Towards Preserving Correctness in Self-Managed Software Systems*

Lieven Desmet, Nico Janssens, Sam Michiels,
Frank Piessens, Wouter Joosen and Pierre Verbaeten
DistriNet Research Group, Department of Computer Science,
Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Lieven.Desmet, Nico.Janssens}@cs.kuleuven.ac.be

ABSTRACT

Currently, paradigms such as component-based software development and service-oriented software architectures promote modularization of software systems into highly decoupled and reusable software components and services. In addition, to improve manageability and evolveability, software systems are extended with management capabilities and self-managed behavior. Because of their very nature, these self-managed software systems often are mission critical and highly available. In this paper, we focus on the complexity of preserving correctness in modularized self-managed systems. We discuss the importance of consistent software compositions in the context of self-managed systems, and the need for a correctness-preserving adaptation process. We also give a flavor of possible approaches for preserving correctness, and conclude with some remarks and open questions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.9 [**Software Engineering**]: Management; D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software

Keywords

software architectures, distributed software compositions, runtime software reconfiguration

1. INTRODUCTION

*Research for this paper has been carried out with financial support of the Fund for Scientific Research Flanders, Belgium – F.W.O. RACING # G.0323.01, and of the Institute for the Promotion of Innovation by Science and Technology in Flanders, Belgium – I.W.T. SOBENET # SBO-2003 IWT-30313

WOSS'04 Oct 31-Nov 1, 2004 Newport Beach, CA, USA

Modern software systems evolve towards modularly composed services, in which existing software components are reused within new compositions [27]. The different services are interconnected into distributed software systems, using service-oriented software architectures [17]. Both the component-orientation as well as the service-orientation promote a shift towards highly decoupled and reusable software units.

Furthermore, two important requirements for maintaining large-scale software systems are *manageability* and *evolveability*. In order to address these concerns, an open software architecture is required, as well as support for adding, removing and replacing parts of the running software system. DiPS+ [20], a rapid prototyping framework for self-manageable protocol stacks, is an example of such an open architecture. Pluggable DiPS+ extensions exist for adding resource control and self-management support [21, 22] as well as adding support for component hotswapping [12, 13].

Over the last decade, there has been a trend towards using open architectures in the development of *highly available and mission critical systems* such as computer networks [15], application servers[26] and avionics systems [24]). Because of the stringent dependability requirements inherent in such critical systems, self-management should be prevented from jeopardizing among others the correct functioning of the system. By consequence, *correctness* is a crucial property in architecting dynamic and self-managed dependable systems.

In our opinion, the complexity of *preserving functional correctness* in modularized, self-managed distributed systems is often underestimated. The correctness of a dynamic and distributed system implies (1) a consistent software composition for each of the services with respect to the dependencies between software components, (2) a correct composition of the different services into a distributed system and (3) functional correctness of the overall system during and after any possible adaptation.

In section 2, we focus on the complexity of preserving functional correctness in self-managed distributed systems. In section 3, we highlight some approaches to achieve functional correctness. Finally, in section 4 we summarize with some of our open questions.

2. DISTRIBUTED CORRECTNESS

In this section, the different aspects of preserving correctness in a self-managed distributed software system are discussed. Section 2.1 focusses on the *consistent composition* of software systems and services. In section 2.2, the ad-

ditional challenges to maintain functional correctness in a self-managed system during an online adaptation (i.e. *dynamic consistency*) are discussed.

2.1 Consistent composition

Building distributed software systems requires the different software nodes and services to be correctly interconnected. Moreover, in component-based software development, each of these software nodes or services needs to represent a correct composition.

This *consistent composition* with respect to dependencies between components in distributed systems is already investigated in the context of ADL's [4, 19], component contracts [2, 9] and software dependency analysis [25, 31, 18]. Some problems in achieving consistent compositions are still to be resolved.

The importance of component contracts and dependency analysis is commonly accepted. However, there is no consensus yet about what information should be contained in a component's contract. Similarly, several analysis techniques are described in literature, but no exhaustive list of dependency types exists, that is crucial to analyze in order to achieve a robust and dependable system.

For example, the current specification of component dependencies in loosely-coupled software systems is sometimes incomplete. In [6], we observed that the dependencies between components in systems with a shared data repository, or in systems with implicit invocation through events, are typically left implicitly in current contracts and ADL's. In applications with a shared data repository, data providing components put their data on the shared repository, available for data consumers to fetch this data for internal computations. Since such applications are loosely-coupled, there exists no concrete binding between the data providers and consumers at a component's design-time. The dependency instance between the data sharing parties is only introduced during composition-time and is composition dependent.

