Formal methods in dynamic software updating: a survey

Razika Lounas*

LIMOSE Laboratory,
Faculty of Sciences,
University of M'hamed Bougara of Boumerdes,
Avenue de l'indépendance, 35000, Boumerdes, Algeria
Email: razika.lounas@univ-boumerdes.dz
and
Xlim Laboratory,
University of Limoges,
123 Avenue Albert Thomas, 87700, Limoges, France
*Corresponding author

Mohamed Mezghiche

LIMOSE Laboratory,
Faculty of Sciences,
University of M'hamed Bougara of Boumerdes,
Avenue de l'indépendance, 35000, Boumerdes, Algeria
Email: mohamed.mezghiche@univ-boumerdes.dz

Jean-Louis Lanet

INRIA LHS-PEC,
263 Avenue Général Leclerc, 35000, Rennes, France
Email:jean-louis.lanet@inria.fr

Abstract: Dymanic software updating (DSU) consists in updating running programs on-the-fly without any downtime that leads to systems unavailability. The use of DSU in critical applications raises several issues related to update correctness. Indeed, an erroneous dynamic update may introduce safety vulnerabilities and security breaches. In this perspective, the use of formal methods has gained a large interest since they respond to the high need of rigor required by such applications. Several frameworks were developed to first express update correctness which is based on several criteria. Then, the proposed formalisms are used to specify DSU systems, express correctness criteria and establish them. In this paper, we present a review of researches on the application of formal methods to DSU systems. We give a classification of systems according to the paradigms of programming languages and then we explain the correctness criteria and categorise the articles regarding the approaches of formalisation to establish the correctness. This information is useful to help ongoing researches in having an overview on the application of formal methods in DSU.

Keywords: dynamic software updating; DSU; formal methods; correctness criteria; critical systems; systems safety; code update; data update; update timing; semantical correctness.

Reference to this paper should be made as follows: Lounas, R., Mezghiche, M. and Lanet, J-L. (2019) 'Formal methods in dynamic software updating: a survey', *Int. J. Critical Computer-Based Systems*, Vol. 9, Nos. 1/2, pp.76–114.

Biographical notes: Razika Lounas is a final year PhD student. She has the grade of Teacher-Researcher at the Computer Science Department at the University of Boumerdes, Algeria since 2009. She is also a member of LIMOSE Research Laboratory at the same university and a member of Xlim Laboratory at the Limoges University, France. She received her Magister Diploma form the Boumerdes University in 2009. Before that, she received her Diploma of Computer Engineering at the Tizi Ouzou University, Algeria. Her main research interests are dynamic software updating, formal methods and java card applications.

Mohamed Mezghiche is a Professor in the Computer Science Department at the University of Boumerdes Algeria. He is also a Team Leader and Director of LIMOSE Laboratory at the same university. He received his PhD in Theoretical Computer Science from the University Paris 6 (France). His research interests are formal methods, program certification, theorem proving, logic and functional programming.

Jean-Louis Lanet is the Director of the High Security Labs of Inria-RBA. Previously, he was Full Professor in the Computer Science Department at the University of Limoges (2007–2014). He is also the Team Leader of the Smart Secure Device (SSD) research group. Prior to that, he was a Senior Researcher at the Gemplus Research Labs (1996–2007). During this period he spent two years at the INRIA (Sophia-Antipolis) (2003–2005) as an Engineer at the Direction des Relations Industrielles (DirDRI) and as a Senior Research Associate in the Everest team. He started his career as a researcher at the Elecma, Electronic division of the Snecma, now a part of the Safran group (1984-1995) and his field of research were on jet engine control. His research interests include security of small systems like smart cards and software engineering.

1 Introduction

1.1 Context and motivations

In order to add features or fix bugs in programs, dynamic software updating (DSU) systems allow code to be patched on-the-fly without requiring downtime. This is an important feature in critical systems that must run continuously or that can suffer attacks during their life time and must be upgraded. DSU is also referred as on-the-fly program modification, online version change, hot-swapping, or dynamic software maintenance (Seifzadeh et al., 2013).

Updating software at runtime is a challenging activity: when we think about critical systems, we are able to identify objects such as smart-phones, modern cars and personal electronic documents and application areas such as air traffic control, banking or healthcare. We are aware that in this kind of applications, any discontinuity of service to perform a classical shut down, update and restart leads to considerable losses. Besides, this kind of systems needs to be managed from a safety and security point of view, keeping their software up to date. For example, DYMOS (Lee, 1983) was implemented in banking system and EmbedDSU (Noubissi et al., 2011) is a system developed to support DSU for Java Card applications. Challenges raised by DSU are categorised into the following aspects:

- *Code update:* the essential functionality of any dynamic updating system is the ability to access new versions of program methods and classes. The challenges related to this aspect are to define techniques to identify updated parts of the code and techniques to allow the system to access the new versions of functions and classes.
- *Data update:* the second task in developing a DSU system is to transfer the state of the old program in order to make it compatible with the new version. A program state is defined, according to the style of programming, by a set of variables, stacks, objects in the heap or tables in a data base. This aspect of DSU requires the developer to write code that migrate the data structures to the new version. This code consists in state transfer functions (STFs). The challenge consists in defining techniques to allow the developer to write or infer STFs and define semantics for their application on running programs.
- *Update timing:* the timing at which the update can take place is an important decision when designing a solution for DSU. The greatest challenge of update timing is to establish a trade-off between applying updates quickly and consistently. Techniques related to this aspect are designed to detect and to calculate points where the update can be updated without violating the system safety. Such points are called safe update points (SUP) or quiescent states. This aspect requires also, in some cases, to define mechanisms to bring the system to such points.
- *Correctness:* the considerable contribution of DSU systems in term of high availability may be mitigated by errors introduced by the application of DSU. A challenging task is how to ensure that a system after being updated will behave as expected and that DSU systems will not bring any inconsistencies or system crash.

Among the challenging aspects of DSU, the correctness represents a transverse question. Indeed, in order to make sure a system is correctly updated, it is important to ensure that we obtain the expected semantics of the program, that STFs are correctly applied and that selected update points guarantee the safety of the system.

For critical systems, DSU correctness is a major concern. Indeed, erroneous updates lead systems to safety vulnerabilities and security breaches: an update may behave in an unexpected way and violates security invariants. Besides, malicious attempts to upgrade a running part of a system may have disastrous consequences. In this perspective, the use of formal methods is a means to ensure DSU correctness since they offer the rigor required by critical applications and represents an answer to several concerns related to DSU correctness. Besides, critical systems and in particular security-based systems must

pass certification procedures. The common criteria for information technology security evaluation (Common Criteria, 2016) are an international standard for the security of software and hardware products established in 1999. There are seven evaluation assurance levels (EALs) defined in common criteria, where EAL 7 is the strongest. They define the degree of rigor and depth that have been applied to assure the claimed security properties. Assessment at the two highest levels, EAL 6 and 7, requires formal methods to some extent, and gives not only the assurance that the security functions are implemented, but also that these functions are correct with respect to the security policies defined in the security target of the product. The use of formal methods in DSU offers means to specify the systems, the changes and criteria to ensure correctness with the rigor required by critical applications, security-based systems and high certification levels.

1.2 Focus and contribution

DSU systems are developed for a large variety of systems. They range from operating systems (Baumann et al., 2005; Arnold and Frans, 2009) to automation systems (Wahler and Oriol, 2014), networking (Hjálmtýsson and Gray, 1998), embedded systems (Noubissi et al., 2011) and service oriented applications (Chen and Huang, 2009). The establishment of the formal correctness in such diverse applications leads to the definition of several notions of correctness based on different criteria and related to the different aspects of DSU and to the use of various formalisms to specify DSU systems and establish correctness.

Researches in terms of formalisation in DSU are more recent than researches in developing DSU techniques based on evaluation metrics and tests. But although quite recent, many researches were done and several ideas emerged, making the crossing of formal methods and DSU a promising research area. The emergence of formalisation of DSU systems in the rich and diverse tapestry of software formalisation is reflected by the increasing number of papers about this topic. The aim of this paper is to present a comprehensive review of the literature on the use of formal methods to establish DSU correctness. Surveys related to the techniques for code and data update, update timing, comparative studies related to DSU techniques, testing and evaluation metrics are out of the scope of this paper. These concerns are discussed especially in Seifzadeh et al. (2013), Miedes and Muñoz-Escoi (2012b) and de Pina (2016). As far as we know, this survey is the first related to the application of formal methods on DSU systems. The contribution of our paper is threefold:

- We present DSU systems from diver's application area classified on the base of the programming style. The aim of this part is to provide a representative overview about DSU which forms a required background to address details related to correctness criteria.
- We give a comprehensive presentation of correctness criteria and propose to classify research papers in a way that gives a clear reading of the link between correctness criteria, style of programming and application areas.
- We propose a classification of the research papers according to the formalism used to specify DSU and correctness criteria. We outline how correctness is established by formal techniques. This contribution is meant to help ongoing researches in

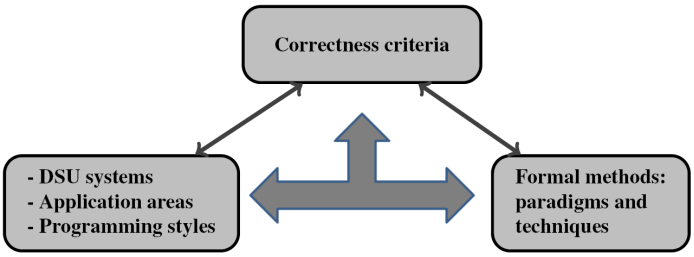
understanding existing works. We provide a practical representation in term of deciding the suitable formalisms and techniques to establish correctness criteria.

Figure 1 illustrates the elements of our contribution which ends by discussing trends in the application of formal methods in DSU systems and highlighting some interesting research directions.

1.3 Outline

This paper is organised as follows: Section 2 gives an overview of different DSU systems trough a classification according to paradigms of programming languages. Section 3 surveys properties that ensure DSU correctness. In Section 4, we present paradigms of formalisms used to specify and establish DSU correctness. In Section 5, we discuss the state of the art and point out some directions of future research. We present related works in Section 6 and then conclude in Section 7.

Figure 1 Elements of the contribution (see online version for colours)



2 Dynamic software updating systems

In this section, we give an overview about DSU systems classified according to the main paradigms of programming languages. For every presented system, its application area is outlined if provided in the research paper. The considered classification allows first to present in the same subsection technical choices to address DSU challenges in systems that belong to the same paradigm and thus raises similar concerns. Secondly, each programming language has its own mechanisms for running code, accessing, manipulating and protecting data. These concerns, combined with DSU techniques for code update, data update and update timing, are crucial to define DSU correctness criteria and formalisation issues which will be detailed in Section 3 and Section 4.

2.1 Sequential programming

Sequential programming languages have been considered for the implementation of DSU systems in Arnold and Frans (2009), Neamtiu et al. (2006), Hicks (2001), Gupta and Jalote (1993), Holmbacka et al. (2013) and Lv et al. (2012).

2.1.1 DSU for server applications

In Neamtiu et al. (2006), the authors presented Ginseng, a DSU system for C server applications. The system is composed of three parts: a compiler, a patch generator and a runtime system. First, the compiler generates an updateable program from an initial version of the program by mainly adding calls to mediator functions to access updatable structures. The patch generator computes the difference between the updatable version and the new desired version. The patch is then passed to the runtime system with information about the program state to perform the dynamic update. This system is based on function indirection for code update. Ginseng does not support changes to active methods on stack, so SUPs are identified when no changed methods are running. Data update is performed using STF generated by the patch generator. These functions are applied when the updated data is first invoked using the previously inserted mediator calls.

2.1.2 DSU for embedded systems

In Holmbacka et al. (2013), the authors developed a DSU system for C-based embedded systems. Their framework is based on FreeRTOS (FreeRTOS, 2016). The system relies on a separation of FreeRTOS tasks in the form of executable and linkable format (ELF) and a dynamic linking of ELF binaries for code update. This provides the ability to insert or to suppress binary files during execution. The programmer uses annotations to express updates and chooses update points. A state is safe to perform an update if no external events such as inter task communication or open file descriptors may disturb the state of the task. Due to the resources limits in embedded systems, the state transfer is performed by storing temporally the context in a special memory segment. This allows the new version to continue with the same addresses. When the new code is available, the state is transferred to its permanent emplacement.

In the same context, the authors in Lv et al. (2012) presented DSSUS: dynamic satellite software updating system for on-board softwares that support Vxworks systems (Vxworks, 2016). It consists on dynamically loading/unloading modules on the basis of several analysis. An ELF analysis records the size of code and data segment, the name and size of global variables and static variables. This information is used by dependencies analysis in data and modules so that software is loaded and linked according to the dependency relation. In order to perform that update, the updater inspects if tasks (current task and tasks in calling stack) are in the module to be updated to detect a SUP to ensure update safety.

