint prevention in DSU systems is not limited to only code clean-up after the update. Many dynamic updating systems add DSU code to the applications at the compile time, meaning the application should contain extra code even if it does not use it at all. Therefore, a DSU system should reduce the code it adds to the program after and even before the update if it wants to tackle the memory footprint well.

Empirical studies show that changes to modules' interface is very common in application upgrades. Thus, in order for a DSU system to be able to dynamically update a wide range of applications, it should support changes to modules' interface. Otherwise, it may only be used in some special-purposed applications such as adding profiling behaviour or applying security patches to running programs.

Regarding 'Programming Languages', DSU systems that do not alter the syntax and semantics of the target language are easier to use by programmers. Another criterion is the popularity of the programming language. A DSU system that uses C language without changing its syntax and semantics is both popular and practical. Nowadays, such systems are not in short supply.

'Unit of Update' and 'Upgrading Dependants of an Updated Module' are evaluation metrics whose incorporation into a DSU system impacts on the performance and the update duration of the system under development. As mentioned before, the update duration and the performance have a reverse relationship and its reason is the techniques that are available for unit of update and upgrading dependants of an updated module nowadays. For example, choosing 'Module Update' as the unit of update causes a short update duration, while it usually degrades the application's performance. However, we believe that DSU systems should omit this reverse relationship by dynamically switching between different techniques for unit of update and upgrading dependants of an updated module.

With respect to 'Update Atomicity' we do not recommend 'Nonatomic' updates. This type of dynamic updating makes state transformation very difficult, or else consistency violations may occur. Instead, we recommend integrating 'Partially Atomic' techniques. Although these do not cause any type-safety or consistency problem, they shorten the service disruption time caused by applying updates. Although some systems have attempted to do so, to our knowledge, no work integrates all these related techniques.

One scope in which nonatomic updates are unavoidable is distributed systems. It is almost impossible that all nodes in a distributed system converge into a particular time at which dynamic updates are applied atomically. In such cases, we recommend that the state of each node is encapsulated in the node, so that its state is not accessible to other nodes and no data version inconsistency can occur.

Regarding 'Time of State Transformation', we think 'Lazy' technique would be more popular in the future, because it prevents DSU systems from performing unnecessary state transformations. Moreover, we believe that if the mechanism of state transformation is only 'Global', then the 'Programmer Specified State Transformers' will suffice for transferring the state of running program. Otherwise, some other methods of state transformation such as 'Re-executing' or 'Roll-back' should be used. The reason for our belief here is that when a program is updated, its functions might be drastically changed, but the overall behaviour of the program usually remains unchanged.

According to 'Type-Safety', if the application is developed by a single programmer and updates are applied atomically, the simplest action of a DSU system should be to do nothing. The rationale behind this suggestion is that if a single programmer modifies each application module, she/he usually modifies all the module's dependants as well. In addition, she/he can use the target language compiler to see if there is a type-safety error. Therefore, any other type-safety check seems unnecessary in this case.

On the other hand, alternative type-safety preservation techniques are helpful when updates are applied nonatomically and the interface of modules in the program might be changed. Note that 'Update Re-ordering' should be included as a part of type-safety preservation technique in these circumstances. If 'Stub Modules' method has been chosen in a special case, the DSU system should be able to remove stubs when dependants of the updated module are also upgraded.

DOI: 10.1002/smr

Table III. DSU evaluation metrics, their maturity level, and authors' remarks about each of them.