Furthermore, most ADL's and dependency analysis techniques are based upon an architectural representation of the system. Since these descriptions could differ from the actual implementation or the running system, their results could be inapplicable. Therefore, a stricter correspondence between descriptions and the actual running system is needed [1].

In the context of self-managed software systems, having a consistent composition is even much more challenging. Every adaptation in the system involves in fact defining and deploying a new consistent composition. Moreover, keeping the software description consistent with the running self-managed system at any moment in time is very hard without incorporating the description into the system. Problems related to the actual online deployment of the new composition are further discussed in the following section.

2.2 Dynamic consistency

Due to availability and performance requirements, mission-critical and highly available systems cannot be brought down easily or be switched off for a long time (e.g. in case of applications that need to provide 7x24 hours availability). Therefore, deploying a new composition in a self-managed system should be accomplished *dynamically*, without interrupting those parts of the application that are unaffected by the change (hot-swapping).

To prevent dynamic changes from jeopardizing the correct functioning of a system, change actions must be carried out such that the *consistency* of the software modules making up the system is not compromised [16, 7]. Due to their interdependencies, the *cooperation* between dependable components is essential to successfully perform the service (illustrating the need for consistent composition as discussed in the previous section). This cooperation is formalized by means of a transaction, consisting of a sequence of one or more interactions between the tightly-coupled collaborating entities. Through this, from a reconfiguration point of view, dependable components are only consistent after termination of a transaction. By consequence, as we stated in [12], *coordinating* the runtime deployment of dependable components is essential to avoid breaking the consistency, and as such preserving the integrity of the service under evolution.

In case of distributed dependable systems, two types of online reconfigurations can be enforced, *isolated* and *distributed* adaptations.

2.2.1 Isolated adaptations

For this kind of online reconfigurations, replacing the functionality of one dependable component will not break the semantic consistency with its collaborating counterparts (and vice versa), nor will the communication protocol be changed. An example of such adaptation is the replacement of a reassembling component (restoring the original data packet out of a number of fragments as part of a fragmentation service) by a new bug-fixed version or a version extended with additional non-functional support (such as logging). From a composition point of view, this will not affect the correct cooperation with the existing fragmentation component (breaking up each data packet into a number of fragments).

From a reconfiguration point of view, *coordinating* the online replacement is essential to avoid inconsistencies, caused by replacing a component when transactions are only partially completed, as we presented in [12]. Illustrated by the fragmentation service, replacing the reassembling component at the moment when it has not yet received all fragments (and as such has not yet reassembled the original packet) will break the consistency between the fragmenting and reassembling component. Subsequently, the correct functioning of the fragmentation service will be compromised.

2.2.2 Distributed adaptations

In contrast to the isolated adaptations, distributed adaptations require all collaborating components to be replaced in order to preserve semantic consistency. As an example, the fragmention service could be replaced by a new version that is extended with error-correction support. It is obvious that (for the fragmentation service to work correctly) both the fragmentation and the reassembling component will have to be replaced by a new version that can handle error-correction.

To prevent online adaptation of such services from jeopardizing the correct functioning of dependable distributed systems during and after the adaptation, *safe deployment* is essential. When a sending node is extended with the new fragmentation component before the new reassembling module has been installed on its peer, packets cannot be processed correctly. By consequence, the correct functioning of the network is compromised. Performing adaptations of peer nodes at runtime requires a *coordinated deployment proto-*

col to preserve the integrity of the distributed service while such an online adaptation is in progress [14, 28]. Hence, the development of self-management in dependable systems is often complex and error-prone.

3. APPROACHES

In the previous section, some problems were discussed in preserving correctness in self-managed software systems. In this section, a number of ongoing research tracks are briefly presented. This section does not claim to contain a exhaustive listing of approaches, but rather a flavor of possible solutions and their complexity is given.

3.1 Consistent composition

In [6], we presented a first approach for specifying and enforcing implicit dependencies between components, interacting through a shared data repository. First, each component is extended with a specification of required and provided data interactions with the shared repository. Next, data dependencies through the shared repository are explicitly described for each composition. Finally, the dependencies are analyzed at deployment time, or enforced at runtime. This approach has been validated in a Java Servlet-based web application and in the DiPS+ rapid prototyping framework for dynamic protocol stacks.

ArchJava [1] offers a unique binding between architectural description and actual implementation. Architectural styles and constraints could be expressed within language constructs of the actual program. Byte-code compiled with the ArchJava compiler is guaranteed to comply with the expressed architectural constraints, as such preserving the intended correctness.

Although ArchJava is able to enforce architectural constraints independently for each component instance within a composition, all architectural constraints are hard-coded within the components. In our opinion, component characteristics, constraints and styles that can be expressed for each component at implementation time are not sufficient. Provisions are also needed in systems such as ArchJava to express composition-typical constraints at deployment time. In this way, components are much more reusable in different contexts and compositions.