2.1.3 DSU for operating systems

The system Ksplice (Arnold and Frans, 2009) is developed to dynamically update the kernel of Linux operating system. To perform updates, Ksplice works in three phases: first, a patch is generated by comparing ELF files of old and new versions, then a replacement code in which addresses and symbols are resolved is generated. The system uses then binary rewriting to insert the changes into the running kernel. Ksplice uses function indirection to jump to the new version of a method by installing trampolines at update time. The system overwrites the first few instructions of the old version's method

body with a jump instruction to the new method. Future calls to an updated method jump through this trampoline to the latest version's body. Data update is implemented with the use of the concept of shadow structures. It consists in extensions to the original structures that have the extra fields that do not fit in the original structure. The DSU system adds a pointer to a shadow structure to the end of every updatable structure in the initial version. When preparing each update, the DSU system rewrites the code that accesses the new fields to use the shadow instead.

In Giuffrida and Tanenbaum (2013), the authors present PROTEOS, a new research operating system designed to safely and automatically support many classes of live updates. The main novelty of this system is its approach to detect safe states to perform updates: updates are installed only when particular constraints, defined by the programmer, are met by the global state of the system. This is done by defining state filters: they are generic Boolean expressions written in a C-like language and evaluated at runtime. The system allows update at process level instead of the separation of code and data. This approach allows stable update process and ensures that only one version at the time is logically visible to the rest of the system and allows to perform rollback in case of run-time errors.

2.1.4 DSU for cloud applications

In Qiang et al. (2016), the authors proposed a DSU system for cloud applications written in C called multi-version execution for updating of cloud (MUC). The system uses the multi version execution approach to handle the inconsistent issue. The approach is based on four parts: first, a static analysis is performed to produce update information: an update file indicating modifications and an inter process communications files containing the calls of the application. When an update is received, the system forks a new process of the old version. Then, a synchronisation step is used to ensure that the recently created process executes the same calls as the original process. After that, it is updated when it reaches an update point by two transformations: state transformation and data transformation. The two versions are synchronised again at system calls level so that they could be seen as one process. Finally, the system ensures that when the old version executes system calls related to interprocess communication, it will copy its parameters and outputs to the address space of the new process to guarantee the consistency of the application.

2.2 Object oriented programming

Several systems are proposed for DSU in object oriented programming (Hjálmtýsson and Gray, 1998; Noubissi et al., 2011; Orso et al., 2002; Malabarba et al., 2000; Subramanian et al., 2009). The most studied languages in this category are Java and C++. DSU systems for Java applications are classified into two categories: the first one concerns the system-based on the modification of the Java Virtual Machine to support DSU functionalities and the second category represents systems based on other techniques like proxy classes to implement DSU. Jvolve (Subramanian et al., 2009), DVM (Malabarba et al., 2000) and EmbedDSU (Noubissi et al., 2011) are systems of the first category. DUSC (Orso et al., 2002) and Rubah (de Pina et al., 2014) belongs to the second category. DSU systems for C++ are presented in Baumann et al. (2005) and Hjálmtýsson and Gray (1998).

2.2.1 DSU for Java with VM support

Jvolve (Subramanian et al., 2009) is based on a virtual machine called Jikes RVM. It has a module for update preparation tool (UPT) which determines, from the old and the new version of an application, what are the modified, deleted and added classes. Instances are updated using functions written by the programmer. New classes are loaded using the standard Java class loader. The virtual machine checks if there is no updated method in the stack to perform the update. EmbedDSU (Noubissi et al., 2011) is very similar to Jvolve. It is a system developed to implement DSU functionalities in Java Card (Java Card, 2016). It is based on two parts: first, in the off-card part, a module called DIFF generator computes the syntactic changes between the old and the new version of the application and generates a patch. This patch is then sent on the card to perform the second part of the system (on-card part) by modules implemented by extending the Java Card virtual machine. Both Jvolve and EmbedDSU implement a mechanism to detect SUPs. These points restrict the update so that no stack contains restricted methods. A restricted method is an updated method or a method using instances from updated classes. The main difference between Jvolve and EmbedDSU is that EmbedDSU requires that no active method is on the stack to define safe points while Jvolve defines several kinds of restricted methods. Principally, it distinguishes between methods to update and methods that refer to updated classes. The system performs a return barrier on the first category, i.e., it blocks all calls to this kind of restricted methods. It executes an on stack replacement for the second category. This technique consists in a recompilation of methods while they are being executed.

2.2.2 DSU for Java without VM support

DUSC (Orso et al., 2002) is a DSU system for Java applications based on proxy classes. This technique rewrites first the application to make it updatable and then performs dynamic class replacement during execution. The original class is rewritten to four classes. The first is an implementation class that contains the implementation of each version of the updated class. The second and the third classes are the wrapper class and the interface class. The wrapper is used to provide the same interface to any client class of the updated class. Each implementation class refers to other updatable classes through their wrapper. The interface class is an abstract class that all implementation class extend. The wrapper class uses interface classes to refer to each implementation class indirectly. This way, all calls to a class are redirected to the current implementation. The fourth class is called state class. It encodes the state of an instance of the implementation class, and is used to transfer the state of existing instances to the new version when there is no active method in the stack.

In de Pina et al. (2014), the authors present Rubah: a DSU system for Java. It requires no changes to the JVM and works by bytecode rewriting for code update using an existing rewriting tool. This enhances the portability of the system. The main novelty of Rubah is related to data update. The authors propose two new algorithms to deal with state transfer: a parallel state transformation algorithm and a lazy state transformation algorithm. The first algorithm acts eagerly and uses the basic idea of starting from the root references, and following each object reference transitively until all the program state is visited and prepared to be transformed. The proposed algorithm performs using multiple threads and this speed up the process. The second algorithm takes place while

the program is running. The goal is to delay the transformation of each object to its first use by the new version of the application. This solution uses proxies on objects in order to inform the update process when the program uses an object (reads or writes in its fields). The system considers that update points are chosen and inserted by the programmer according to the semantics of the application.

2.2.3 DSU for C++ classes

In Hjálmtýsson and Gray (1998), the authors presented a system for DSU in connection management services in telecommunication systems. The system is based on dynamic classes to allow new functionalities to be introduced into a running C++ application. The solution is implemented using proxy classes. Each dynamic class is written as two separate parts: an abstract interface class and one or more implementation classes that inherit from the interface. An implementation class corresponds to each version of the class. Indirect method resolution is used to call the correct implementation using map that associates the class name with the current implementation version. The method to update objects is to consider that new objects are created with the new version and existing objects continue with their current versions. Once the existing objects finish their tasks they are destroyed. Specific methods allow an object to determine if its version is the most recent version and provide to the programmer the possibility to migrate explicitly objects.

2.3 Functional programming

Several works were published on dynamically update functional programs (Buisson and Dagnat, 2010; Gilmore et al., 1997; Duggan, 2005).

2.3.1 ReCaml

The system ReCaml (Buisson and Dagnat, 2010) represents a new functional programming language designed for manipulating execution states in a safe manner. The system is built on top of a simply typed lambda-calculus and is considered as an extension to the Caml byte code interpreter.

ReCaml allows active functions updating. The programmer annotates the code with SUP. When an update is detected, the execution is restored to the closest annotation to apply the update. The programmer provides functions to update program states represented on the basis of the concept of continuations: an abstract representation of program states that allows a prospective view of a program execution from a given point. The manipulation of continuation is based on the definition of a new constructor for the language. The constructor is a pattern matching operator that implements several policies for update such as waiting the end of old versions of methods and version coexistence.

2.3.2 DynamicML

In Gilmore et al. (1997), the authors presented DynamicML, a system to support Dynamic update for ML language. It extends the standard ML to support online update. DynamicML is a statically typed language. To support the compiling time type checking, DynamicML limits the changes allowed in a type. A type $S1$ can be replaced by a type $S2$

only if *S*2 is a subtype of *S*1. The system allows the update of ML modules called fonctors. The new version of a module is checked to have the same signature as the old version and updates are restricted to abstract data types. DynamicML relies on SUP searching and installation functions provided by the programmer to perform updates. The presence of installation functions is checked at compilation time. The system uses a rollback mechanism in the case of exceptions; the system is brought back to its initial state. DynamicML poses restrictions on the expressivity of updates but allows to ensure safety with concepts inherited from functional programming such as strong static typing.

2.4 Multi-threaded programming

Several studies explored the application of DSU techniques on multi-threaded programs (we cite, Chen et al., 2007; Makris, 2009; Makris and Ryu, 2007; Neamtiu and Hicks, 2009).

2.4.1 POLUS

POLUS (Chen et al., 2007) implements DSU for multi-thread C programs. Data update relies on a patch generation module to perform differences between the old and the new version of a program. The patch contains code that maintains the coherence among the threads. POLUS does not wait for SUP to perform updates; it uses an immediate update based on the coexistence of old and new versions of code. Synchronisation functions are used to maintain consistency among the different versions: when a dynamic update is being applied, the patch injector write-protects both the old and the new versions of an instance. This is performed by associating a signal handler to catch each write attempt to either version of the instance. The signal handler invokes the corresponding state synchronisation functions to transfer the modified state from one version to the other.

2.4.2 UpStare

The system UpStare (Makris, 2009) is a DSU system developed to support DSU in C multi-threaded programs. It is composed of several modules: a compiler, a patch generator and an update module. For code update, UpStare instruments the code while compiling an application by inserting information needed for the update and uses indirections in function calls. For data update, UpStare has an improved state transformation technique that is able to update active functions at run-time. The technique is called stack reconstruction meaning extracting the stack and reconstructing it to be coherent with the new version. To our knowledge, it is the unique system to allow stack reconstruction. To perform an update after detecting an update point, UpStare uses a thread coordinator to apply an atomic update to all the threads concerned by the update without causing incoherence of the system.

2.5 Component programming

Component programming was the subject of several DSU systems, like Noubissi et al. (2010b), Solarski (2004), Panzica La Manna (2011), Chen and Huang (2009), Wahler and Oriol (2014), Felser et al. (2007) and Liu and Tong (2011).

2.5.1 DSU for FASA framework

A system for dynamic software update for automation systems is presented in Wahler and Oriol (2014). The authors described a solution based on future automation system architecture (FASA), a component-based software architecture and runtime platform for real-time applications. First, the code of the new components needs to be loaded into memory. The system configuration is then prepared by creating a new system configuration, a clone of the active one. The proposed mechanisms consider the case of state transfers that requires a significant amount of time. The proposed solution allows FASA to distribute the state transfer between two components across multiple cycles. This is implemented using a synchronisation mechanism that keeps track of state changes and retransmits changed parts of the state. The system implements also a *switchover* mechanism to ensure that all updates on all involved controllers become active at the same time. Safety is ensured by a rollback mechanism to restore original configuration if new components do not behave as expected.

2.5.2 DSU for OSGi

In Noubissi et al. (2010b), the authors proposed an approach for dynamic update in OSGI inspired by their works on EmbedDSU (Noubissi, 2011; Noubissi et al., 2011). OSGi is a platform to build Java applications from a number of modular, reusable and collaborative components (called bundles), that can be dynamically reloaded (Miedes and Muñoz-Escoi, 2012a). The proposition (Noubissi et al., 2010b) is based on the analysis of bundles to prepare the update by comparing files and classes, generating a DIFF file (difference between two versions of an application) and by including STFs. The update module is structured as bundles. There is mainly two parts: the first is encapsulated in an OSGi bundle and update services and the second is in the virtual machine and offers functions of introspection, SUP detection and rollback mechanism. The state transfer is done in two parts: architecture adaptation and interface adaptation. In Chen and Huang (2009), the authors present two techniques. In the first one, the new service that has the same name with the old one is required to register in the framework before update. The new version is registered with a higher property so that it will be automatically chosen by the client. A SUP is then indicated in order to perform state transfers correctly. The second technique is based on the combination of runtime source compilation, class reloading and the proxy design pattern. Once the changes in implementation classes are detected, source code needs to be recompiled at runtime. The basic idea is to load the dynamic service class using a dedicated class loader. A proxy design pattern is used to replace the old service instances. A proxy class operates as a dynamic service class's access interface. It allows to invoke indirectly the dynamic service class by the service objects so that when the dynamic service class reloads, the service objects continues to use the same proxy instance to access new classes.