Evaluation metric	Maturity level	Remarks
Scope	Partially mature	Some scopes such as real-time, database, and tablet systems have not been mature yet. In addition, remote updating require more researches
Level of abstraction	Immature	High level languages for getting the updates are desired
Ability to start previously started code	Mature	*ptrace* system call can be used
Simplicity	Requires minor improvements	GUI or some other kinds of interfaces should be used
Time of update, consistency, wait time	Requires minor improvements	Updating active code preserves consistency and results in the minimum wait time, but it is difficult to implement. Techniques for making active code state transformation more automatic are required. Predictable delayed updates are immature. Programmer specified update points and Quiescence are the most consistent ways, but they are unpredictable
Multi-threading support	Require minor improvements	DRACO and Ginseng have carried out good researches in reducing threads' wait time in the delayed DSU systems. The immediate multi-threaded DSU systems should become more consistent and automatic
Update duration, code clean-up, performance	Immature	There is no work with focus on the mentioned evaluation metrics
Supported changes	Mature	Each DSU system must support changes to modules interface
Programming language	Mature	Implementing DSU in a popular programming language without changing its compiler or runtime environment is suggested
Unit of update, upgrading dependants	Requires minor improvements	One should select the appropriate technique based on performance needs
Update atomicity	Requires minor improvements	Integrating partially atomic and queueing requests techniques is suggested
State transformation	Requires minor improvements	For the time, *lazy* is suggested. For the way, if the mechanism is global, programmer specified transformers are suggested. Otherwise, it the mechanism is local, there is not any perfect way for the state transformation, yet
Type-safety	Mature	An ostrich approach is suggested when the application is written by single programmer
Level of code differencing	Requires minor improvements	Higher level tools are suggested. Several heuristics should be used

DOI: 10.1002/smr

In regard to 'Level of Code Differencing', we are considering a high level and easy-to-use language via which the programmer can specify an application updates. With such language in hand, a DSU system does not require to have a code differencing tool to discover differences between old and new version of the code. If no such language is available, the code-differentiating tools that work on high levels work more efficiently, because they detect less unnecessary updates than the tools that work on lower levels.

The justification behind using low level code-differentiating tools is that they can detect some updates that high level tools cannot. Method in-lining and implicit casting are examples that have been given as part of the rationale. However, we tend to differ with this rationale, because method in-lining or implicit casting are modifications that are applied at run-time and cannot be detected by comparing codes even at very low levels as in, for example, object.

To summarize this discussion, evaluation metrics of dynamic updating are shown in Table III once more adding the maturity level of each and our corresponding remarks.

4.2. Dynamic software updating position in software engineering trend

Software engineers have always considered two distinct aspects of cost, namely (1) implementation costs, and (2) maintenance costs. Nowadays, DSU'ing helps to reduce the latter especially in the server scope. We believe that the technology is mature enough to dynamically update simple one-loop servers such as File Transfer Protocol, Web, or Secure Shell servers.

In contrast, a big obstacle to dynamic updating preventing its use in many organizations is the persistent inability to update in parallel an application and its database. Typically in most organizations, a central database server is connected to several clients. Failure to dynamically update clients and their database in a synchronized manner makes DSU'ing unusable in such environments. In addition, DSU systems must support more commercial programming languages such as Microsoft C# or Visual Basic because these are widely used in business environments.

Another software engineering trend is the emergence of Cloud and mobile applications. The importance of dynamic updating in the Cloud lies in the fact that taking a whole Cloud offline and updating its nodes is impossible. In mobile applications, by contrast, DSU is not of major importance, because mobile applications are generally short-lived, except for their operating systems, which are rarely restarted.

5. CONCLUSIONS

This article presents a framework for the evaluation of dynamic updating features. We discuss this framework alongside the evaluation metrics defined in the literature within the context of 'cause and effect' relationships. A pseudo-ontology is provided to show these relationships among evaluation metrics available in the literature and in the proposed framework. This, in itself, is a contribution to the body of literature, because with this road map at hand, one can select the desired DSU evaluation metrics, ascertaining which techniques can be taken, and how such techniques might affect other DSU evaluation metrics.

Furthermore, through the review and classification of the most influential papers of the field, shortcomings of the evaluation metrics are identified and areas requiring further research are highlighted. Evaluation metrics such as performance, code clean-up and level of abstraction in addition to a number of scopes such as real-time, database, and web applications are examples of areas where more research is warranted.

ACKNOWLEDGEMENTS

We would like to thank Mostafa Kermani for providing us helpful ideas in the DSU area. We also thank Manouchehr Seyfzadeh, Hossein Seif Zadeh, and the anonymous reviewers for their constructive comments on drafts of this paper.

