Testing Run-time Evolving Systems

Tudor Dumitraş,[1] Frank Eliassen,[2] Kurt Geihs,[3]
Henry Muccini,[4] Andrea Polini,[5] Theo Ungerer[6]

[1] Carnegie Mellon University, Electrical & Computer Engineering Department
5000 Forbes Ave., Pittsburgh, PA 15217, USA
tudor@cmu.edu
[2] University of Oslo, Department of Informatics
N-0316, Oslo, 1080 Blindern, Norway
frank@ifi.uio.no
[3] Universität Kassel, Verteilte Systeme
FB 16, Wilhelmshöher Allee 73, D-34121 Kassel, Germany
geihs@uni-kassel.de
[4] University of L'Aquila, Department of Computer Science
67100 L'Aquila, Via Vetoio, 1, Italy
henry.muccini@di.univaq.it
[5] University of Camerino, Department of Mathematics and Computer Science
62032, Camerino, Via Madonna delle Carceri,9 – Italy
andrea.polini@unicam.it
[6] University of Augsburg, Institute of Computer Science
86159 Augsburg, Germany
ungerer@informatik.uni-augsburg.de

Abstract. Computer systems undergoing runtime evolution, such as online software-upgrades or architectural reconfigurations, must cope with changes that happen during the system execution and that might be unpredictable at design-time. The evolution requires a sequence of transition phases, where the system enters configurations emerging at runtime that could not have been validated in advance. Reasoning about such emerging behavior is difficult because previously-established system invariants do not hold, changes are implemented by both human and software agents, and externally-imposed deadlines might determine the success of the evolution. The workgroup has discussed three concrete scenarios for runtime evolution and has identified a set of challenges that are not adequately addressed by current testing approaches.

Keywords. Software Testing, Dynamically Evolving Systems

1 Introduction

Software evolution is an integral part in the life cycle of a computer system. Evolution refers to the possibility of changing or updating system characteristics in order to preserve their usefulness. As discussed in [1], and depending on the motivation leading to the change, we can generally distinguish among three different types of changes:

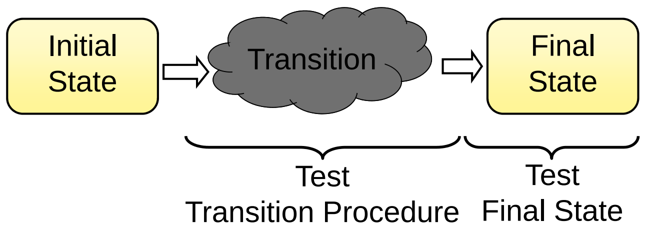


Fig. 1. Testing evolving systems.

- *Perfective*: the modification is required to satisfy new emerging requirements;
- *Corrective*: the modification is required to remove a bug reported by the final user of the system;
- *Adaptive*: the modification is required to adapt the working system to a new running environment (such as a new operating system).

Software Engineering textbooks describe successful strategies for putting in place evolution activities. These best practices recommend to accurately plan the changes off-line, taking into account the characteristics of the whole system. Once the change has been designed thoroughly, it is typically implemented off-line and integrated in a copy of the running system. The integration is checked, for instance by running a regression test suite, and, if the tests are successful, the old system is substituted with the new version. With this traditional testing approach, any running system is (or, at least provides the opportunity to be) accurately checked before being exposed to real workloads.

This is an ideal situation in which there are no strict time constraints for concluding the evolution step. However, the reality is much more complex, and changes are not always performed as cleanly as described above. Nevertheless, evolution can be considered, so far, an activity to be carried out by a single organization controlling and having a clear view on the whole system.

For many reasons, the situation is rapidly changing. Software today has high availability requirements and, increasingly, must be extended at run-time. In some cases, the various system components become available after the initial deployment of the system, and integration tests cannot be conducted off-line. This is the case, for instance, for Service-Oriented Applications, where several services developed, deployed, and owned by different organizations can start co-operating at run-time to fulfill a common goal. Moreover, there are situations where the whole system changes continuously, integrating and substituting components on-the-fly. In other situations, the system cannot be switched off to be upgraded, or it may not be possible to make a faithful copy of the entire running system for performing off-line testing.

Such scenarios require a novel approach for verification and validation (V&V). To account for the *transition phases*, when the system undergoes changes that could not have been tested in advance, V&V operations must be be executed at run-time to ensure that the evolution does not impact the dependability of the live system. These operations must have two goals, shown in Figure 1: (i) test the correctness of the transition procedure (*e.g.*, the online upgrade mechanism); and (ii) test the final state of the system, which represents the outcome of the transition, in its operational environment.

2 Evolution scenarios

In order to identify the concrete challenges for testing evolving systems, the workgroup has discussed three realistic scenarios where the current testing approaches testing are insufficient.