2.5.3 DSU for Distributed embedded systems

In Felser et al. (2007), the authors described an update infrastructure that allows dynamic reprogramming of the network nodes based on the binary code of the application. The update preparations are executed by a dedicated node called remote or manager node. Then, the system constructs a fine-grained modularisation of the application to detect dependencies between its elements. When the administrator changes a function, the

system identifies which functions and data in the dependency graph are affected and where these parts are installed in the network. This information represents the basis of semi-automatic policies that are used to determine if the update operation can be performed. The system creates and manages a memory image model that represents the usage of the memory in the device. This memory model is initialised with the initial state of the node and used for keeping track of the currently installed software layout. Based on this information, the system determines a location where to put modified code on the node.

2.5.4 DSU for internet of things

In Liu and Tong (2011), the authors present a framework for DSU to ensure high availability in sensor data transmission and cooperation service for internet of things. The authors suppose periodical execution of upgrades and define an update management framework to manage sensor software according to work context containing work environments and state. They set a generic update service pack between update coordinator and sensors and propose three algorithms for update service pack for improving service pack accessibility according to several scenarios given by the number of sinks and sensors implied in the update. The framework uses replication of the original version of a program which is held by an update coordinator. The replica becomes invalid after the update is performed. Each sensor holds a table containing information about updates times. This information is used in routing and synchronisation and ensures consistency of the system.

2.6 Discussion

We classified in this section DSU systems according to paradigms of programming languages through the description of representative systems from different application areas. In literature, other classifications exist (Seifzadeh et al., 2013; Miedes and Muñoz-Escoi, 2012b; Giuffrida and Tanenbaum, 2010). The classification according to paradigms of programming languages is mainly motivated by the objective of our paper which is to present a comprehensive survey on the use of formal methods to establish DSU correctness. Indeed, it is established that the style of programming language is tiled with the definition of correctness criteria. In Gupta (1994), the author studied three programming paradigms (sequential, object oriented and distributed) and showed how the definition of correctness criteria changes with the notions introduced by each paradigm. In Murarka (2010), the author presented correctness criteria related principally to the definition of SUPs and data update. They established different definitions for correctness criteria according to the style of programming (sequential and multi-threaded). Detailed definitions of DSU correctness criteria are the subject of the following section.

3 Correctness criteria for DSU

Ideally, in order to ensure DSU correctness, one wants to establish that the behaviour of the application must be the same as the one that may be obtained by starting and running the application once the updates have been applied statically. In Gupta (1994), Gupta presented the first formal framework for DSU correctness. He studied several types of

programming languages: sequential, object oriented and distributed. The author proved the undecidability of the DSU validity. This means that given two versions $P0$ and $P1$ of a program, a STF F and a state S, there are no algorithms to establish if the DSU with these parameters is valid or not. Consequently, validity of an update is ensured by a set of conditions that can be summarised by the fact that a modified procedure in the program should be a functional enhancement of the old procedure with respect to the STF. A procedure pc1 is a functional enhancement of $pc0$ if the process is in the same state in both the following cases: procedure $pc0$ is executed in state S and after that the process is updated, or, the process is updated in state S and after that procedure $pc1$ is executed.

For several authors (Murarka, 2010; Hayden et al., 2012; Zhang et al., 2012), these conditions are too restrictive. Several studies presented formalisms to model DSU systems and proposed other definitions of correctness. The next section presents more details about works to formally model, define and establish correctness for DSU systems. The application of formal methods in DSU leads generally to consider two aspects: the definition of correctness criteria and the approaches used for formalisation. In this section, we discuss correctness criteria for DSU. Formalisations techniques and paradigms will be presented in Section 4. Two categories of properties are outlined in research papers for DSU correctness. The first category regroups common properties that are shared by all updates such as type safety and no crash. The second category refers to specific properties related to the expression of the semantics of updated programs and requirements.

3.1 Correctness based on common properties

3.1.1 Reachability

In Gupta (1994) and Gupta et al. (1996), the authors proposed a formal framework for DSU modelling and define condition for its validity. The framework is based on the notion of reachability. A process is defined as a code $P0$ and a state S changing with transition function. A state S is reachable if and only if the execution of $P0$ from an initial state leads to S in a time T. When updating, the behaviour of a process is changed from $P0$ to $P1$. This change is modelled with a mapping function (or a STF). A change from $P0$ to $P1$ at a time T is valid if after the change, the process leads to a reachable state of $P1$. This means that the process behaves as if it executed $P1$ from scratch.

This validity property based on reachability suffers from the following drawback: it is both too permissive and too restrictive. It is permissive (de Pina, 2016) because the property allows a program to behave arbitrarily during a transition period after performing a DSU. However, the new program must eventually behave as if it was executed from the start. In Hayden (2012), the author exhibited an example that illustrated how this property could be restrictive: he considers an update to a server program that adds a limit to the maximum number of connected clients. Performing a DSU to install the new version on a server that already has more connections than the allowed limit raises a problem: allowing those clients to remain connected violates the validity because the clients may remain connected for an indefinite amount of time. On the other hand, terminating clients abruptly causes a loss in the program state which is not allowed in DSU. These limits are principally related to the fact that reachability is designed to target correctness in a general way. In research papers afterward, authors defined correctness criteria for targeted aspects of DSU.

3.1.2 Activeness safety

This property is related to update timing. It characterises SUPs and quiescent states by ensuring that an update may be performed only if the functions [or components in Buisson et al. (2016)] that are impacted by the update are not running (active). This means that changed functions are not on the activation stack of a running program. This is ensured by analysing the applications based on introspection of the running environment to define SUPs (Noubissi et al., 2011; Lv et al., 2012; Hayden, 2012) or by static analysis that examines the call graph of the old version.

Figure 2 An example of an updated code (see online version for colours)

```
1. main () {
2. while (c) {
3.   nbr = get (arg1);
4.   unit= get (arg2);
5.   code= get (arg3);
6.   t1=tax1(code, nbr);
7.   t2=tax2(code);
8.   p = price(nbr, unit, t1+t2);
9.   sum(p); }
10.    genbill(); }

11.  int price (int n, int u, int c) {
12.   int prc = (nbr * unit)+c;
13.   return prc; }

14. int tax1 (int c, int nbr){
15.   int p = tax(c).a*nbr;
16.   return p; }

17. int tax2 (int c){
18.    if tax(c) == null then
19.    return 0 else
20.      return 1; }
```

Old version

```
1. main () {
2. while (c) {
3.   nbr = get (arg1);
4.   unit= get (arg2);
5.   code= get (arg3);
6.   t1=tax1(code, nbr);
7.   t2=tax2(code);
8.   p = price(nbr, unit, t1+t2);
9.   sum(p); }
10.    genbill(); }

11.  float price (int n, float u, int c) {
12.   float prc = (nbr * unit)+c;
13.   return prc; }

14. int tax1 (int c, int nbr){
15.   int p = tax(c).a*nbr +tax(c).b;
16.   return p; }

17. int tax2 (int c){
18.    if tax(c) == null then
19.    return 0 else
20.      return 1; }
```

New version

Let us consider the example on (Figure 2). It illustrates the old version and the new version of a program written in a c-like style. It represents a bill generation system. The system reads first information related to the code, the unit price and the number of items of a product. Then the functions *tax*1 and *tax*2 are called to calculate the taxes for the product before calculating the final price (*price* function) which is summed at the end. If the calculus performed by function tax1 is modified in the new version (line 15 of the new version), then activeness safety property is ensured if the update is performed when the function *tax*1 is not running. Activeness safety is a very popular property (Altekar et al., 2005; Arnold and Frans, 2009; Felser et al., 2007; Noubissi et al., 2011) for the identification of update points but it suffers from some limitations: multi threaded programs and function that may contain long running loops cause the system to delay update application for a considerable time. Besides, the authors in Subramanian et al. (2009) showed that in some cases, this restriction may cause errors when a not updated function calls an updated function. If the update occurs just at the beginning of the caller and the updated method signature is updated, an error occurs.

Research papers propose techniques such as loop extraction (Neamtiu et al., 2006) in order to deal with long running loops in updated methods. The body of the loop is extracted to its own function so that if an update changes the loop body, the extracted function will be restricted for the update, and thus will ensure activeness safety, but the update can happen after it returns. Other systems use return barrier technique to inform the system every time an updated method returns and thus speeds up reaching a safe point. Some research papers proposed techniques for stack replacement to update active code (Subramanian et al., 2009; Makris, 2009). The following correctness criteria are related to the case of systems that allow active code to be updated.

3.1.3 Con-freeness

In Stoyle et al. (2007), the authors defined correctness trough the notion of con-freeness. This property expresses that the remaining code to be executed at an update point does not make use of the format of an updated data. The framework distinguishes between concrete and abstract use of a data type. The uses of the format of the data is called concrete. It is abstract otherwise. For each update point and for all types that an update changes, the program does not use these types concretely after an update point. Update points are inferred by a static analysis. Expressions of type coersion are used to identify concrete and abstract uses of data types in a program.

In the example on Figure 2, the function tax1 uses the format of the data *tax*(*c*) representing a tax on a product represented by its code (*c*). The notation *tax*(*c*).*a* expresses an access to the field a of the structure *tax*(*c*). If the programmer wants to update the format of the data *tax*(*c*).*a* (for example, add another field) con-freeness property is not satisfied at the line 6 because from this line, its format is used concretely. Con-freeness is verified at line 7, because the use of the data *tax*(*c*) from this line does not refer to its structure.

Con-freeness is less restrictive than activeness safety since it allows functions to be updated while running but the limit of these criteria is that if update points are not frequent enough, an update may be delayed for a long time.

3.1.4 Type safety

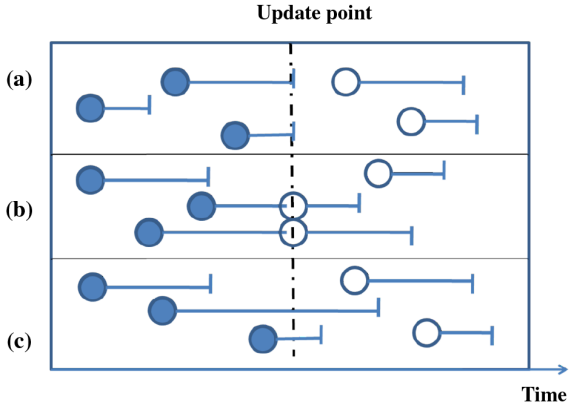
Type safety, studied in Chen et al. (2007), Buisson and Dagnat (2010) and Zhang et al. (2012), means that functions of different versions can not refer to data of an inappropriate types. This property is highly desired in DSU systems since it represents the corner stone of several applications safety such as Java-based applications.

In Zhang et al. (2012), the difference is noted with type safety in programming languages which means that a language is type safe if it prevents type errors. In DSU, if a function *f* is defined as *f*(*Aa*)… in an old program, and if the new version brings changes in the type *A*, then, after the update, when *f* is called, its argument must be of the new version of the type *A*. In the example on Figure 2, the type of the third argument in the function *price* is changed to *float*. In order to the updated code to be type safe, it must be ensured that every call to the function uses the right type of the argument.

3.1.5 Consistency

The problem of consistency is related to errors that may be introduced by the coexistence of several versions of codes or data. In the literature, updating data is performed according to different models. Figure 3 illustrates three models for data update (Hjálmtýsson and Gray, 1998) in DSU. In the first model [part (a) on Figure 3], object creation is blocked until all existing objects of older versions have expired. This solution lacks the flexibility required by DSU systems. In the second model [part (b) on Figure 3], the solution is to transform eagerly all existing objects of the old version to objects of the new version. The third model [part (c) on Figure 3] considers that new objects are created with the new version, and existing objects continue with their current version. Other research papers consider an update model where the programmer has the possibility to select objects to be transferred to the new version and objects that can continue to execute with the old version (Murarka, 2010; Malabarba et al., 2000).

Figure 3 Models for data update (see online version for colours)



These models for data update require a careful definition of conditions to avoid the manipulation of the wrong version of data by the program and thus ensure consistency. In de Pina et al. (2014), the consistency is preserved through the definition of two conditions based on the notion of *safe to access* objects. These objects are either up to date or a proxies used by the garbage collector-like update algorithm. In Boyapati et al. (2003), the authors used the notion of modularity conditions to ensure consistency. These conditions define orders to the application of STFs with regard to the updated objects and links between classes and the bodies of STFs.

Another aspect of consistency is related to code versions: in some cases, different functions needs to be managed together to avoid errors. For example in Rinderle et al. (2004), 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. In the example on Figure 2, if the value of the tax on a product changes between the two calls on lines 6 and 7, this mean that we consider different information about the tax on the same product in a single price calculus. This problem is called version inconsistency problem. The consistency is ensured generally

with techniques such as function indirections (Stoyle, 2006), synchronisation (Neamtiu and Hicks, 2009) and the definition of transaction blocks (Neamtiu et al., 2008).