DOI: 10.1002/smr

REFERENCES

1. Hicks M, Nettles S. Dynamic software updating. *ACM Transactions on Programming Languages and Systems* 2005; **27**(6):1049–1096.
2. Baumann A. Dynamic update for operating systems. PhD thesis, Computer Science & Engineering, Faculty of Engineering, UNSW, 2007.
3. Dmitriev M. Safe Class and Data Evolution in Large and Long-Lived Java Applications. PhD thesis, Department of Computing Science, University of Glasgow, 2001.
4. Bidan C, Issarny V, Saridakis T, Zarras A. A Dynamic Reconfiguration Service for CORBA. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, Washington, DC, USA, 1998; 35. IEEE Computer Society.
5. Sallem MAS, da Silva FJ. Adapta: A framework for dynamic reconfiguration of distributed applications. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, New York, NY, USA, 2006; 10. ACM.
6. Bennour B, Henrio L, Rivera M. A reconfiguration framework for distributed components. In *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, New York, NY, USA, 2009; 49–56. ACM.
7. Multicians Group. MultiCS Dynamic Linkage Features, 2009.
8. Segal ME. Online Software Upgrading: New Research Directions and Practical Considerations. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, Washington, DC, USA, 2002; 977–981. IEEE Computer Society.
9. Vandewoude Y, Berbers Y. An overview and assessment of dynamic update methods for component-oriented embedded systems. In *Proceedings of the International Conference on Software Engineering Research and Practice*, 2002; 521–527.
10. Ajmani S. A Review of Software Upgrade Techniques for Distributed Systems. Technical report, MIT Laboratory for Computer Science, 2004.
11. Sterritt R, Parashar M, Tianfield H, Unland R. A concise introduction to autonomic computing. *Advanced Engineering Informatics* 2005; **19**:181–187.
12. Huebscher MC, McCann JA. A survey of autonomic computing â€" degrees, models, and applications. *ACM Computing Surveys* 2008; **40**:7:1–7:28.
13. Horn P. Autonomic Computing: IBM's Perspective on the State of Information Technology. *IBM Corporation* 2001; 1 – 39.
14. Laddaga R, Robertson P. Self Adaptive Software: A Position Paper. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, Bertinoro: Italy, 2004.
15. Jacob B, Lanyon-Hogg R, Nadgir DK, Yassin AF. *A Practical Guide to the e IBM Autonomic mic Computing Toolkit*. IBM Redbooks, http://www.redbooks.ibm.com/redbooks/pdfs/sg246635.pdf, 2004.
16. Tanenbaum AS, Van Steen M. *Distributed Systems: Principles and Paradigms* (second edition). Pearson Prentice-Hall, Upper Saddle River, NJ, USA, 2007.
17. Altekar G, Bagrak I, Burstein P, Schultz A. OPUS: Online patches and updates for security. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005; 19–19. USENIX Association.
18. Chen H, Yu J, Chen R, Zang B, Yew P-C. POLUS: A POwerful Live Updating System. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, Washington, DC, USA, 2007; 271–281. IEEE Computer Society.
19. Makris K, Ryu KD. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA, 2007; 327–340. ACM.
20. Jalili M. A hybrid Model of Dynamic Software Updating in C. Master's thesis, Computer Eng. Department, Islamic Azad University of Shabestar, 2009.
21. Appavoo J, Hui K, Soules CAN, Wisniewski RW, Silva DMD, Krieger O, Auslander MA, Edelsohn DJ, Gamsa B, Ganger GR, McKenney P, Ostrowski M, Rosenburg B, Stumm M, Xenidis J. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal* 2003; **42**(1):60–76.
22. DedaSys LLC. Programming Language Popularity. 2010. http://langpop.com/
23. Ertl A. Programming Language Popularity. 2010. http://www.complang.tuwien.ac.at/anton/comp.lang-statistics/
24. Martin T. Software Development Topics: Most Popular Programming Languages. 2010. http://www.devtopics.com/most-popular-programming-languages/
25. Gregersen AR, Jørgensen BN. Dynamic update of Java applications - balancing change flexibility vs programming transparency. *Journal of Software Maintenance and Evolution: Research and Practice* 2009; **21**(2):81–112.
26. Neamtiu I, Foster JS, Hicks M. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes* 2005; **30**(4):1–5.
27. Microsoft Co. Using the Edit and Continue Feature in C# 2.0, 2009.
28. Soules CAN, Appavoo J, Hui K, Wisniewski RW, Silva DD, Ganger GR, Krieger O, Stumm M, Auslander M, Ostrowski M, Rosenburg B, Xenidis J. System Support for Online Reconfiguration. In *USENIX 2003 Annual Technical Conference*, 2003; 141–154.
29. Potter S, Nieh J. AutoPod: Unscheduled System Updates with Zero Data Loss. In *Second International Conference on Autonomic Computing*, 2005; 367–368.
30. Arnold J, Kaashoek MF. Ksplice: Automatic rebootless kernel updates. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, 2009; 187–198, New York, NY, USA, ACM.

