FUNCTIONALITY RECOMPOSITION FOR SELF-HEALING

Josu Martinez, Simon Dobson
Systems Research Group, School of Computer Science and Informatics, UCD Dublin IE, Ireland
josu.martinez@ucd.ie, simon.dobson@ucd.ie

Keywords: Autonomic Computing; Formal Methods; Distributed Architectures; Software Composition.

Abstract: Autonomic computing aims to provide self-management and adaptation in the implementation of complex (large, heterogeneous, distributed) systems over time. Such adaptations must be stable, in the sense of maintaining the system's high-level goals across environmental changes, which may lead to functionality loss. In this paper we present FReSH, a decentralised component-based framework which main objective is to self-heal the operation of complex systems in the face of behavioural disruptions. FReSH deals with formally specified components that provide a single piece of functionality. The reusable and shareable nature of these building blocks makes them eligible for dynamically recomposing the functionality provided by any failing component of the system without human intervention. FReSH supports the construction of more flexible, adaptive and robust software structures suitable to cope with the environmental changes of complex systems.

1 INTRODUCTION

The technological advances of the recent years have enabled the creation of new forms of dynamic interactions and global cooperation among social organizations and business communities. Due to such constantly increasing societal expectation of this new era of computing, a myriad of heterogeneous and distributed components are emerging to constitute corporate-wide computing systems that extend beyond company boundaries (Nachira, 2007). However, the construction of these type of systems introduce new levels of complexity, not only because they may evolve in size over the time, but also because both new and old components must co-habit sharing their resources while ensuring the correct operation of the system.

One of the consequences is the possibility that some functionality of the system becomes unavailable at run-time. As it can be inferred from the results obtained from some case studies that analyse complex systems (Patterson et al., 2002), in these environments the probability of experiencing service interruption is proportional to amount and grade of interdependencies between components. Thus whenever components become unavailable e.g., due to the high network traffic or operation latency issues (components may depend on other components' services to perform their operations), because they crash or they are inconsistently updated, or simply because the machines they reside in are suddenly switched off, system users may perceive that the system cannot provide some specific functionality any more. The worst case scenario is that the activity of the system results unexpectedly interrupted, which is not admissible at all.

Because these type of systems evolve very rapidly human administrators cannot cope with healing tasks to ensure the correctness of their operation, and therefore a new formula to do so is required. This paper presents a different self-healing approach specifically designed to handle component unavailability without operation interruption in such complex environments. We introduce FReSH, a decentralised component-based Java framework able to detect operation disruptions at run-time and recompose any missing functionality by dynamically identifying, reusing and self-assembling software pieces (i.e., local and remote services) distributed over the various nodes that belong to the system environment. Ideally, these

software components can be functions, methods, procedures, web services and even compositions of the previous, and each provides some basic functionality. Hence, FReSH can be used to build tractable and dependable applications out components of different nature, that flexibly adapt to environmental changes.

2 RELATED WORK

Since IBM coined the term autonomic computing in 2001 many self-healing initiatives have been researched. None of them are able in their own to cope with the difficulties arisen in complex environments described above in a completely autonomous manner. However, far from rejecting them our proposed self-healing strategy comprises and reuses some of their conceptual properties, as exposed in Section 4.

We have analysed some approaches that fall into four different self-healing categories: component redundancy, architecture models, component micro-rebooting and SOA-based process reorganization.

A component redundancy technique suggests a self-assembly mechanism based on an agent entity that replicates components to replace dead neighbours and enables recomposition of entire structures (Nagpal et al., 2003). Another strategy inspired from biology is providing the system with the ability of replicating cells in excess to combat external intrusions (George et al., 2003). On another hand, Recovery-Oriented Computing (ROC) (Berkeley/Stanford, 2008) suggests to isolate faulty components and replace them with redundant ones.

However, a full replication of all the components that populate the system may introduce performance issues due to the need of having to perform complex redundancy management tasks. Another problem of this approach is that many of the nodes containing the components may be portable devices, which are constrained by tight memory limits.

One subset of self-healing techniques based on architecture models focuses on dynamically reconfiguring the connector links among components to correct performance deviations (Georgiadis et al., 2002). Some other approaches (Appavoo et al., 2003; de Lemos and Fiadeiro, 2002) replace failing services or components by functionally equivalent ones. Some decision policies determine which alternative component replaces the original one.