3.1.6 No crash

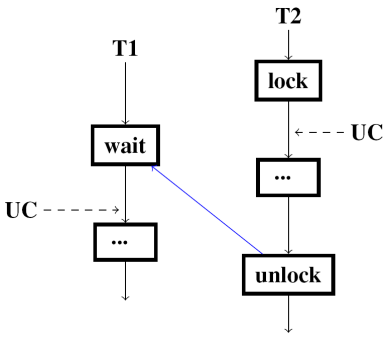
A crash occurs when a piece of software stops performing the activities it has been designed for. No crash property guarantees that any correct update must never cause target systems to crash during and after the update. Since system crash in DSU can occur for several reasons (de Pina, 2016) such as bad timing points, wrong expression of STF's or misbehaviour in updated code, this property is stated, in several research papers, on the basis of other correctness criteria such as consistency, safe update timing. Some research papers express no crash property on the basis of patches verification (Noubissi, 2011) and system verification (Zhang et al., 2012).

3.1.7 Deadlock free

Deadlock occurs when a program can never proceed. Concurrent programs sometimes include instructions for blocking to eliminate data races or communication errors. Deadlock happens when two processes sharing the same resource prevent each other from accessing the resource. In the application of DSU in concurrent programs, programmers ensure that they do not insert blocking update calls (UCs) that can lead to deadlock. Figure 4 illustrates a deadlock scenario: the thread $T2$ owns a lock on the thread $T1$. An UC is inserted just after the lock. Due to this blocking UC, the unlock instruction will not be executed. The thread $T1$ will be locked at the wait instruction and will not reach the UC point.

In Lounas et al. (2017b), the authors propose solutions to tackle deadlock situations by improving UCs to ensure that threads do not hold a lock while waiting for the update. This property is also guaranteed in Anderson and Rathke (2012) by using type analysis and in Murarka and Bellur (2008) by using analysis of parallel execution and inter procedural call graphs.

Figure 4 A deadlock scenario (see online version for colours)



3.1.8 Updatability

This property is related to update timing. In order to define SUPs, systems rely principally on two major approaches. The first approach relies on techniques that perform

SUPs before the update. Update points are inserted in the program as annotations or as procedure calls. These points are inferred by a static analysis of the program. In the second approach when an update is detected, the introspection mechanisms are activated to detect if the application is in a quiescent state to launch the update. If not, DSU systems activate mechanisms that bring the application to SUPs.

Updatability means that once an update request is made, old system must eventually reach an updatable state (Zhang et al., 2015). It is a temporal property which ensures that the application reaches update points or that the update mechanism brings the system to update points that satisfy properties defined by the user.

3.2 Correctness based on specific properties

We discuss in this subsection the use of specific properties for DSU correctness. This kind of properties, also called behavioural or semantical properties, requires writing formal specifications of the programs and the changes in the desired behaviour.

In Charlton et al. (2011) and Lounas et al. (2015), the desired properties are expressed within the updated code using Hoare Logic (HL) style (Hoare, 1969) by writing preconditions, post-conditions and assertions of the desired behaviour within the code. The system computes proof obligations that are discharged by theorem proving. In Murarka (2010), the author proposed a system for DSU that allows writing user specifications within a patch. The user writes specifications according to concepts of tasks and activities and by choosing, for particular parts of a program, behaviours defined by the framework (*offline*, *isolate*, *adapt* and *mutate*). These behaviours control the way the update is applied. For example, choosing an *offline* behaviour means that all task instances started till a specified time t execute using the old program and all task instances started after the time t execute using the new program. The user chooses also the behaviour of the program by selecting updates on tasks and activities. The system analyses the patch and computes update points that guarantee safety conditions to apply the update. In Anderson (2013), the behavioural properties are expressed in a type system extended with effects. The effects allow to express the desired properties and keep track on the evolution of the system behaviour from the initial configuration to the desired specification by defining operations to evaluate the differences between effects of the initial version and the following versions. In Zhang et al. (2013), the authors presented a framework that allows the expression of behavioural properties of dynamic updating models. The behaviour of updated systems is expressed whether by using linear temporal logic formulas (LTL) or through algebraic sorts. These notions will be detailed in the following section.

3.3 Discussion

DSU mechanisms represent a response to the growing need of high availability, but one must ensure that these updates should not cause the running system to be shut down or performing erroneous behaviour. The system should verify DSU correctness criteria to rule out system crash, avoid deadlock situations and errors in updating semantics, data and timing which expose it to vulnerabilities. For example, several applications build their protection policies on type safety. In Java applications type checks act like security protection for permissions to information. Malicious code can exploit DSU mechanisms

to bypass existing security mechanism to perform forbidden actions and access illicitly sensitive data. Introducing DSU in these applications implies to ensure well typed actions and to preserve confidentiality of information. The behaviour of the update is another example, indeed, we would like to be sure that the update does not introduce code that alter data and functions in order to corrupt the use of the system. Expressing the semantical properties of the update and the program after the update ensures the integrity of the applications. We summarise and classify in Table 1 DSU researches on correctness properties according to the style of programming language and outline the applications area where they are required. The next section presents the use of formal techniques and approaches to establish correctness criteria.

Table 1 Correctness properties in different programming paradigms and application areas

Paradigms	*Properties*	*Application area*
Sequential	• *Type safety:* Neamtiu et al. (2006); Hicks (2001); Zhang et al. (2012); Stoyle (2006); Bierman et al. (2003). • *Consistency:* Frieder and Segal (1991); Lee (1983); Zhang et al. (2012); Stoyle (2006); Hashimoto (2007); Qiang et al. (2016); Giuffrida and Tanenbaum (2013). • *Semantical correctness:* Hayden et al. (2012); Altekar et al. (2005); Hashimoto (2007). • *Reacheability:* Gupta (1994); Gupta et al. (1996); Stoyle et al. (2007). • *No crash:* Zhang et al. (2012); Stoyle (2006). • *Activeness safety:* Noubissi et al. (2011); Arnold and Frans (2009); Hayden (2012). • *Con-freeness:* Stoyle et al. (2007).	• *General:* Altekar et al. (2005); Lee (1983); Hicks (2001); Neamtiu et al. (2006); Gupta (1994); Hayden et al. (2012); Zhang et al. (2012); Gupta et al. (1996); Stoyle et al. (2007); Stoyle (2006); Hayden (2012); Bierman et al. (2003); Hashimoto (2007). • *Distributed systems:* Frieder and Segal (1991). • *Operating systems:* Arnold and Frans (2009); Giuffrida and Tanenbaum (2013). • *Cloud computing:* Qiang et al. (2016).
Object oriented	• *Type safety:* Malabarba et al. (2000); Hjálmtýsson and Gray (1998); Orso et al. (2002); Subramanian et al. (2009); Boyapati et al. (2003). • *Consistency:* Noubissi et al. (2011); Baumann et al. (2005); Hjálmtýsson and Gray (1998); Makris and Ryu (2007); Murarka and Bellur (2008); Murarka et al. (2006); de Pina et al. (2014). • *Con-freeness:* Lv et al. (2012). • *Deadlock free:* Murarka and Bellur (2008); Lounas et al. (2017b). • *Activeness safety:* Lv et al. (2012); Subramanian et al. (2009); Lounas et al. (2017b). • *Updatability:* Lounas et al. (2017b); Zhang et al. (2015).	• *General:* Malabarba et al. (2000); Subramanian et al. (2009); Murarka and Bellur (2008); Boyapati et al. (2003). • *Operating systems:* Baumann et al. (2005); Makris and Ryu (2007). • *Embedded systems:* Noubissi et al. (2011); Lv et al. (2012); Lounas et al. (2017b). • *Low level networking:* Hjálmtýsson and Gray (1998).

Table 1 Correctness properties in different programming paradigms and application areas (continued)

Paradigms	*Properties*	*Application area*
Functional	• *Type safety:* Gilmore et al. (1997); Duggan (2005); Buisson and Dagnat (2010). • *Consistency:* Gilmore et al. (1997).	• *General:* Gilmore et al. (1997); Duggan (2005); Buisson and Dagnat (2010).
Multi-threaded	• *Type safety:* Makris (2009); Neamtiu and Hicks (2009); Anderson (2013). • *Consistency:* Makris (2009); Chen et al. (2007); Neamtiu and Hicks (2009); Murarka and Bellur (2008). • *No crash:* Chen et al. (2007). • *Deadlock free:* Anderson (2013); Murarka and Bellur (2008); Anderson and Rathke (2012). • *Semantical correctness:* Anderson (2013); Anderson and Rathke (2012, 2009).	• *General:* Makris (2009); Chen et al. (2007); Neamtiu and Hicks (2009); Anderson and Rathke (2012); Anderson (2013); Murarka and Bellur (2008).
Component	• *Type safety:* Chen and Huang (2009); Chen et al. (2010). • *Consistency:* Noubissi et al. (2010); Solarski (2004); Panzica La Manna (2011); Wahler and Oriol (2014). • *Semantical correctness:* Panzica La Manna (2011); Chen et al. (2010); Wu et al. (2008). • *No crash:* Wahler and Oriol (2014). • *Activeness safety:* Felser et al. (2007).	• *General:* Chen and Huang (2009); Wu et al. (2008). • *Service oriented applications:* Chen et al. (2010). • *Distributed embedded systems:* Felser et al. (2007). • *Telecommunication systems:* Solarski (2004). • *Automation systems:* Wahler and Oriol (2014). • *Distributed systems:* Panzica La Manna (2011).

4 Approaches for formalisation in DSU

According to the level of formalisation, we distinguish four categories in the applications of formal methods in DSU. These categories fit in Rushby's classification of formal methods into four levels of rigor (Rushby, 1997) classified from lowest to the highest level of rigor:

- Level 0: this level corresponds to the absence of formal methods but formalisation can exist in development process. The validation is based on testing.
- Level 1: this level corresponds to the use of concepts and notations from logic and mathematics such as set theory and functions (Hicks, 2001; Bierman et al., 2003),

type systems (Altekar et al., 2005; Neamtiu et al., 2006; Boyapati et al., 2003) and control flow graphs (Murarka, 2010).

- Level 2: the methods of this level uses formalised specification languages with some mechanised supporting tools like type checkers or model checkers (Lee, 1983; Hayden et al., 2012; Chen et al., 2010; Stoyle, 2006; Wu et al., 2008).
- Level 3: this level represents research works that use specification language with a corresponding formal proof method. Proof methods are mechanised by a proof checker or a proof assistant (Buisson and Dagnat, 2010; Zhang et al., 2012).

Techniques from level 0 are out of the scope of this survey. These techniques have been surveyed in Seifzadeh et al. (2013) and Miedes and Muñoz-Escoi (2012b). Research works adapted formal techniques and used several formal approaches to specify different entities in DSU: the updated system, the update mechanism, the update itself and the desired properties. The notion of paradigm refers to the approach that expresses the specifications, the way that concerned elements and properties are expressed. We use the notion of technique to explain the way that the property is established.

4.1 Formal techniques in DSU

The effort to ensure DSU correctness leads to the use of a variety of techniques from different formal levels. We review in this subsection the principles of these techniques. The use of these techniques in formal approaches to establish correctness criteria is illustrated by Table 2.

4.1.1 Theorem proving

This technique (Bertot and Castéran, 2013) consists in specifying a program by the means of inductive properties satisfying verification conditions. Basically, properties of interest have the form of predicates that are proved using properties of the system and the rules and axioms of the theory used. The theorem-prover basically automates the demonstration (theorem prover) or checks the demonstration (proof checker). In some cases, due to the undecidability issue, the systems need guidance to fully discharge the proof (proof assistant).

4.1.2 Model checking

Model checking (Baier and Katoen, 2008) is a technique for the verification of finite state systems typically modelled by automata. The expected properties of the model are expressed by temporal logic formulas. Efficient symbolic algorithms are used to explore the model to verify if all possible configurations validate those properties. If a property is not verified, a counterexample is exhibited.

4.1.3 Static analysis

Static code analysis (Silva et al., 2008) is the analysis of computer software which is performed to collect some information about the behaviour of a system without executing it. This technique provides assistance and computing relevant information from a program to help developers to understand their programs and correct mistakes. There are

different forms of static analysis such as dataflow analysis, constraint-based analysis, type analysis and abstract interpretation.

4.1.4 Program annotation

Annotation (Hoare, 1969) consists in writing within the code the conditions that should be met before the annotated code is executed, as well as describing the logical state of the program after its execution. The logical formalisms underlying this approach are program logics like HL. A verification condition generator is used to produce a set of logical formulas that must be proved to ensure the correctness of the program with regard to the annotations.