DOI: 10.1002/smr

31. Hashimoto M. A Method of Safety Analysis for Runtime Code Update. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, 2007; 60 – 74.
32. Buisson J, Dagnat F. Introspecting continuations in order to update active code. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008; 1–5. ACM.
33. Buisson J, Dagnat F. ReCaml: Execution state as the cornerstone of reconfigurations. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, New York, NY, USA, 2010; 27–38. ACM.
34. Duggan D. Type-based hot swapping of running modules (extended abstract). In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, 2001; 62–73. ACM.
35. Giuffrida C, Tanenbaum AS. Cooperative update: A new model for dependable live update. In *HotSWUp '09: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2009; 1 – 6. ACM.
36. Lee I. Dymos: A dynamic modification system. PhD thesis, The University of Wisconsin - Madison, 1983.
37. Gupta D, Jalote P. On line software version change using state transfer between processes. *Software - Practice and Experience* 1993; **23**(9):949–964.
38. Gupta D, Jalote P, Barua G. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering* 1996; **22**(2):120–131.
39. Hayden CM, Smith EK, Hicks M, Foster JS. State Transfer for Clear and Efficient Runtime Upgrades. In *Proceedings of the 3 rd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '11, Hannover - Germany, April 2011. IEEE Computer Society.
40. Lyu J, Kim Y, Kim Y, Lee I. A Procedure-Based Dynamic Software Update. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, Washington, DC, USA, 2001; 271 – 284. IEEE Computer Society.
41. Stoyle G, Hicks M, Bierman G, Sewell P, Neamtiu I. Mutatis Mutandis: Safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems* 2007; **29**(4):22.
42. Neamtiu IG. Practical dynamic software updating. PhD thesis, University of Maryland, College Park, 2008.
43. Subramanian S, Hicks M, McKinley KS. Dynamic software updates: A VM-centric approach. *SIGPLAN Notices* 2009; **44**(6):1–12.
44. Alpern B, Attanasio CR, Cocchi A, Lieber D, Smith S, Ngo T, Barton JJ, Hummel SF, Sheperd JC, Mergen M. Implementing jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, New York, NY, USA, 1999; 314–324. ACM.
45. Sidiroglou S, Ioannidis S, Keromytis AD. Band-aid patching. In *HotDep'07: Proceedings of the 3 rd workshop on on Hot Topics in System Dependability*, Berkeley, CA, USA, 2007; 6. USENIX Association.
46. Pressman RS. *Software Engineering: A Practitioner's Approach* (5th edn). McGraw-Hill Higher Education, New York, NY, USA, 2000.
47. Hjálmtýsson G, Gray R. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 1998; 6–6. USENIX Association.
48. Stanek J, Kothari S, Nguyen TN, Cruz-Neira C. Online Software Maintenance for Mission-Critical Systems. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Washington, DC, USA, 2006; 93–103. IEEE Computer Society.
49. Kim DK, Tilevich E. Overcoming JVM HotSwap constraints via binary rewriting. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008; 1–5. ACM.
50. Zhang S, Huang L. Type-Safe Dynamic Update Transaction. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, Washington, DC, USA, 2007; 335–340. IEEE Computer Society.
51. Orso A, Rao A, Harrold M. A Technique for Dynamic Updating of Java Software. In *Software Maintenance, IEEE International Conference on*, Los Alamitos, CA, USA, 2002; 0649. IEEE Computer Society.
52. Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon SA, Gujarathi A. Regression test selection for Java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, New York, NY, USA, 2001; 312–326. ACM.
53. Malabarba S, Pandey R, Gragg J, Barr E, Barnes JF. Runtime Support for Type-Safe Dynamic Java Classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, London, UK, 2000; 337–361. Springer-Verlag.
54. RP Bialek. Dynamic Updates of Existing Java Applications. PhD thesis, Faculty of Science, University of Copenhagen, 2006.
55. Seifzadeh H, Kermani M, Sadighi M. Dynamic Maintenance of Software Systems at Runtime. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, Washington, DC, USA, 2008; 859–865. IEEE Computer Society.
56. Gregersen AR, Simon D, Jorgensen BN. Towards a dynamic-update-enabled JVM. In *RAM-SE '09: Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, New York, NY, USA, 2009; 1–7. ACM.
57. Bierman G, Parkinson M, Noble J. UpgradeJ: Incremental Typechecking for Class Upgrades. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, Berlin, Heidelberg, 2008; 235–259. Springer-Verlag.
58. Tempero E, Bierman G, Noble J, Parkinson M. From Java to UpgradeJ: An empirical study. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008; 1–5. ACM.