The main problem that this strategy presents is that complex environments change very often, and thus many of those components may not survive in the system for a long time whereas others may appear or evolve (Nachira, 2007; Kephart and Chess, 2003). This fact makes repair plans become obsolete even before applying them to the running system, and enforces the administrator to constantly update them. Furthermore, most of the researched solutions rely on a centralized approach.

On another hand, faulty modules can be micro-rebooted independently and automatically to avoid fault propagation whenever they are suspected of not functioning properly (Patterson et al., 2002). The efficiency of this technique resides on the fact that re-starting single components takes less time than re-booting the whole system.

In highly distributed environments where components may have a large number of inter-dependencies re-starting failing software artefacts may not be an efficient and reliable option (Tanenbaum and Steen, 2001). In certain situations where components may have to be re-started e.g., due to multiple machine power-cuts or heavy and persistent network traffic overloads other components that request services from the failing ones may remain blocked too long, which may degrade the performance of the entire system to unacceptable limits.

Finally, Service-Oriented Architectures (SOA) is a flexible coordination paradigm that enables components to export services over the network (Papazoglou and Georgakopoulos, 2003). These services can be discovered and dynamically bound at run-time to provide higher level services to other components (Baresi et al., 2004).

Despite the apparent success on efficiently building reliable and robust service structures, web-based mechanisms do not properly address self-healing in complex environments. There is a trade-off between constructing loose-coupled service-based structures to improve maintenance and flexibility, and fulfilling the non-functional requirements of the system during its execution (Baker and Dobson, 2005). Certain properties of the environment such as the network bandwidth may negatively affect the operational latency of some components. Hence, a more suitable architectural approach that reduces the inter-dependencies among remote services is required for complex systems.

3 ATOMIC AND COMPOSITE COMPONENTS

A software component is a unit of composition that provides some functionality with contractually specified interfaces and explicit context dependencies only (Clemens Szyperski, 2002). However, once compiled these components become self-contained

pieces of software that remain as unalterable black boxes during their execution. Hence, their reuse to build new applications exclusively depends on the knowledge programmers may have about the functionality they provide. From this fact it can be inferred that once these components become run-time software entities they cannot be automatically composed at run-time.

Similarly to some of the existing ontology languages created by the SOA community, such as OWL-S (Martin et al., 2007), our approach consists on associating components with some formal specification of the functionality they provide, so that they can be identified and reused to autonomously compose higher level software structures. At this preliminary development stage we are using the JML state-based specification language (Burdy et al., 2005) to specify the functional behaviour of atomic components developed in Java. JML is a rich specification language used to formally describe the functionality of Java objects. Other atomic components such as web services also must also have some associated state-based specifications to be identified and reused.

These unambiguous specifications can be matched with other specifications to find equivalences among the components they describe (Zaremski and Wing, 1995). Furthermore, the specification of a complex component may be automatically decomposed into some simpler, lower-level specifications (van Lamsweerde, 2000), which in turn could be matched with the specifications of smaller components. One of the key points of our proposal consists on providing rich post-condition expressions to every reusable component of the system, so that whenever it becomes unavailable it can be autonomously replaced by another fully equivalent component or a specially combined set of components which specifications match with the sub-specifications of the former.

To create component compositions we are using ORC (Misra and Cook, 2007), a concurrent and distributed component orchestration language with all the required operators to implement sequential and parallel structures. Moreover, ORC also provides other features to reproduce a whole set of classic programming idioms such as e.g., the conditional and iterative statements. ORC components constitute computational recipes where single components can be glued together to compose higher-level components, and therefore these relationships can be autonomously altered at run-time. On another hand, similarly to the atomic components these recipes perform some computation operations, and thus they are considered composite software components.

Hence, ORC components must also be provided with some associated formal specifications. This is the base of our self-healing strategy and main contribution. In the case of component unavailability (because e.g., a node that contains certain components crashes, or the network traffic is so high that some nodes become disconnected) an ORC component is automatically created to replace the failing one. An ORC component may comprise other ORC components. If the unavailable component is an ORC component, depending on the causes of failure the newly created ORC component may reuse some of the components that were used by the failing structure, or it may use completely different ones.