4.1.5 Refinement

Refinement (Potet and Rouzaud, 1998) is the technique that synthesizes a program from a specification step by step. Each step increases the degree of precision with respect to the initial specification and introduces implementation details, such as the choice of algorithm for implementing a given function, or the choice of a concrete data type. Every step generates a number of refinement proof obligations that must be discharged to obtain a final program that has the same properties as the original specification.

4.1.6 Rewriting

Rewriting systems (Dershowitz and Jouannaud, 1990) are directed equations used to compute by repeatedly replacing sub-terms of a given formula with equal terms until the simplest possible form is obtained. This kind of transformation steps have applications in many areas such as specification and verification in software engineering, functional programming and computer algebra. In this technique, the system and its operations are represented by equations and rewriting rules and the properties are written in an equational form. The program satisfies the properties if they are deduced from the specification of the program by applying rewriting rules.

4.1.7 Bisimulation

Bisimulation (Sangiorgi, 2009) is a technique intended to characterize state equivalences and process equivalences in labelled state transition (LTS). A binary relation $R(pRq)$ over an LTS is a bisimulation if for all state p' with $p \rightarrow^u p$, it exists q' with $q \rightarrow^u q$ and (pRq) and if for all state q' with $q \rightarrow^u q$, it exists p' with $p \rightarrow^u p$ and (qRp), where u represents an action that brings the state p to p' (resp. q to q'). Bisimilarity is the union of all bisimulations. It offers a technique to prove process equivalences with two manners: state a relation R and prove that it is a bisimulation or construct R to establish the bisimulation between two processes.

4.2 Formal approaches in DSU

In order to ensure DSU correctness, research papers explored several formalisations approaches. In this subsection, we propose a classification of these works according to the paradigm of formalisation. This classification is motivated by our interest to explain

the basic ideas and concepts of formalisms and to show how they are exploited to formalise DSU and correctness criteria.

4.2.1 Algebraic formalism

This formalism relies on the modelisation of systems as many-sorted algebra consisting of a collection of data, typed operations which are specified on the basis of axioms.

In Zhang et al. (2012), the authors presented an algebraic framework for specification and verification of DSU systems based on the mechanism of POLUS (Chen et al., 2007). The main idea is to express a DSU system as a rewriting system in which one can verify properties and check incorrect DSU updates. Three distinct parts are formalised:

1 The programs in term of sort and operations.

2 The update mechanism as a rewriting system.

3 The patch that contains: added (deleted) variables and functions; states synchronisation and indirection functions.

The authors defined the following main notions for program formalisation:

- The sorts for sets of functions $Set\{F\}$ and instructions $List\{S\}$.
- The sorts for expressions (E) and programs (P).
- The operator for program constructor: Pg: $Set\{F\} * List\{S\} \rightarrow P$.
- The sorts to represent threads, thread identifiers, and stack calls.

Patches are formalised using the following sort: $Set\{F\} * Map\{F, F\} * Map\{F, V\} * Map\{V, V\} \rightarrow H$, where $Map\{F, F\}$ is a map from old versions of functions to new versions, $Map\{F, V\}$ is a mapping from functions to used variables, and $Map\{V, V\}$ is a mapping from old variables to new variables. The symbol H represents a constructed patch.

The authors define then a rewriting system on programs configurations. For every instruction or update instruction, the rewriting system formalises how the configurations change. Four states are identified for updates: *BfUpdate*, *Updating*, *Done* and *Abort*. They represent respectively states: before the update, while updating, a finished update, and an interrupted update. The following formula (1) represents the rewriting rule for update initialisation. It expresses that the program P passes from the *bfUpdate* state to *updating* by the initialisation of memory storage M with information of the patch H. C represents the threads stacks.

$$< bfUpdate, P, M, C, H > \Rightarrow < updating, P, init(M, H), C, H > \quad (1)$$

This work focuses on two types of correctness: common property (such as consistency and no crash) and correctness based on properties defined by the user. Correctness criteria are formalised as predicates on configurations. The following formula is the specification for the no crash property:

$$isAbort?(< U, P, M, C, H >) = true\ if\ U = abort,\ false\ otherwise \quad (2)$$

If the application of rewriting rules, using the system sorts and operations, leads to true then the criteria is established. The process of verification is based on three parts: choose

an initial configuration, formalise properties and then verify. The formalisation is done within the Cafeobj method (Ogata and Futatsugi, 2003). This method allows to specify systems as observational transition systems (OTS). Then, the OTS are tailored to be specified in algebraic formalism as equational theory. This representation leads to formulas that are verified using theorem proving or model checking (Zhang et al., 2013, 2014).

In Chen et al. (2010), the presented framework implements dynamic service update on a service-oriented application on top of OSGi. The process of dynamic service update is formalised using the process algebra language finite state process (FSP). The authors defined the elements of the system including the update modules as FSP processes. They identify mainly the following processes:

- *Client* and *server* processes: they represent the modules of the updated system.
- *Updatemanager* and *StateTransfert*: these processes represent modules to respectively simulate an administrator to control the update sequence and to state transfer from the old version to the new state.
- *CurrentService* (*Cs*) and *NewService* (*Ns*) to represent the implementations of the old and new version of an updated service.
- *Service interface* and *delegate*: these processes are used to respectively define the service's public methods and to coordinate interactions between the service interface and its service implementations.

Figure 5 An example for FSP formalisation

1. *Client* = (*client.send* → *client.receive* → *Client*).
2. *Server* = (*server.receive* → *server.send* → *Server*).
3. *StateT ransfer* = (*cS.getstate* → *nS.setstate* → *StateTransfer*).
4. *UpdateManager* = (*update.start* → *cS.block* → *nS.start* → *ns.run* → *cS.stop* → *update.end* → *END*.
5. *Server_interface* = (*Server*) @ {*server.receive*, *server.send*}.
6. *property ClientTrans* = (*server.receive* → *server.send* → *ClientTrans*).
7. *Check_tran* = (*Server* || *ClientT rans*).

Methods of the different components are modelled as FSP actions. Both processes and actions use FSP operators to describe different functionalities. An extract of the formalisation is shown on Figure 5. The first four lines represent the specification of these processes: *Client*, *Server*, *Updatemanager* and *StateTransfer*. For every process *P*, the expression *P.a* represents an action *a* of the process. The notation (→) in FSP represents the prefix operator: a description ($a \rightarrow P$) represents a process that initially engages in an action *a*, and then behaves exactly as described by *P*. Line 5 represents the server interface. It expresses, using the operator hide (@), that all behaviour except *send* and *receive* are hidden from the user. The authors ensure that the framework achieves deadlock freedom, type safety and behavioural correctness. The line 6 of the extract represents the property of transparency for the client. The property expresses that the client only sees the behaviour of sending messages and receiving messages from the

server. In line 7, the expression of the property is composed using the operator (||) with the server behaviour to check if it is verified by the system.

The properties are expressed using fluent LTL and verified using the model checker labelled transition system analyser (LTSA).

4.2.2 Functional formalism and type systems

In this paradigm, we analyse the use of lambda-calculus and functional modelling to reason formally on DSU systems.

The research in Bierman et al. (2003) proposed a formal framework for specification and reasoning on DSU based on a typed lambda-calculus. The authors introduce the update-calculus, an extension of lambda-calculus to support update. The framework is flexible, simple and extensible. The framework considers a program as a set of modules and an expression to evaluate. The expressions are represented by standard constructors (for instance: projection, application and *let … in*). Each module declaration has the form *module Mn = m*, where *M* is a module name, *n* is a version number, and *m* is a module body. An *update* primitive is introduced to load new versions of modules. The obtained *update-calculus* allows to update any module, including changes to types and their definitions. The calculus is based on a type system which guarantees that updates do not alter the type safety of a program.

The semantics is defined by giving a set of reduction rules to evaluate expressions in their context. Updates are correct as long as the updated program is able to apply any reduction rule after the update since the type system accepts only updates that are correct. If an expression for loading an update module *M* is to be evaluated, the system verifies if the new module does not invalidate the type safety of the program, and if the number version of the module is greater than any existing version of *M*, the new module is added then to the set of modules. The authors presented a concrete example of a server with a standard loop for getting and handling events. The update operation is represented as an event. When it occurs, the system calls a special *update handler* which applies the defined semantics to load new functionalities, to redirect to new versions of functions and to initialise data.

The authors presented in Stoyle et al. (2007) a program calculus that supports DSU on procedural, C-like languages. The framework is called Proteus. It is an extension of the update-calculus: the authors developed the possibility of inserting update points within the program to ensure con-freeness property, presented in Subsection 3.1.3. The limitations of con-freeness due to update postpones are addressed in other publications (Stoyle, 2006; Neamtiu et al., 2008): the authors propose Proteus-tx, an extension of Proteus which considers that updatable programs are structured around transactions and it proposes transactional version consistency (TVC) property for correctness. This property is used to deal with inconsistencies raised by multiple versions. They propose to distinguish functions that can be updated in mid-transaction without violating TVC and to reduce limitations related to timing. This is performed by extending a standard type system with effects to capture contextual effects of updates by expressing precisely the effect returned by the computation that has already occurred (the prior effect) and the effect of the computation that will take place (the future effect). These effects are used to ensure TVC in multi-threaded programs.

In Anderson (2013), the authors presented a framework for the use of formal methods to specify and reason about DSU in multi threaded programs. The framework is based on

a type system with effects. The idea behind this work is that the safety of an update depends on a state characterised by the code and the shared resources, which is the key difference with (Neamtiu et al., 2008) who reasoned about transactions. The considered language is a simple lambda-calculus with primitives to handle explicitly resources access and update points.

The type system ensures that the modified system will be well typed and behave as expected by keeping track of the effect of each update operation. The formalism includes a notion of *world constraints* to keep the difference between the effect of an update operation and the expected specification of an update (prior effects and modified effects). The main properties established are consistency of the update system by *subject reduction*, i.e., every reduction (except update reduction) preserves the effect and an update reduction leads to the desired effect. These principles are used by the author to establish deadlock free and type safety on concurrent programs (Anderson and Rathke, 2009) and message passing programs (Anderson and Rathke, 2012).

In Hashimoto (2007), the author presented a method to ensure behavioural safety based on the definition of a set of SUPs. The author models first safe runtime code update with a variant of high-order call-by-value language. In this language, a program is modelled as a labelled lambda-expression; and a labelled tree is then constructed for a program. The notion of code mapping is then introduced to identify point-wise correspondence between program $P0$ and its revision $P1$ which is used to extract (calculate) the modified tree nodes representing deleted, changed or inserted code in the labelled tree representing $P0$. The model precisely tracks the effect of update by defining an exact update model. This model makes use of explicit flow and dependency information which have been extracted from the labelled trees and modified tree nodes. The exact model uses information from execution traces about: the affected nodes, used values and state match operation to compute SUPs where no dependence on modified code exists and no affected values have subsequent critical uses. The model is then approximated by abstract interpretation of the semantics to derive a realistic set of SUPs. The program itself is used to obtain valid state transformers by reusing the computation of the initial program $P0$.

In Buisson et al. (2016), the authors proposed a formal framework based on Coq proof assistant, which is based on a typed lambda-calculus, to establish DSU correctness for a component model based on python language called Pycots. The formal framework consists on an abstract model called Coqcots that allows proving properties on architectures or on manipulating architectures. The two models are used in an approach that ensures the construction of correct updates of the components. First, the current architecture of the target software is extracted using Pycots platform. The result is a Coq module containing a Coqcots architecture. The designer defines the reconfiguration and build the proof of its correctness within Coq. This is based on the formal definition of primitive reconfiguration operations such as *create* to add a new component and *hotswap* to change the behaviour of an existing component. This leads to proofs related to operations and preservation of architectural constraints. A reconfiguration script is then extracted using an extension of the Coq extraction mechanism to target Python language. The script is sent to the Pytcots manager after being improved with by the programmer with glue code such as concrete Python objects. At the reception of the script, the manager applies it to the target software system.

4.2.3 State-transition paradigm

This formalism refers to the use of states and transition rules to describe systems.

In Hayden et al. (2012), the authors proposed a framework for formal verification of DSU for C-like programs. They first give an approach to write specifications categorised in three types:

1 backward-compatible specifications which are verified in both new and old version

2 post-update specifications which concern only the new version

3 conformable specifications expressing changes to conform existing features to the new version.

They define then a formal transformation of the specifications to conform them to the update.

A program is represented by a triple $< p, \sigma, e>$ where p is the code, σ is the heap and e an expression to execute. The expression e represents either standard expressions or events defined by the programmer such as *update* event. The formal semantics expresses that if an update occurs, the configuration is changed to $<p_\pi; \sigma; (e_\pi; 1)>$ where $\pi = (p_\pi, e_\pi)$ represents the patch consisting of the new program code (including unmodified functions) and an expression e_π that transforms the current heap as necessary. The integer 1 is used to indicate that an update occurred.