3.2 Dynamic consistency

3.2.1 Isolated adaptations

A first solution to prevent isolated adaptations of dependable components from compromising the correct functioning of a service (as discussed in section 2.2.1), implies imposing a *safe state* over software modules under change. Kramer and Magee [16] have stated that achieving *safe software reconfigurations* requires the software modules that are subject to adaptation (such as a reassembling component) to be both *consistent* and *frozen*. When software modules are consistent, they do not include results of partially completed services (or transactions). By forcing software modules to be frozen, state changes caused by new transactions are impossible. Kramer and Magee describe this required consistent and frozen state as the *quiescence* of a component. They propose a mechanism to impose such a quiescent state by means of a configuration manager, which is employed to recognize and deactivate (or freeze) the relevant transaction initiators [16]. These transaction initiators are the components in the system that are able to start transactions capable of causing state changes on the components that are targeted for reconfiguration.

As an alternative, Hofmeister [8] has proposed to freeze a component immediately instead of waiting for it to reach a desirable state. However, since there is no guarantee about termination of a transaction in process at the moment the actual reconfiguration is conducted, reconfiguration could endanger the consistency. By consequence, safe reconfiguration can only be enforced by means of *additional consistency recovery* support, which requires software modules to capture and reinstate module specific application-state at runtime. Inconsistencies are allowed during reconfiguration, as long as consistency returns when the reconfiguration is complete.

In [12], we presented a mechanism to obtain a safe state for unanticipated reconfiguration of producer-consumer based systems. The presented solution requires minimal contribution from the programmer and causes minimal interference to the rest of the system. In addition, support to achieve safe reconfiguration (which has been demonstrated to cross-cut through the system undergoing change) has been *modularized* and *separated* from basic application functionality.

The problem of safe and reliable unanticipated software evolution has also been recognized in the area of *dependable real-time systems*. The HERCULES framework [5] and the SIMPLEX architecture [23, 24] propose a different strategy to achieve safe reconfigurations. Both frameworks allow for the old and new version of a module to coexist during a reconfiguration. Once the output of the new software module converges with that of the old one according to user provided criteria, a safe reconfiguration has occurred. Consequently, the output of the old module is turned off and the new module is used. However, testing for convergence of the output generated by two coexisting components can be very difficult. In addition, there might be no guarantee concerning convergence anyway. By consequence, we believe this approach is limited in its ability to be employed in a wide range of applications.

3.2.2 Distributed adaptations

A lot of research in the field of online distributed adaptations has been done in the past. Systems like Cactus [3] and Ensemble [30] perform *distributed adaptations* at runtime (as described in section 2.2.2), preserving the consistency of the system during the adaptation.

However, these adaptable systems are developed to service a specific application domain: building adaptable protocol stacks and middleware. This limits their potential to cover a broad range of mission critical and highly available systems. Therefore, we believe that to simplify the development of self-managed distributed systems, a *generic distributed coordination platform* responsible for *safe distributed service deployment* at runtime is needed. As such, by packaging this complexity, the developer of a self-managed distributed system is spared from implementing a service specific deployment protocol. In this way, we aim to make the development of correct functioning self-managed systems less error-prone. In [11], we have presented the current version of NeCoMan, a middleware platform that is developed to comply with these requirements in flow-based component architectures.

Lasagne [29], presented as an architecture for building

runtime adaptable middleware layers, has adopted a different approach to guarantee service consistency during and after adaptation. Rather than replacing components with new versions (so as to change the composition of the system), the Lasagne model employs a *wrapper based* component extension mechanism. Customizations are based on *additive refinements* of stable core components, to be expressed on a *per-request basis*. Once an extension (such as reliability or encryption) is selected by a client, this client-specific customization propagates together with the message flow of the entire collaboration, providing in system-wide execution consistency. By consequence, conducting safe distributed software adaptation boils down to coordinating the proper installation of the wrappers on each host before being selected by a client request [28]. However, in the same way that NeCoMan is targetted to flow-based component architectures, the Lasagne methodology is limited to be applied to wrapper based compositions.

Finally, to ensure safety in re-composable software systems, Zhang et al. proposed a method that determines viable sequences of adaptation actions and according safe starting states, based on dependency analysis [10]. Since all determined adaptation actions of their method can be considered as small service deployments in a distributed environment, the approach presented in [11] is complementary to their method and can be combined in a more general safe adaptation technique for larger systems.

4. CONCLUSION AND OPEN QUESTIONS

In this paper, we tried to express our concern that preserving functional correctness in a dynamic, self-managed system can be quite challenging. We believe that preserving correctness is an important issue to resolve in order to get self-managed software systems commonly accepted and used in the wild.