DOI: 10.1002/smr

59. Parr TJ, Quong RW. ANTLR: A predicated-LL(k) parser generator. *Software - Practice and Experience* 1995; **25**:789–810.
60. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter* 2009; **11**:10–18.
61. Zhenxing Y, Zhixiang Z, Kerong B. An Approach to Dynamic Software Updating for Java. *Pacific-Asia Workshop on Computational Intelligence and Industrial Application, IEEE* 2008; **2**:930–934.
62. Zhang Z, Yao Z, Jie K. Case Study on Dynamic Evolution of Software Based on AOP. In *Information Engineering, International Conference on*, volume 2, Los Alamitos, CA, USA, 2009; 3–8. IEEE Computer Society.
63. Makris K. Whole-program dynamic software updating. PhD thesis, Arizona State University, 2009.
64. Fabry RS. How to design a system in which modules can be changed on the fly. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, 1976; 470–476. IEEE Computer Society Press.
65. Boyapati C, Liskov B, Shrira L, Moh C-H, Richman S. Lazy modular upgrades in persistent object stores. *SIGPLAN Notices* 2003; **38**(11):403–417.
66. Lin D-Y, Neamtiu I. Collateral evolution of applications and databases. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, New York, NY, USA, 2009; 31–40. ACM.
67. Bloom T, Day M. Reconfiguration in argus. In *International Workshop on Configurable Distributed Systems*, 1992.
68. Ajmani S, Liskov B, Shrira L. Modular Software Upgrades for Distributed Systems. In *LNCS ECOOP 2006: Object-Oriented Programming*, volume 4067, 2006; 452–476.
69. Srinivasan R. RPC: Remote procedure call specication version 2. RFC 1831. Network Working Group, 1995.
70. Vandewoude Y. Dynamically updating component-oriented systems. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering, K.U.Leuven, Leuven, Belgium, 2007.
71. Montgomery J. A Model for Updating Real-Time Applications. *Real-Time Systems* 2004; **27**(2):169–189.
72. Seifzadeh H, Kazem AAP, Kargahi M, Movaghar A. A Method for Dynamic Software Updating in Real-Time Systems. In *ICIS '09: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science*, Washington, DC, USA, 2009; 34–38. IEEE Computer Society.
73. Wahler M, Richter S, Oriol M. Dynamic software updates for real-time systems. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2009; 1–6. ACM.
74. Gracioli G, Fröhlich AA. An operating system infrastructure for remote code update in deeply embedded systems. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, New York, NY, USA, 2008; 31–35. ACM.
75. Augusto A, Fröhlich M. Application-Oriented Operating Systems. *GMD - Forschungszentrum Informationstechnik*, (17), 2001.
76. Noubissi AC, Iguchi-Cartigny J, Lanet J-L. Hot Updates for Java Based Smart Cards. In *Proceedings of the 3 rd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '11, Hannover - Germany, 2011. IEEE Computer Society.
77. Kramer J, Magee J. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering* 1990; **16**(11):1293–1306.
78. Liskov B, Adya A, Castro M, Ghemawat S, Gruber R, Maheshwari U, Myers AC, Day M, Shrira L. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, New York, NY, USA, 1996; 318–329. ACM.
79. Bruneton E, Lenglet R, Coupaye T. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002: Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Grenoble, France, 2002.