The verification is based on another transformation called program merging. This transformation takes a configuration of the old program and a patch and produces a configuration of a new program. The merge transformer contains transformation rules for both code and specifications written within the code. A proof that the obtained program is equivalent to the original program with the patch (a new code with a STF) is done by bisimulation. They proved that the program obtained by the merge transformation simulates every execution step in the old program with the update from the old to the new version, and that for every trace in the merged program, there is a corresponding trace in the old program with the patch. The authors used a tool (Otter) for symbolic execution and a tool for verification (Thor).

In Wu et al. (2008), the authors presented an abstract state machine (ASM)-based high level semantical model for service-oriented systems using the OSGi framework. The modelled architecture is based on the coordinator bundle agents responsible of update coordination and the functional bundle agents responsible for the tasks of the application. In order to capture the requirements of both agents, ground models are specified for both of them.

Ground models are based on the definition of states, behaviours and conditions. For instance, the states of the ground model of a coordinator bundle agent are in the set: {*Init*, *waiting for update*, *prepare updating*, *update monitoring*, *exit*}. They represent respectively the initial state; the stage of waiting updating; the stage of pre-updating with information collection about the program state; the stage at which updating is running and the coordinator bundle agent monitor the procedure and the final state.

The ground model represents an abstraction of the system. Its formalisation is based on the concepts of universes, signatures and behaviours. Universes represent the manipulated data (for example: universe of applications, universe of bundles, universe of services, universe of versions …). Signatures represent types of operations. For example, the expression *updating*: *service* → *boolean*, is used to indicate whether the service is

updating or not. The behaviours are expressed as formal rules to capture system functionalities. For example, in order to choose the right moment for updating, a formal rule is specified to express that the moment to update a bundle is when no other bundles depend on it. The specification of ground models is refined to a concrete representation. The defined rules are used in refinements to ensure the correctness of the update such as updating order, selection of updating moment, dependencies and service compatibility. The verification of the framework is performed within the model checker SMV.

In Zhang et al. (2015), the authors presented a formal framework based on state machines formalism to model dynamic software updates and use the formal model to identify SUPs. The approach determines update points that satisfy properties required by the updates on both the new and the old version of the program. The properties are expressed using temporal linear logic (LTL) and the verification is done with model checking. If counterexamples are found, the corresponding states are excluded and model checking is run again. The process is iterated until all desired properties are successfully verified.

The authors studied the case of a RaiCab system. Autonomous vehicles establish connections with a device called *controller* in order to pass a crossing. The authors propose to add a feature to the system. The desired feature states that when that vehicle approaches the crossing, it must send a message check to the controller so that it must receive two messages from the controller to enter the crossing instead of one message in the first version of the system. They studied two properties related to the behaviour of the system: *crossing property* and *passability*. The first property says that if the RaiCab is in a *no return* (noRet) section, then the gate must be closed. The second property expresses that if the gate opens, and the RaiCab did not pass yet, it must finally reach a no return section. The following LTL formulas represent respectively the properties:

$$crossing \equiv \Box(noRet \rightarrow gateClose) \tag{3}$$

$$passability \equiv gateOpen \wedge \neg passed \rightarrow \Diamond noRet \tag{4}$$

The symbols □ and ◊ are temporal operators globally (always) and eventually. To apply the proposed methodology, the authors considered that initially, all states of the systems can be considered as safe to perform the update. By applying the proposed algorithm, model checking revealed counterexamples that violate these properties. The states in which the violation occurs are removed from the set of SUPs. Finally, they obtain safe points that always satisfy both the properties. The authors studied also updatability of the system. The formalisations and verification are performed within the Maude system (Maude, 2017).

4.2.4 Axiomatic formalism

The axiomatic paradigm belongs to the category of works that aim to establish formally behavioural properties. The basis of this family is an extension of HL (Hoare, 1969).

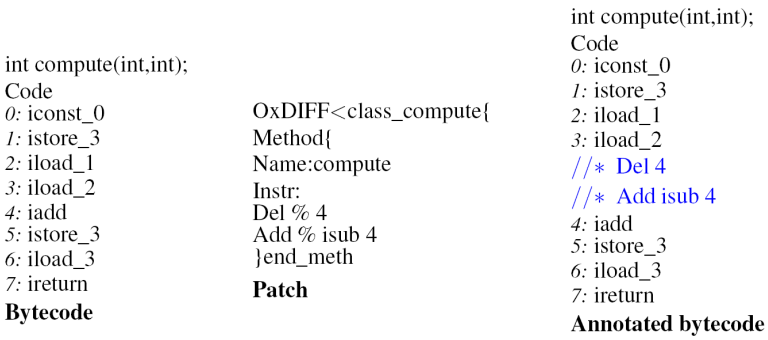
In Lounas et al. (2015), the authors described a method to establish the equivalence between specifications desired by the programmer for the new version of an updated program and specifications that are performed actually by the DSU system. The considered DSU system is dedicated to Java Card applications. The authors established code update semantical correctness at bytecode level. The specification of the update is contained in a patch obtained by a module which compares the old and the new version

of a class. The application of the updates is expressed as annotations. An annotation module produces an update-annotated program. Figure 6 illustrates an example (Lounas et al., 2015) of the application of the content of a patch to a program bytecode. The bytecode represents a function computing the sum of two integers. First, it initialises the arguments and loads them (*iload* instructions) then the sum operation is performed (*iadd*). The result is then stored. The function is supposed to be updated to perform a substraction. The patch expresses that to perform the update, the instruction at program counter (line) 4 is deleted *Del @ 4* and a substraction instruction is inserted instead *isub* at *pc 4*. The information about update are inserted as special comments on the annotated bytecode.

The authors present a predicate transformation calculus to derive the specification of the updated code [weakest precondition (WP) calculus]. WP calculus is based on Hoare triples. A program S specified by its precondition P and its post condition Q represents a triple $\{P\}S\{Q\}$. The WP calculus is used to derive precondition given a program and its post-condition. The authors extend the calculus to include update operations inserted as annotations. The obtained specification is then matched with the specification desired by the programmer. The equivalence of specification obtained by predicate transformation and the specification initially written by the programmer, using theorem proving, leads to state update correctness.

In Charlton et al. (2011), the authors proposed an extension of HL to reason about DSU on imperative languages and establish behavioural properties. The approach is based on the definition of an imperative language with features for memory allocation: the procedures are stored within the heap. An assertion language is then defined to extend HL in a way that keeps track of the code specification and memory access. Specifications are written as special comments within the updated code. The author revisited the example studied in Bierman et al. (2003). The web server code example is modelled in the imperative language and then annotated using the assertion language with specifications and properties to verify in terms of pre and post conditions. Demonstrations are performed within the tool Crowfoot.

Figure 6 An example of an annotated code (see online version for colours)



4.2.5 *Graph-based modelisation*

Researches of this category use graph-based representations of the systems, and build theories upon this modelisation to derive correctness properties.

In Murarka (2010) and Murarka and Bellur (2008), the authors present a formal framework for DSU correctness for multi threaded and object oriented programs. They built formalism upon graph-based modelisation in order to state theorems that ensure correct DSU by determining SUPs and performing update schedules.

The authors used flow graphs and inter-procedural flow graphs (IFG). The flow graphs are used to represent control among the statements of methods and activities. IFG represent the control flow in a program: the flow of control within the methods and the flow of control across the methods. These graphs are used to compute execution points and to deduce SUPs and update schedules. In order to represent the interthread dependencies in a program, the authors used a parallel execution graph (PEG). This representation uses different edges for notification and synchronisation between threads and allows to state sufficient conditions that an update must satisfy to avoid deadlock situations. After the insertion of blocking UCs, a blocking state graph (BSG) is used for checking the feasibility of updates and to compute update schedules.

The authors studied also consistency of updates. They proposed several models for data update (by applying STFs) and program activities. The consistency is based on the notion of compatibility: every objects is compatible with the old version or compatible with the new version. According to this classification, several cases are considered, for instance, an object may be created by an old version but is compatible for use by the new version (backward-state compatible objects) or an object created by the new version can be used by the old version (forward-state compatible objects). Correctness conditions are then stated for both old version and new version of the program. In order to reach a consistent updated program, every request must either ensure conditions over the old version or over the new version.

Graph-based representation is used (Murarka et al., 2006) in order to ensure the consistency related to function versions. The main idea is to isolate some methods from the process update. These methods called encapsulated methods are used to represent atomic actions. An update dependency graphs (UDG) is used to define correct update sequences that preserve the isolation and determine the order in updating classes. This same idea was used by Lee (1983) to ensure update consistency.

4.3 Trends in formalisation

We presented in this section formal techniques and formalisms used to establish correctness in DSU systems. Table 2 shows the classification of research papers according to formalisms and techniques used to establish correctness properties. The goal of Table 2 is to outline for every correctness criteria the formalism used to its specification and the way it is established.

Some facts are pointed out from the table with regard to correctness criteria. The first fact is that some formalisms are used exclusively to one criteria, for example, axiomatic formalism is used exclusively to specify semantical correctness. The table outline also that there are several ways to establish one correctness criteria. For instance, consistency is formalised in four formal paradigmes (algebraic, functional, state transition and graphs) and it is verified using two techniques: theorem proving and static analysis. These two techniques belong to different levels of rigor.

The study of formal specification and verification of DSU correctness leads us to the following observation: the application of formal methods in DSU can be categorised into

two levels: abstract or design level and code level. Code level refers to criteria such as type safety and version consistency (Stoyle et al., 2007; Neamtiu et al., 2008; Lounas et al., 2015; Hayden et al., 2012). The second category relates to the application of formal methods when designing DSU systems. Abstract level approaches (Zhang et al., 2014, 2015; Wu et al., 2008; Murarka and Bellur, 2008; An et al., 2015) ensure system properties such as deadlock free. This is performed by specifying abstract behaviours for the system. The specified properties are related for example to update operations order or interactions with the application environment (An et al., 2015). We note that formal methods may be suitable for one or both levels. For example, formalisms based on state transition paradigm with refinement techniques are suitable to the application of formal methods at the earliest stage of the development process and thus are suitable for the design level whereas annotation is suitable for the code level. Some formal approaches are used in both levels. For instance, model checking is used to design correct updates in (Zhang et al., 2015) and it is used in Lounas et al. (2017b) at code level to the same property (updatability).

Finally, Table 2 may suggest the choice of a formal method or the choice of a higher level of formalisation. For example, researches analogous to Hayden et al. (2012) and Wu et al. (2008) may use model checking to automatically establish correctness of semantical properties. Researches using functional formalism may use theorem proving to establish type safety and semantical correctness.

5 Discussion

The use of formal methods in DSU is quite recent and doubly challenging:

- *At methodology and formalisms level:* in conventional software development methods, software is designed without consideration to the possibility of dynamic updating. Although the use of formal methods during the software development process at different levels, most of applications were not developed to be dynamically updated. DSU introduces techniques and features that make necessary an early thinking about formalisation in term of documentation, prediction and constraints.
- *At the level of the correctness criteria:* DSU is used in critical applications areas. The use of formal methods is necessary to establish DSU correctness. The differences between applications area, techniques used in DSU systems and constraints related to the application area lead to several definitions of the notion of DSU correctness. The choice of the suitable formalism is impacted by the criteria that a DSU system has to ensure.

At the light of the surveyed papers, we outline some research trends about methodology and formalisation. Table 1 allows to outline the kind of properties to take into account in some applications: for example, in concurrent programming, one has to verify that the introduced dynamic update does not alter the deadlock free property. The use of DSU in object oriented programs must consider the consistency while updating the different objects created by the application. Table 1 is designed to allow the selection of correctness criteria related to an application style and area.