Components are selectively replicated in other nodes of the system to reduce the inter-dependencies among remote services, and thus overcome the SOA drawbacks discussed in Section 2. However, some of the components may be too heavy to be sent through the network or there may exist certain legal issues that may prevent components to be shared among nodes, and therefore they may only be remotely invoked.

4 FUNCTIONALITY RECOMPOSITION

FReSH is an autonomic component-based framework with reflective capabilities. Similarly to other autonomic frameworks such as Unity (Tesauro et al., 2004), in FReSH every node of the system contains an autonomic manager (AM) that supervises all the components included in the node and performs functionality recomposition whenever any of these components become unavailable.

More in detail, an autonomic manager must be able to perform *introspection* i.e., it must constantly monitor the operation of its managed components and detect behaviour inconsistencies. At this stage of the framework development we assume the existence of a component unavailability detection mechanism based on probe signals (Balasubramaniam et al., 2005), which regularly sends probes to the components to check the validity of its status. If a component does not return any feedback to the autonomic manager within a reasonable amount of time it is considered unavailable. In this case, the corresponding autonomic manager in charge of that component must execute some *intercession* actions i.e., it must carry out certain procedure to fix the problem without interrupting the operation of the system. This procedure consists first on realising the functionality that the unavailable component was providing, and second on finding a functionally equivalent component to replace the failing one (direct substitution). If no

equivalence is found, the autonomic manager must make a local or global search for a combination of other components that can be used to recompose the missing functionality (composite replacement).

This strategy differs from direct replication and replacement of components as it supports the recreation of unavailable functionality from already existing components. Moreover, it increases the probability of obtaining suitable software structures to replace unavailable components while decreasing operational costs related to replication. For all these reasons, FReSH is a more efficient strategy to perform self-healing in complex environments than other existing alternatives. The example exposed in Figure 1 shows how the system obtains the functionality of a failing component by identifying, selecting, obtaining and recomposing some other components. Although the example shows how composite replacement is performed, it is illustrative enough to understand how direct substitution works.

Each node contains a catalogue with all the specifications of the components included in it so that the corresponding autonomic manager explicitly knows the software entities it must supervise. Some formal specifications contain more complex predicates than others. As mentioned above, complex specifications can be interpreted as a combination of simpler specifications associated to other component implementations that may exist in the system, which is the base for achieving functionality recomposition.

Whenever any component of any particular node becomes unavailable at run-time, the autonomic manager responsible of its health must first suspend the execution of any other component that consumes services from it. Furthermore, the autonomic manager must extend this requirement to the rest of the autonomic managers to avoid cascading failures (this step is not shown in Figure 1 for simplicity.)

First the corresponding autonomic manager must internally search for an equivalent component. To do so, it compares the functional specifications of the unavailable component with the specifications of other components described in the local component catalogue (step 2). If no equivalent component is obtained, this search is extended to other autonomic managers so that they look up their catalogues in the pursuit of an equivalent component (steps 3 and 4). Should any existing remote component match the specifications of the unavailable component, the remote autonomic manager sends that component to the local autonomic manager over the network (steps 5, 6 and 7). However, due to certain constraints such as e.g., size, amount of inter-dependencies or commercial restrictions the transfer of some of these components may not be permitted, and thus they must be invoked remotely. Because they comprise components of different nature and characteristics, many ORC components may not be suitable to be shared among environments.

If no equivalent component exists in the entire system its functionality must be recomposed through the combination of other components. Depending on the complexity of the specification of an atomic component, the corresponding autonomic manager may have to split it into simpler specifications to facilitate the search of equivalent components. In the case of a composite component, the autonomic manager must check the availability of the comprised local and remote components, and just find equivalent components for the unavailable bits of functionality. Although this is the simplest and quickest alternative, in some cases the combination of other distinct components may result in a more suitable composite entity. Hence, autonomic managers must implement a specification combination algorithm to successfully select the most appropriate components from all the partially equivalent specifications it receives. Some of the properties that must be taken into account to appropriately decide the best selection of components are e.g., the transferability nature of the components or the grade of equivalence among components.