Since the presented work is ongoing research, we still have some remarks and open questions that may be interesting to consider in the context of this workshop:

- Which software systems can benefit of self-managed behavior? Is there any need for mission critical, self-managed software systems?
- Can our presented correctness be relaxed in some cases, such as allowing a temporary non-consistent composition or a correctness-unpreserving update process? Can fault-tolerance techniques be seen as complementary to the correctness preserving features and allowing some of the relaxations?
- What is the unit of adaptation in self-managed systems: parameters, software components, services? Are the adaptations local to one service or component, or are they distributed throughout the system? Are all adaptations known a priori, or need the system to be open to unanticipated adaptations?
- What is the right balance between formal description and analysis at the one hand, and useability by the end user at the other hand? Can tool support completely bridge this gap?

5. REFERENCES

[1] J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. In *ieee-software*, pages 38–45, june 1999.

[3] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing Adaptive Software in Distributed Systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'02)*, pages 635–643. IEEE Computer Society, 2002.

[4] P. C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, page 16. IEEE Computer Society, 1996.

[5] J. E. Cook and J. A. Dage. Highly Reliable Upgrading of Components. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE '99)*, pages 203–12, Los Angeles, CA, 1999.

[6] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten. Improving software reliability in data-centered software systems by enforcing composition time constraints. In *Proceedings of Third Workshop on Architecting Dependable Systems (WADS2004)*, pages 32–36, Edinburgh, Scotland, May 2004.

[7] K. M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, pages 62–69, Annapolis, MD, USA, May 1996. IEEE Computer Society Press.

[8] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Department of Computer Science, University of Maryland, Jan. 1994.

[9] I. M. Holland. Specifying reusable components using contracts. In *European Conf. on Object-Oriented Programming*, pages 287–308, Utrecht, Netherlands, 1992. Springer Verlag Lecture Notes 615.

[10] B. H. C. J. Zhang, Z. Yang and P. K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of Third Workshop on Architecting Dependable Systems (WADS 2004)*, pages 17 – 21, Edingburgh, Scotland, May 2004.

[11] N. Janssens, L. Desmet, S. Michiels, and P. Verbaeten. NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks. In *Middleware 2004 Workshop on Reflective and Adaptive Middleware (RM 2004)*, Toronto, Ontario, Canada, Oct 2004. To Appear.

[12] N. Janssens, S. Michiels, T. Holvoet, and P. Verbaeten. A Modular Approach Enforcing Safe Reconfiguration of Producer-Consumer Applications. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, Chicago Illinois, USA, Sept. 2004.

[13] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *Proceedings - The Seventh International Workshop on Component Oriented Programming (ECOOP-WCOP 2002)*, 2002.

[14] N. Janssens, E. Steegmans, T. Holvoet, and

P. Verbaeten. An Agent Design Method Promoting Separation Between Computation and Coordination. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004)*, pages 456–461. ACM Press, 2004.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[16] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[17] I. H. Krüger and R. Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, Oslo, Norway, 2004. IEEE/IFIP, IEEE.

[18] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14. ACM Press, 1996.

[19] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer-Verlag, 1997.

[20] S. Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, Nov. 2003.

[21] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten. Self-adapting concurrency: The DMonA architecture. In D. Garlan, J. Kramer, and A. Wolf, editors, *Proceedings of the First Workshop on Self-Healing Systems (WOSS'02)*, pages 43–48, Charleston, SC, USA, 2002. ACM SIGSOFT, ACM press.

[22] S. Michiels, L. Desmet, W. Joosen, and P. Verbaeten. The DiPS+ software architecture for self-healing protocol stacks. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-4)*, Oslo, Norway, 2004. IEEE/IFIP, IEEE.

[23] J. G. Rivera, A. A. Danylyszyn, C. B. Weinstock, L. R. Sha, and M. J. Gagliardi. An Architectural Description of the Simplex Architecture. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.

[24] L. Sha, R. Rajkumar, and M. Gagliardi. Evolving dependable real-time systems. In *1996 IEEE Aerospace Applications Conference. Proceedings*, pages 335–46, Aspen, CO, 3–10 1996. IEEE New York, NY, USA.

[25] J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–451, 2001.

[26] Sun Microsystems, Inc. J2EE platform specification. Available at http://java.sun.com/j2ee/.

[27] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley Professional, 2nd edition, November 2002.

[28] E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. Consistency Management in the Presence of Simultaneous Client-Specific Views. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 501–510. IEEE Computer Society, Oct. 2002.

[29] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May 2001.

[30] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software – Practice & Experience*, 28(9):963–979, 1998.

[31] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. In *Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 241. IEEE Computer Society, 2002.