Table 2 Formal paradigms and techniques for correctness criteria

Property	*Formalisms used*	*Techniques*
Type safety	• *Algebraic:* Zhang et al. (2012); Chen et al. (2010). • *Functional:* Hicks (2001); Gilmore et al. (1997); Duggan (2005); Buisson and Dagnat (2010); Anderson (2013); Bierman et al. (2003); Stoyle (2006); Anderson and Rathke (2012). • *Type system:* Malabarba et al. (2000); Boyapati et al. (2003); Neamtu et al. (2006); Duggan (2005); Stoyle (2006); Bierman et al. (2003); Chen and Huang (2009).	• *Static updatability analysis:* Neamtu et al. (2006); Stoyle (2006). • *Session typing analysis:* Anderson (2013); Anderson and Rathke (2012). • *Theorem proving:* Buisson and Dagnat (2010). • *Static analysis with type checker:* Gilmore et al. (1997). • *Rewriting:* Zhang et al. (2012). • *Model checking:* Chen et al. (2010).
Consistency	• *Algebraic:* Zhang et al. (2012). • *State transition:* Panzica La Manna (2011). • *Functional:* Hicks (2001); Gilmore et al. (1997); Stoyle (2006); Hashimoto (2007); Buisson et al. (2016); Neamtiu et al. (2008). • *Graph dependencies:* Lee (1983).	• *Rewriting:* Zhang et al. (2012). • *Static analysis with consistency checker:* Lee (1983); Gilmore et al. (1997). • *Updatability analysis:* Stoyle (2006). • *Static analysis:* Murarka and Bellur (2008); Murarka et al. (2006). • *Abstract interpretation:* Hashimoto (2007). • *Theorem proving:* Buisson et al. (2016).
No crash	• *Algebraic:* Zhang et al. (2012). • *Functional:* Stoyle (2006).	• *Rewriting:* Zhang et al. (2012).
Semantical correctness	• *Algebraic:* Zhang et al. (2012, 2013, 2014); Chen et al. (2010). • *Functional:* Hashimoto (2007); Hayden (2012). • *Axiomatic:* Charlton et al. (2011). • *State transition:* Hayden et al. (2012); Altekar et al. (2005); Panzica La Manna (2011); Wu et al. (2008); Zhang et al. (2015). • *Type and effect systems:* Anderson and Rathke (2009).	• *Bisimulation:* Hayden et al. (2012); Hayden (2012). • *Program annotation:* Charlton et al. (2011); Hayden (2012). • *Static analysis:* Altekar et al. (2005). • *Refinement:* Wu et al. (2008). • *Abstract interpretation:* Hashimoto (2007). • *Model checking:* Zhang et al. (2013, 2014, 2015); Chen et al. (2010).
Con-freeness	• *Functional:* Stoyle et al. (2007).	• *Static updatability analysis:* Stoyle et al. (2007); Lv et al. (2012).
Reachability	• *State transition:* Gupta (1994); Gupta and Jalote (1993); Gupta et al. (1996).	• *Static analysis:* Gupta (1994); Gupta and Jalote (1993); Gupta et al. (1996).
Deadlock free	• *Graph-based modelisation:* Murarka (2010); Murarka and Bellur (2008). • *Functional:* Anderson (2013); Anderson and Rathke (2012).	• *Session typing analysis:* Anderson (2013); Anderson and Rathke (2012). • *Static analysis:* Murarka (2010); Murarka and Bellur (2008).
Activeness safety	• *State transition:* Hayden (2012). • *Functional:* Buisson et al. (2016).	• *Dependencies analyser:* Lv et al. (2012). • *Static analysis:* Lv et al. (2012). • *Theorem proving:* Buisson et al. (2016).
Updatability	• *State transition:* Zhang et al. (2015).	• *Model checking:* Zhang et al. (2015).

In Table 2, we highlighted the formalisms and techniques used to establish correctness. This table allows to establish links between approaches of formalisation and correctness criteria and thus help to decide which is the most suitable approach for some correctness criteria. It appears that interesting extensions may be explored. For example, in Anderson (2013), type systems are improved with effects to capture specifications for multi-threaded C programs. We think that such extension will help formal reasoning on DSU for other types of languages (object oriented for example). Another possible extension concerns the behavioural properties that use principally HL and state transition formalism. Extensions of HL are defined in formal reasoning for DSU in sequential programs, analogously the idea may be used to establish DSU correctness in some paradigms like object oriented or functional programming, in addition to the use of an algebraic formalism to reason about DSU in object oriented programs as extension to Zhang et al. (2012).

Another observed point concerns formalisation. Two categories are outlined considering whether the formalisation is performed after the full development of the system or before and during its development. In some papers, the process of formalisation is done after the development of the system (Noubissi et al., 2011; Chen et al., 2007; Zhang et al., 2012) whereas in other papers, this formal study is done before or during the process of programming (Murarka, 2010; Hayden et al., 2012; Anderson, 2013; Stoyle, 2006). In this trend, research papers pointed recently the importance to build update aware applications (Giuffrida and Tanenbaum, 2009; Giuffrida et al., 2017), the authors used the expression live update-friendly systems to describe this idea. In An et al. (2015), Zhang et al. (2014, 2015) and Wu et al. (2008), the authors used formal methods at design level. The interaction between this approach for DSU formalisation and the promising idea of building dynamic update aware software may foster research with regard to this trend. In ANSSI (2015), the French Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) the authors has proposed a dedicated process allowing to certify a product that can be dynamically changed, certifying the update code and the loader. It defines the concepts and the methodology applicable to the evaluation of a product embedding a code loading mechanism and the usage of this loader as part of the assurance continuity process.

The surveyed researches raise the following issues in formalisation: some DSU systems are implemented by transforming the initial program to be updatable (Stoyle et al., 2007; Giuffrida and Tanenbaum, 2009; Orso et al., 2002). This is performed for example in Orso et al. (2002) with proxy classes. The obtained updatable program is supposed to be equivalent to the initial version. We believe that this equivalence have to be verified using formal methods in order to avoid erroneous behaviours in such program transformations. Another issue is related to the application area of updated systems. As outlined, the application of formal methods for DSU correctness is a challenging task. This is even more accentuated with resource limited applications such as embedded systems. Recently, in Lounas et al. (2017a), the authors pointed out the difficulty to embed formal verification for semantical correctness in a DSU system for Java Card applications.

Finally, research trends concerning DSU correctness outlined that common properties have attracted more research that semantical correctness of programs. The use of DSU in critical applications and their need for certification in security issues suggest that both properties should be considered. A dynamic update for a critical system must be verified to be type safe but this is not enough, indeed, we must ensure that the updated program

behaves as expected and does not introduce malicious behaviours. We notice through the surveyed articles that some applications of DSU system in critical domains do not use formal methods but project it in future works. Information collected in this survey is useful to such a perspective to formally ensure DSU correctness.

6 Related works

Researches about DSU were surveyed in several papers according to different points of view. In Miedes and Muñoz-Escoi (2012b), the authors presented a chronological survey of DSU systems. This survey explores the goals of DSU, its techniques and application in several domains with a focus on distributed programs. In Seifzadeh et al. (2013), the authors presented a taxonomy of DSU systems according to several views such as evaluation metrics (e.g., supported changes and predictability of the updates) and reviews based on the presentation of DSU techniques like state transfer techniques and choice of the time of update. In Giuffrida and Tanenbaum (2010), the authors presented a classification according to the nature of the update whether it concerns changes in code, data or resources like memory. In Miedes and Muñoz-Escoi (2012b), the authors presented a survey that focuses on goals and requirements of DSU and the presentation of its principle techniques.

In de Pina (2016), the author presents a considerable survey to compare DSU systems with regard mainly to used techniques, flexibility and efficiency. The notion of correctness in this work is based on the development of a testing framework for DSU. He also presented several classifications for DSU systems. For example, the document compare systems whether they are transformed to support update or no. They are also compared to point out if they respond to all DSU challenges to point out systems that do not implement some features such as Dmitriev (2001). Another presented comparison is related to either systems adopts patch generation approaches to prepare updates or relies on whole program update approaches.

Existing surveys helped us in the assimilation of different mechanisms and techniques, but there is an evident lack in response to the increasing need to establish formal correctness of DSU. Our survey complements the existing surveys by bringing new points of view. We proposed a classification according to the style of the programming languages. This classification includes more styles than the one presented in Seifzadeh et al. (2013) (procedural, object oriented and functional).We were aware to present systems from diverse applications areas. We detailed correctness properties and did more than a recapitulation. Our survey explains properties in order to choose suitable formal methods to establish them. The novelty of our survey is the categorisation of the use of formal methods to establish DSU correctness.

7 Conclusions

DSU systems are increasingly gaining interest as the most promising solution to software evolution and high availability systems. DSU solutions are developed for systems from different application areas and programming styles.

Applying DSU raises correctness issues because an update can introduce errors that corrupt systems functionalities. In this paper, we presented a state of the art related to the application of formal methods to DSU correctness. We outlined that DSU is a critical feature in applications requiring high availability and that the application of formal methods strengthen considerably safety and security of DSU systems. We began by presenting the major challenges in DSU and pointed out the importance of correctness. Then, we gave a classification of DSU systems according to programming paradigms. We studied the different notions of DSU correctness and studied the use of formal methods to establish them.

As far as we know, this is the first survey about the application of formal methods in DSU. We presented a comprehensive review of DSU correctness criteria and proposed a classification of formalisation approaches. The classification shows which formal methods have been used in DSU and how they were used to deal with correctness criteria. This contribution is thought to assist ongoing researches and help in selecting the appropriate approach to formally establish DSU correctness. We identified on the base of the surveyed articles, some open research directions by outlining relations between formalisms, properties and DSU development. Formalisation in DSU is a quite recent research field. The application of DSU in several application areas and the richness of formalisation approaches lead to papers from different scientific background and thus to interesting potential scientific collaborations. We believe that this contribution provides a reference in comprehension and investigating the application of formal methods to DSU.

References

Altekar, G., Bagrak, I., Burstein, P. and Schultz, A. (2005) 'OPUS: online patches and updates for security', *Proceedings of the 14th Conference on USENIX Security Symposium*, USENIX Association, Vol. 14.

An, S., Ma, X., Cao, C., Yu, P. and Xu, C. (2015) 'An event-based formal framework for dynamic software update', *IEEE International Conference on Software Quality*, *Reliability and Security*, QRS, Vancouver, BC, Canada.

Anderson, A. and Rathke, J. (2009) 'Migrating protocols in multi-threaded message-passing systems', *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades* (*HotSWUp*).

Anderson, G. (2013) *Behavioural Properties and Dynamic Software Update for Concurrent Programs*, PhD thesis, University of Southampton.

Anderson, G. and Rathke, J. (2012) 'Dynamic software update for message passing programs', *Programming Languages and Systems*, Springer Berlin Heidelberg, pp.207–222.

ANSSI (Secrétariat Général de la Défense et de la Sécurité Nationale Security) (2015) *Requirements for Post-Delivery Code Loading*, National Agency of Information Systems Security, Paris, France.

Arnold, J. and Frans, K.M. (2009) 'Ksplice: automatic rebootless kernel updates', *Proceedings of the 4th ACM European Conference on Computer Systems* (*EuroSys*), pp.187–198.

Baier, C. and Katoen, J.P. (2008) *Principles of Model Checking* (*Representation and Mind Series*), The MIT Press, Cambridge, Massachusetts; London, England.

Baumann, A., Kerr, J., Da Silva, D., Krieger, O. and Wisniewski, R.W. (2005) 'Module hot-swapping for dynamic update and reconfiguration in K42', in *6th Linux Conf. Au*.

Bertot, Y. and Castéran, P. (2013) *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Springer Science & Business Media.

Bierman, G., Hicks, M., Sewell, P. and Stoyle, G. (2003) 'Formalizing dynamic software updating', *On-line Proceedings of the Second International Workshop on Unanticipated Software Evolution* (*USE*).

Boyapati, C., Liskov, B., Shrira, L., Moh, C-H. and Richman, S. (2003) 'Lazy modular upgrades in persistent object stores', *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing*, *Systems*, *Languages*, *and Applications*, ACM publisher.

Buisson, J. and Dagnat, F. (2010) 'ReCaml: Execution state as the cornerstone of reconfigurations', *SIGPLAN Not*., ACM, New York, pp.27–38.

Buisson, J., Dagnat, F., Leroux, E. and Martinez, S. (2016) 'Safe reconfiguration of Coqcots and Pycots components', *Journal of Systems and Software*, Vol. 122, No. C, pp.430–444..

Charlton, N., Horsfall, B. and Reus, B. (2011) 'Formal reasoning about runtime code update', in *ICDE Workshops*, pp.134–138.

Chen, H., Yu, J., Chen, R., Zang, B. and Yew, P-C. (2007) 'POLUS: a powerful live updating system', *Proceedings of the 29th International Conference on Software Engineering* (*ICSE*), pp.271–281.

Chen, J. and Huang, L. (2009) 'Dynamic service update based on OSGi', *Software Engineering*, (*WCSE*), *WRI World Congress*, pp.493–497.

Chen, J., Huang, L., Du, S. and Zhou, W. (2010) 'A formal model for supporting frameworks of dynamic service update based on OSGi', in *17th Asia Pacific Software Engineering Conference* (*APSEC*), pp.234–241.

Common Criteria (2016) [online] http://www.commmoncriteria.org (accessed 15th October 2017).

de Pina, L., Veiga, L. and Hicks, M. (2014) 'Rubah: DSU for java on a stock JVM', *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, *OOPSLA'14*, ACM New York, NY, USA, October, Vol. 49, No. 10, pp.103–119.

de Pina, L.G.G. (2016) *Practical Dynamic Software Updating*, PhD thesis, University of Lisbon, Portugal.