For every specification resulted from the previous action an alternative equivalent component is searched in the local repository (step 2). If no one exists, the search is extended to other remote autonomic managers (step 3), which must check if any of the services they comprise matches the specification (step 4). Notice that in the example exposed in Figure 1 the autonomic manager of the second environment detects that three of its components have some partial equivalence with the unavailable component of the first environment, while the autonomic manager of the third environment discovers two partially equivalent components. These specifications are sent over the network to the autonomic manager of the first environment (step 5). Then, it must decide which combination of components is the most suitable one. In this case the autonomic manager of the first environment is interested in one component of the second environment and the two components of the third environment. After selecting the most suitable components, the autonomic manager of the first environment requests them to the autonomic managers of the corresponding environments (step 6). The components that satisfy some transference constrictions are sent over the network so that they can be invoked locally, while the rest must be invoked remotely (step 7).

Whether the substituting entity is atomic or com-

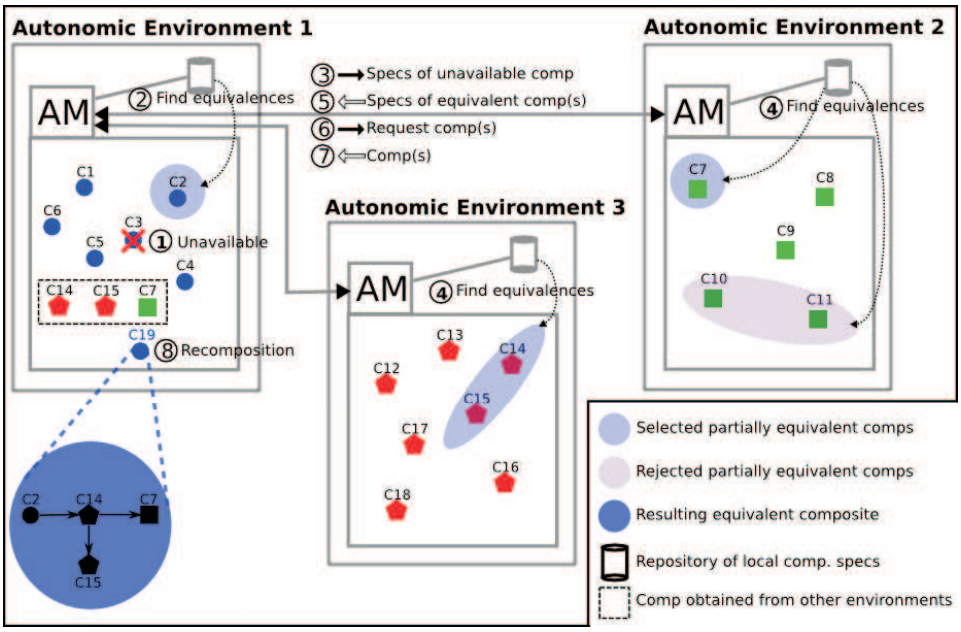

Figure 1: Functionality Recomposition for Self-Healing.

posite, once the corresponding autonomic manager has received all the required resources and resolves how they need to be composed to re establish the missing functionality, it must dynamically generate a new composite component and replace the failing one with it (step 8). ORC provides some orchestration operators and primitive functions to glue all the required components together.

Finally, the autonomic manager must modify the existing relationships that the local components had with the failing component, so that they can consume the service provided by the newly obtained structure. Similarly, other autonomic managers are then requested to change the relationships that any of their components may had with the unavailable component. Then, the execution of all the affected components is re-activated (these last two steps are not shown in Figure 1 for simplicity.)

5 CONCLUSIONS AND FUTURE WORK

In this paper we have introduced FReSH, a decentralised component-based framework currently under development that is able to dynamically detect unavailable components and recompose the functionality delivered they were delivering. Functionality recomposition is possible by making components shareable and reusable among the different nodes of the system. To be identified, these components must have some associated behavioural specifications that describe the functionality the component delivers.

However, to materialise this approach various tasks remain as future work. At the moment we are extending ORC to support autonomous and formally specified compositions out of the JML specifications associated to the atomic components implemented in Java, so that the resulting software structures satisfy the requirements for creating open systems (Nierstrasz and Meijler, 1994). A next step is producing sophisticated specification discovery, matching and combination algorithms to successfully obtain components allocated anywhere in the system and construct the most suitable equivalent structures to the failing software entities.