80. Wahler M, Richter S, Kumar S, Oriol M. Non-disruptive Large-scale Component Updates for Real-Time Controllers. In *Proceedings of the 3 rd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '11, Hannover - Germany, 2011. IEEE Computer Society.
81. Gamma E, Helm R, Johnson RE, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
82. Bazzi RA, Makris K, Nayeri P, Shen J. Dynamic software updates: The state mapping problem. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2009; 1–2. ACM.
83. Necula GC, McPeak S, Rahul SP, Weimer W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, London, UK, 2002; 213–228. Springer-Verlag.
84. Bhattacharya P, Neamtiu I. Dynamic updates for web and cloud applications. In *APLWACA '10: Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, New York, NY, USA, 2010; 21–25. ACM.
85. Neamtiu I, Hicks M. Safe and timely updates to multi-threaded programs. *SIGPLAN Notices* 2009; **44**(6):13–24.
86. Vandewoude Y, Ebraert P, Berbers Y, D'Hondt T. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering* 2007; **33**(12):856–868.
87. Gharaibeh B, Rajan H, Chang JM. A Quantitative Cost/Benefit Analysis for Dynamic Updating. Technical report, Iowa State University, 2009.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES

Habib Seifzadeh received his B.Ss. and M.Sc. in Software Engineering, both from Islamic Azad University, Najafabad Branch, in 2003 and 2006 respectively. Habib has extensive experience in programming and also manages his own IT company, where his development team produces accounting and costing software applications for large organizations such as Isfahan Municipality. Habib has over eight years of experience in teaching university-level courses, and holds a Lecturer position at Computer Engineering Department of Islamic Azad University, Najafabad Branch. Habib Seifzadeh is currently a Ph.D. candidate at Islamic Azad University, Science and Research Branch. His areas of research include programming languages, algorithms, and object-oriented analysis and design.

Hassan Abolhassani received his Ph.D. from Saitama University of Japan with a thesis on Automatic Software Design focusing on Learning from Human Designers. His areas of academic research include software automation, semantic Web researches, knowledge-based software design, and design patterns. He worked as Senior Technologist providing software based solutions for top-level clients in Japan when he was with Xist-Interactive (Razorfish Japan), until the end of September 2004, when he joined Sharif University of Technology as an assistant professor. He is now an associate professor with the computer engineering department of Sharif University of Technology.

Mohsen Sadighi Moshkenani received his B.S. in Mathematics and Statistics from Shiraz University, Shiraz, Iran, in 1973, and his M.S. in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 1977, and his Ph.D. in Computer Engineering from Indian Institute of Science (IISc) Bangalore, India, in 1991. He has over three decades of professional experience in well known universities of Iran: Shahid Beheshti University, Isfahan University of Technology, and Sharif University of Technology-International Campus at Kish Island. His research interests are knowledge engineering, semantic Web, software engineering and education. Dr. Sadighi Moshkenani is a member of ACM and Informatics Society of Iran.

DOI: 10.1002/smr