Dershowitz, N. and Jouannaud, J.P. (1990) 'Rewrite systems', *Handbook of Theoretical Computer Science*, Vol. B, pp.243–320, MIT Press, Cambridge, MA, USA.

Dmitriev, M. (2001) 'Towards flexible and safe technology for runtime evolution of java language applications', in *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*.

Duggan, D. (2005) 'Type-based hot swapping of running modules', *Acta Inf*., Vol. 41, No. 4, pp.181–220, Springer-Verlag New York, Inc.

Felser, M., Kapitza, R., Kleinöder, J. and Schröder-Preikschat, W. (2007) 'Dynamic software update of resource-constrained distributed embedded systems', *Embedded System Design: Topics*, *Techniques and Trends*, pp.387–400, Springer US.

FreeRTOS (2016) [online] http://www.freertos.org/ (accessed 15th October 2017).

Frieder, O. and Segal, M.E. (1991) 'On dynamically updating a computer program: from concept to prototype', *J. Syst. Softw*., Vol. 14, No. 2, pp.111–128, Elsevier Science Inc.

Gilmore, S., Kéirlí, D. and Walton, C. (1997) *Dynamic ML without Dynamic Types*, Technical Report ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, University of Edinburgh.

Giuffrida, C. and Tanenbaum, A.S. (2009) 'Cooperative update: a new model for dependable live update', *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, ACM Publisher, pp.1:1–1:6.

Giuffrida, C. and Tanenbaum, A.S. (2010) 'A taxonomy of live updates', in *Advanced School for Computing and Imaging* (*ASCI*) *Conference*.

Giuffrida, C., Iorgulescu, C., Tamburrelli, G. and Tanenbaum, A.S. (2017) 'Automating live update for generic server programs', *IEEE Transactions on Software Engineering*, Vol. 43, No. 3, IEEE Press, Piscataway, NJ, USA.

Giuffrida, K.A and Tanenbaum, A.S. (2013) 'Safe and automatic live update for operating systems', in *Architectural Support for Programming Languages and Operating Systems*, *ASPLOS'13*, Houston, TX, USA.

Gupta, D. (1994) *On-line Software Version Change*, PhD thesis, Indian Institute of Technology, India.

Gupta, D. and Jalote, P. (1993) 'On line software version change using state transfer between processes', *Softw. Pract. Exper*., Vol. 23, No. 9, pp.949–964, John Wiley & Sons, Inc.

Gupta, D., Jalote, P. and Barua, G. (1996) 'A formal framework for on-line software version change', *IEEE Trans. Softw. Eng*., IEEE Press, pp.120–131.

Hashimoto, M. (2007) 'A method of safety analysis for runtime code update', *Advances in Computer Science – ASIAN 2006*, *Secure Software and Related Issues*, Springer Berlin Heidelberg, pp.60–74.

Hayden, C.M. (2012) *Clear*, *Corret and Efficient Dynamic Software Updates*, PhD thesis, University of Maryland, USA.

Hayden, C.M., Magill, S., Hicks, M., Foster, N. and Foster, J.S. (2012) 'Specifying and verifying the correctness of dynamic software updates', *Proceedings of the 4th International Conference on International Conference on Verified Software: Theories*, *Tools*, *Experiments* (*VSTTE*).

Hicks, M. (2001) *Dynamic Software Updating*, PhD thesis, Department of Computer and Information Science, University of Pennsylvania, USA.

Hjálmtýsson, G. and Gray, R. (1998) 'Dynamic C++ classes: a lightweight mechanism to update code in a running program', *Proceedings of the Annual Conference on USENIX Annual Technical Conference*.

Hoare, C.A.R. (1969) 'An axiomatic basis for computer programming', in *Commun*. *ACM*, Vol. 12, No. 10, pp.576–580, ACM New York, NY, USA.

Holmbacka, S., Lund, W., Lafond, S. and Lilius, J. (2013) 'Lightweight framework for runtime updating of C-based software in embedded systems', *Proceedings of 5th Workshop on Hot Topics in Software Upgrades*, USENIX Publisher.

Java Card (2016) [online] http://www.oracle.com/technetwork/java/javacard/ (accessed 15th October 2017).

Lee, I. (1983) *Dymos: A Dynamic Modification System*, PhD thesis, University of Wisconsin, Madison, USA.

Liu, J. and Tong, W. (2011) 'A framework for dynamic updating of service pack in the internet of things', *Internet of Things* (*iThings*/*CPSCom*), *International Conference on and 4th International Conference on Cyber*, *Physical and Social Computing*, pp.33–42.

Lounas, R., Jafri, N., Legay, A., Mezghiche, M. and Lanet, J.L. (2017b) 'A formal verification of safe update point detection in dynamic software updating', in Cuppens, F., Cuppens, N., Lanet, J.L. and Legay, A. (Eds.): *Risks and Security of Internet and Systems*, *CRiSIS 2016*, *Lecture Notes in Computer Science*, Vol. 10158, Springer, Cham.

Lounas, R., Mezghiche, M. and Lanet, J.L. (2015) 'An approach for formal verification of updated java bytecode programs', *Proceedings of the 9th Workshop on Verification and Evaluation of Computer and Communication Systems*, pp.51–64.

Lounas, R., Mezghiche, M. and Lanet, J.L. (2017a) 'A formal verification of dynamic updating in a java-based embedded system', *International Journal of Critical Computer-Based Systems* (in press).

Lv, W., Zuo, X. and Wang, L. (2012) 'Dynamic software updating for onboard software', *Proceedings of Second International Conference on Intelligent System Design and Engineering Application*, pp.251–253.

Makris, K. (2009) *Whole Program Dynamic Software Updating*, PhD thesis, Arizona State University, USA.

Makris, K. and Ryu, K.D. (2007) 'Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels', *SIGOPS Oper. Syst. Rev*., Vol. 41, No. 3, pp.327–340, ACM New York, NY, USA.

Malabarba, S., Pandey, R., Gragg, J., Barr, E. and Fritz Barnes, J. (2000) 'Runtime support for type-safe dynamic java classes', in *Proc. ECOOP 2000 Object-Oriented Programming*, Springer Berlin Heidelberg.

Maude (2017) [online] http://maude.cs.illinois.edu (accessed 15th October 2017).

Miedes, E. and Muñoz-Escoi, F.D. (2012a) *Dynamic Software Update*, University of Valencia, Spain, Technical Report, ITI-SIDI-2012/004.

Miedes, E. and Muñoz-Escoi, F.D. (2012b) 'A survey about dynamic software updating', Technical Report ITI-SIDI-2012/003, University of Valencia, Spain.

Murarka, Y. (2010) *Online Update of Concurrent Object Oriented Programs*, PhD thesis, Indian Institute of Technology, India.

Murarka, Y. and Bellur, U. (2008) 'Correctness of request executions in online updates of concurrent object oriented programs', *Proceedings of 15th Asia-Pacific of Software Engineering Conference*, pp.93–100.

Murarka, Y., Bellur, U. and Joshi, R.K. (2006) 'Safety analysis for dynamic update of object oriented programs', *Proceedings of 13th Asia Pacific Software Engineering Conference*, pp.225–232.

Neamtiu, I. and Hicks, M. (2009) 'Safe and timely updates to multi-threaded programs', *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM New York, pp.13–24.

Neamtiu, I., Hicks, M., Foster, J.S. and Pratikakis, P. (2008) 'Contextual effects for version-consistent dynamic software updating and safe concurrent programming', Proce*edings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Publisher, pp.37–49.

Neamtiu, I., Hicks, M., Stoyle, G. and Oriol, M. (2006) 'Practical dynamic software updating for C', *Conference on Programming Language Design and Implementation*, (*PLDI*), *Proceedings of the 27th ACM SIGPLAN*, pp.72–83.

Noubissi, A.C. (2011) *Mise á Jour Dynamique et Sécurisée de Composants Systéme dans une Carte á Puce*, PhD thesis, University of Limoges, France.

Noubissi, A.C., Iguchi-Cartigny, J. and Lanet, J.L. (2010a) 'Incremental dynamic update for java-based smart cards', in *Proc. Fifth International Conference on Systems*, pp.110–113.

Noubissi, A.C., Iguchi-Cartigny, J. and Lanet, J.L. (2011) 'Hot Updates for java based smart cards', in *ICDE Workshops*, pp.168–173.

Noubissi, A.C., Iguchi-Cartigny, J. and Lanet, J-L. (2010b) 'Convergence OSGI JAVACARD: fine grained dynamic update', *Eurosmart Smart Card Security Conference and Java Card*, (*e-Smart*), Sophia Antipolis France.

Ogata, K. and Futatsugi, K. (2003) 'Proof scores in the OTS/CafeOBJ method', *Formal Methods for Open Object-Based Distributed Systems*, Springer Berlin Heidelberg, pp.170–184.

Orso, A., Rao, A. and Harrold, M.J. (2002) 'A technique for dynamic updating of java software', *Proceedings of the International Conference on Software Maintenance*, pp.649–658.

Panzica La Manna, V. (2011) 'Dynamic software update for component-based distributed systems', *Proceedings of the 16th International Workshop on Component-Oriented Programming*, ACM New York, pp.1–8.

Potet, M.L. and Rouzaud, Y. (1998) 'Composition and refinement in the B-method', *B'98: Recent Advances in the Development and Use of the B Method*, Springer Berlin Heidelberg, pp.46–65.

Qiang, W., Chena, F., Laurence, T. and Jin, H. (2016) 'MUC: updating cloud applications dynamically via multi-version execution', *Future Generation Computer Systems*, Vol. 74, No. C, pp. 254–264.

Rinderle, S., Reichert, M. and Dadam, P. (2004) 'Correctness criteria for dynamic changes in workflow systems: a survey', *Data Knowl. Eng*., Vol. 50, No. 1, pp.9–34, Elsevier Science Publishers B.V., Amsterdam.

Rushby, J. (1997) 'Formal methods and their role in the certification of critical systems', *Safety and Reliability of Software Based Systems*, Springer London, pp.1–42.

Sangiorgi, D. (2009) 'On the origins of bisimulation and coinduction', *ACM Transactions on Programming Languages and Systems*, ACM Publisher.

Seifzadeh, H., Abolhassani, H. and Moshkenani, M.S. (2013) 'A survey of dynamic software updating', *Journal of Software: Evolution and Process*, Vol. 25, No. 5, pp.2047–7481.

Silva, V.D., Kroening, D. and Weissenbacher, G. (2008) 'A survey of automated techniques for formal software verification', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE Publisher, pp.1165–1178.

Solarski, M. (2004) *Dynamic Upgrade of Distributed Software Components*, PhD thesis, University of Berlin, Germany.

Stoyle, G. (2006) *A Theory of Dynamic Software Updates*, PhD thesis, University of Cambridge, UK.

Stoyle, G., Hicks, M., Bierman, G., Sewell, P. and Neamtiu, I. (2007) 'Mutatis mutandis: safe and predictable dynamic software updating', *ACM Trans. Program. Lang. Syst*.

Subramanian, S., Hicks, M. and McKinley, K.S. (2009) 'Dynamic software updates: a VM centric approach', *Proceedings of the 30th ACMSIGPLAN Conference on Programming Language Design and Implementation*, ACM, pp.1–12.

Vxworks (2016) [online] http://windriver.com/products/vxworks/ (accessed 15th October 2017).

Wahler, M. and Oriol, M. (2014) 'Disruption-free software updates in automation systems', *Emerging Technology and Factory Automation* (*ETFA*), IEEE, pp.1–8.

Wu, J., Huang, L. and Wang, D. (2008) 'ASM-based model of dynamic service update in OSGi', *SIGSOFT Softw. Eng. Notes*, Vol. 33, No. 2, pp.1–8, ACM Publisher.

Zhang, M., Ogata, K. and Futatsugi, K. (2012) 'An algebraic approach to formal analysis of dynamic software updating mechanisms', *Asia Pacific Software Engineering Conference* (*APSEC*), pp.664–673.

Zhang, M., Ogata, K. and Futatsugi, K. (2013) 'Formalization and verification of behavioral correctness of dynamic software updates', *Electr. Notes Theor. Comput. Sci*., Vol. 294, *Proceedings of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop*, pp.12–23.

Zhang, M., Ogata, K. and Futatsugi, K. (2014) 'Verifying the design of dynamic software updating in the OTS/CafeOBJ method', *Specification*, *Algebra*, *and Software – Essays Dedicated to Kokichi Futatsugi*, pp.560–577.

Zhang, M., Ogata, K. and Futatsugi, K. (2015) 'Towards a Formal approach to modeling and verifying the design of dynamic software updates', *Asia-Pacific Software Engineering Conference* (*APSEC*), pp.159–166.