On another hand, we have to deal with the heterogeneity nature of the components we want FReSH to handle. In large-scale complex systems there may exist components implemented in different technologies, which enforces certain restrictions on the programming languages being used. Furthermore, we also need to figure out what is the most appropriate level of granularity of these components, as too fine-grained components may make FReSH an ineffective technique due to the unaffordable amount of time that composing new structures could take.

Finally, we also need to consider the case where a whole node crashes and therefore even its autonomic manager becomes unavailable. The underlying concepts of ultra-stable systems (Hariri et al., 2006) may support a solution for both component and node unavailability, and thus needs to be further investigated.

ACKNOWLEDGEMENTS

Special thanks to Joseph Kiniry, Emil Vassev and Dalibor Jaklin for all the discussions that helped shaping the underlying concepts of FReSH. This work is partially supported by Science Foundation Ireland under grant number 03/CE2/I303-1, *Lero, the Irish Software Engineering Research Centre*.

REFERENCES

Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., Silva, D. M. D., Krieger, O., Edelsohn, D. J., Auslander, M. A., Gamsa, B., Ganger, G. R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., and Xenidis, J. (2003). Enabling autonomic behavior in systems software with hot-swapping. *IBM Systems Journal*, 42(1).

Baker, S. and Dobson, S. (2005). Comparing service-oriented and distributed object architectures. In *OTM Conferences (1)*, pages 631–645.

Balasubramaniam, D., Morrison, R., Kirby, G., Mickan, K., Warboys, B., Robertson, I., Snowdon, B., Greenwood, R. M., and Seet, W. (2005). A software architecture approach for structuring autonomic systems. In *DEAS'05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA. ACM.

Baresi, L., Ghezzi, C., and Guinea, S. (2004). Towards self-healing service compositions. In *PriSE'04: First Conference on the Principles of Software Engineering*, volume 42, pages 27–46.

Berkeley/Stanford (2008). Recovery-Oriented Computing (ROC). http://roc.cs.berkeley.edu.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232.

Clemens Szyperski, Dominik Gruntz, S. M. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc.

de Lemos, R. and Fiadeiro, J. L. (2002). An architectural support for self-adaptive software for treating faults. In *WOSS'02: Proceedings of the first workshop on Self-healing systems*, pages 39–42, New York, NY, USA. ACM.

George, S., Evans, D., and Marchette, S. (2003). A biological programming model for self-healing. In *SSRS'03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pages 72–81, New York, NY, USA. ACM.

Georgiadis, I., Magee, J., and Kramer, J. (2002). Self-organising software architectures for distributed systems. In *WOSS'02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA. ACM.

Hariri, S., Khargharia, B., Chen, H., Yang, J., Zhang, Y., Parashar, M., and Liu, H. (2006). The autonomic computing paradigm. *Cluster Computing*, 9(1):5–17.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *IEEE Computer*, 36:41–50.

Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D. L., Sirin, E., and Srinivasan, N. (2007). Bringing semantics to web services with owl-s. *World Wide Web*, 10(3):243–277.

Misra, J. and Cook, W. (2007). Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6:83–110.

Nachira, F. (2007). Digital business ecosystems. http://www.digital-ecosystems.org/book/de-book2007.html.

Nagpal, R., Kondacs, A., and Chang, C. (2003). Programming methodology for biologically-inspired self-assembling systems. In *AAAI Spring Symposium on Computational Synthesis: From Basic Building Blocks to High Level Functionality*.

Nierstrasz, O. and Meijler, T. D. (1994). Requirements for a composition language. In *ECOOP'94: Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 147–161, London, UK. Springer-Verlag.

Papazoglou, M. P. and Georgakopoulos, D. (2003). Service-oriented computing. *Communications of the ACM*, 46(10):46–54.

Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. (2002). Recovery oriented computing (roc): Motivation, definition, techniques and case studies. Technical report, Berkeley, CA, USA.

Tanenbaum, A. S. and Steen, M. V. (2001). *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., Kephart, J. O., and White, S. R. (2004). A multi-agent systems approach to autonomic computing. In *AAMAS'04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 464–471, Washington, DC, USA. IEEE Computer Society.

van Lamsweerde, A. (2000). Formal specification: a roadmap. In *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA. ACM.

Zaremski, A. M. and Wing, J. M. (1995). Specification matching of software components. In *SIGSOFT'95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 6–17, New York, NY, USA. ACM.