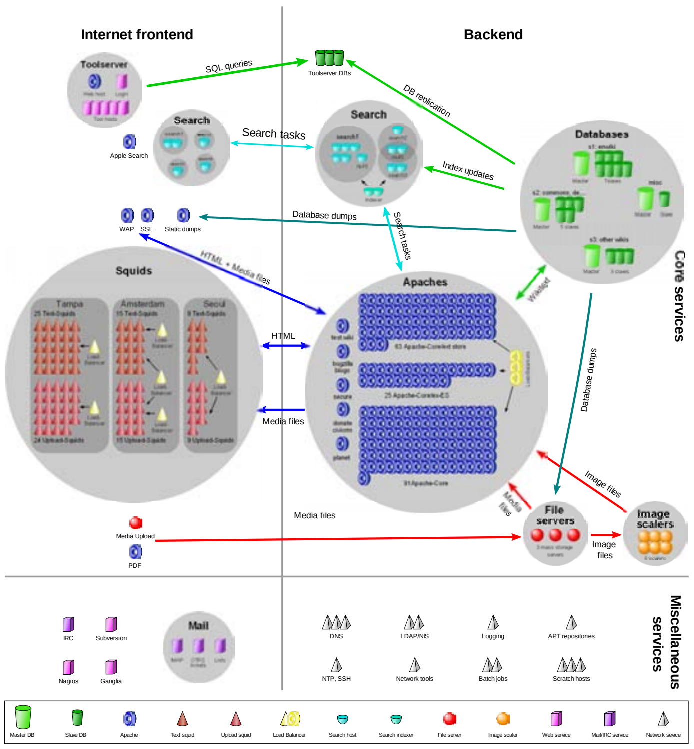
2.1 Online upgrades of enterprise systems

Wikipedia (`http://www.wikipedia.org`) is one of the ten most popular websites to date,[1] providing a multi-language, free encyclopedia. The English-language Wikipedia has 17 million articles, with content stored in a 1 TB database and 850,000 static files (*e.g.* images). The web site is supported by an infrastructure (Figure 2) running on 352 servers, including 23 MySQL database servers configured for master/slave replication. A wiki engine called MediaWiki, implemented as a set of PHP scripts, accesses the database and generates the article content.

Our first scenario would upgrade Wikipedia's business logic to a completely different wiki engine, which provides similar, but not entirely equivalent functionality. This upgrade is performed online, in the presence of Wikipedia's live workload. This upgrade scenario implements major changes: the *persistent data must be migrated to a different data store* (*e.g.*, a distributed file system instead of a database) and *converted to a new format* (the markup language used by the new wiki engine); the *new application exhibits different behaviors* than the previous wiki engine; and the *interface presented to the users is different*. This example simulates the practical scenario of a a competitive upgrade, performed when business reasons mandate switching software vendors.

Existing techniques for online upgrades [2,3,4,5,6] focus on upgrading enterprise systems *in-place*, without additional storage and computational resources, and on supporting *mixed versions*, which interact and synchronize their states in the presence of a live workload. During transitional states, with mixed versions, the system behavior does not correspond to the specification of either the old or the new versions. Testing and validating the behavior of such transitional states is difficult, because the mixed versions can induce behaviors that are difficult to understand and to reason about [2]. For instance, ensuring the correctness of mixed-version interactions [3,6] typically requires tedious and error-prone developer interventions, such as establishing constraints to prevent old code from accessing new data [6]. Moreover, for in-place upgrades, run-time testing would be conducted in parallel with the live workload and would risk overloading the

[1] According to `http://www.alexa.com`. Wikipedia handles peak loads of 70,000 HTTP requests/s.



Source: http://meta.wikimedia.org/wiki/Wikimedia_servers

Fig. 2. Wikipedia architecture.

production system. For these reasons, enterprise-system upgrades are often not tested appropriately; a 2006 survey suggests that 84% of upgrades are tested and deployed in different environments [5], which increases the risk of upgrade failures. Operator errors [4,5] further reduce the effectiveness of offline testing in ensuring the dependability of enterprise upgrades.

Alternatively, an approach for performing atomic, online upgrades, by using additional hardware and storage resources, has been proposed [7]. The new version is installed in a *parallel universe* – a logically distinct collection of resources, realized either through virtualization or by leasing storage and compute cycles, during the upgrade, from existing cloud-computing infrastructures (*e.g.* Amazon's Elastic Compute Cloud) –, and, while the old version continues to service the live workload, the persistent data is transferred, opportunistically, into the new version. This approach supports a series of run-time testing phases before switching over to the new version, without disrupting the live workload. It is possible to validate the new version through either *offline testing* – using pre-recorded or synthetically-generated traces – or *online testing* – using the live requests received and processed by the old version. After adequate testing, the upgrade can be rolled back, by simply discarding the parallel universe, or committed, by making it the production system, thus completing the atomic switchover to the new version. However, in the Wikipedia-upgrade scenario, the outputs of the old and new systems cannot be compared directly because non-deterministic executions and behavioral differences between the two systems may lead to state divergence. The information that can be elicited through such online testing remains an open research question.

2.2 Run-time evolution of component-based applications

Testing applications based on component frameworks, such as OSGi [8], presents additional challenges, because the transition phase constitutes the normal mode of operation for these systems. The run-time evolution is typically specified as structural changes [9], such as *adding, removing, or replacing components* or *changing connectors dynamically.* As in the previous example, the run-time configuration does not correspond to a system state that was tested in advance. Because of this reason, the system must be tested at run-time, in the operational environment. While not eliminating the need for traditional unit- and integration-testing, this approach is needed to improve the dependability of systems with zero-downtime requirements. Moreover, component-based applications often depend on third-party services, and compositions of distributed OSGi modules can be upgraded online [10]. Unlike in the enterprise-upgrade scenario, the online testing procedure might not control all of the system's stateful resources (*e.g.*, the data store), which would render rollback after failed tests more difficult and would present the risk of violating existing service-level agreements.

2.3 Run-time patching of real-time software

Real-time applications must often be upgraded online as well. The control software of satellites in orbit requires periodic patches, to change the memory layout or to correct software bugs. For instance, dynamic memory allocation is prohibited in satellite software, in order to avoid unpredictable behaviors and to meet the real-time deadlines; however, equipment aging and damages due to space radiation impose changing the memory layout. Such damages make each satellite unique, even within the same series, and software must be adapted specifically to each satellite [11]. The changes performed in this case are more restricted than in the previous scenarios and are typically limited to *modifying the implementations* of existing real-time tasks. The biggest testing challenge for this scenario is to ensure that the control software will continue to meet its real-time deadlines – which are imposed by external factors, such as the satellite's orbit and temperature – during and after the upgrade.

2.4 Summary

While strikingly dissimilar, these three scenarios emphasize a set of common challenges for testing evolving systems:

– The transition phase must be tested, taking into account actions performed by both *software and human agents*;
– The final state of the system must be tested in its *operational environment*, in order to ensure a correct and dependable behavior in *unique run-time configurations* and to meet *externally-imposed timeliness constraints*;
– The testing procedure must not assume that the entire operational environment can be controlled and must avoid violating the existing service-level agreements;
– A systematic way is needed for reasoning about the behavior of systems under continuous evolution, either by establishing evolution invariants or by evolving the oracle together with the system it models;
– The testing procedure must provide fault-management strategies, including the ability to roll back after a failed test.

3 Conclusions

The discussion initiated in this abstract highlights that verifying and validating run-time evolving systems is challenging and requires novel techniques beyond those used in systems that undergo off-line evolution. This work is in a preliminary stage and represents the outcome of informal discussions within our workgroup at Dagstuhl. Further research is needed to refine these initial findings and to implement solutions to the challenges presented here.

References

1. Sommerville, I.: Software Engineering. 8th edn. Addison–Wesley (2007)
2. Segal, M.: Online software upgrading: new research directions and practical considerations. In: Computer Software and Applications Conference, Oxford, England (2002) 977–981
3. C. Boyapati et al.: Lazy modular upgrades in persistent object stores. In: Object-Oriented Programing, Systems, Languages and Applications, Anaheim, CA (2003) 403–417
4. Nagaraja, K., et al.: Understanding and dealing with operator mistakes in Internet services. In: USENIX Symposium on Operating Systems Design and Implementation, San Francisco, CA (2004) 61–76
5. Oliveira F., et al.: Understanding and validating database system administration. USENIX Annual Technical Conference (2006)
6. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical dynamic software updating for C. In: ACM Conference on Programming Language Design and Implementation, Ottawa, Canada (2006) 72–83
7. Dumitraş, T., Tan, J., Gho, Z., Narasimhan, P.: No more HotDependencies: Toward dependency-agnostic upgrades in distributed systems. In: Workshop on Hot Topics in System Dependability, Edinburgh, Scotland (2007) 14
8. OSGi Alliance: OSGi service platform, core specification, release 4, version 4.1 (2007)
9. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306

10. Rellermeyer, J.S., Duller, M., Alonso, G.: Consistently applying updates to compositions of distributed osgi modules. In: ACM Workshop on Hot Topics in Software Upgrades. (2008)
11. Buisson, J., Carro, C., Dagnat, F.: Issues in applying a model driven approach to reconfigurations of satellite software. In: ACM Workshop on Hot Topics in Software Upgrades, Nashville, Tennessee (2008)