Self-reconfiguration in Highly Available Pervasive Computing System

Hadi Hemmati[1] and Rasool Jalili[2]

[1] Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway
hemmati@simula.no
[2] Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
jalili@sharif.edu

Abstract. High availability of software systems is an essential requirement for pervasive computing environments. In such systems self-adaptation, using dynamic reconfiguration is also a key feature. However, dynamic reconfiguration potentially decreases the system availability by making parts of the system temporary frozen, especially during incomplete or faulty execution of the reconfiguration process. In this paper, we propose Assured Dynamic Reconfiguration Framework (ADRF), consisting of run-time analysis phases, assuring the desired correctness and completeness of dynamic reconfiguration process. We also specify factors that affect availability of reconfigurable software in pervasive computing systems. Observing the effects of these factors, we present availability improvement of our method in comparison to the other reconfiguration mechanisms.

Keywords: Dynamic Reconfiguration, Pervasive Computing, Autonomic Systems, Availability.

1 Introduction

Pervasive Computing Systems (PCSs) are going to change the focus of software systems from information and services to users. In such user-centric PCSs, availability and adaptability are parts of the software development fundamentals [1]. High availability of services in PCSs forces such systems to be self-adaptive. Self-adaptation or adaptability is the software ability to change its architecture behavior in the execution time whenever it is needed [2]. Changing software architecture in run-time without shutting the system down is called dynamic reconfiguration [3].

Our perception of dynamic reconfiguration covers all kinds of run-time changes on application in the level of software architecture such as upgrading, updating, bug fixing, and adapting to a new situation. Mainly, dynamic reconfiguration is performed to adapt a system to the new situation to improve system performance and software qualities. One of the most important quality attributes, which is necessary in distributed systems and much more in PCSs, is system availability. This is due to the facts that unavailable systems can not (1) be invisible from users for a long time (2) respond to user intent sufficiently (3) be trusted as secure systems and (4) be

C. Rong et al. (Eds.): ATC 2008, LNCS 5060, pp. 289–301, 2008.
© Springer-Verlag Berlin Heidelberg 2008

considered as dependable systems to be used anytime, anywhere, and from any device. Reconfigurable software in PCSs has the following features [4]:

- PCS software potentially has the ability to perform many reconfigurations in their life-time because of systems adaptiveness and reconfigurations context-awareness.
- In ordinary systems, reconfigurations are usually simple such as upgrading a component. But in PCSs, reconfigurations usually consist of several operations to adapt the system to completely new situations. Reconfigurations with several operations are called complex reconfiguration.
- Most of PCSs do not have any external administrator. This fact forces them to be self-managed. From this point of view, PCSs are similar to autonomic systems which need self-reconfigurablity.
- Wireless communication, device mobility, limited power, and other limited resources make pervasive computing environments error-prone. Therefore, the risk of failure during reconfiguration process in such environments is very high.

Hence, if there is not any mechanism for correct execution of reconfiguration process, dynamic reconfiguration decreases the system availability in the case of reconfiguration failure.

Although adaptability in PCSs has been discussed in many papers, the problem of run-time assurance for reconfiguration process has not been considered properly. Most of current run-time monitoring, validation, and verification techniques are at the code level [5, 6]. There exists some tools such as ArchStudio [7] and Mae [8] which manage dynamic reconfiguration but they do not have enough run-time analysis. In [9] replicated components have been used during reconfiguration, and after completion of reconfiguration process. This solution suffers from having an extensive overhead for changing all replicas of a component after a reconfiguration. In addition, the replicated component can not be used, when the old version is functionally wrong or not applicable.

To achieve the assured reconfiguration, we need some assurance analyses in the specification time and run-time. As the main focus of this paper is run-time analysis, we assume that all reconfiguration specifications are correct. Having a verified reconfiguration specification is not enough because of unexpected run-time errors and unsuitable reconfiguration starting time [10]. In this paper, we propose a run-time monitoring method in ADRF to ensure correct and complete execution of dynamic reconfiguration in PCSs. Also, we demonstrate how much the PCS availability can be improved by performing reconfiguration under ADRF supervision.

Section 2 introduces ADRF with its architecture and process. In sections 3 monitoring and analysis of reconfiguration in ADRF is demonstrated. Section 4 discusses some availability issues in ADRF, and section 5 evaluates ADRF in terms of system availability.

2 Assured Dynamic Reconfiguration Framework

Assured Dynamic Reconfiguration Framework (ADRF) is aimed to provide correct and complete reconfiguration in PCSs [11]. In addition, ADRF improves system

availability through reducing the risk of incomplete and faulty reconfigurations. In this section ADRF architecture and its reconfiguration process are explained.

2.1 ADRF Architecture

ADRF, illustrated in Figure 1, is located between the middleware and the user interface. It surrounds the application and monitors it in the reconfiguration period. ADRF rules can be updated through the user interface. Utilizing the middleware distribution facilities, ADRF can support distributed applications, which is out of the scope of this paper. Inside ADRF, there are three main components for providing assured reconfiguration process: A context-manager (CM), a reconfiguration-manager (RM), and a service-manager (SM). CM is responsible for triggering RM and the application when a related context changes. RM is responsible for performing assured reconfigurations when preconditions are triggered by CM. SM is responsible for preparing components and software for reconfiguring in a suitable manner by freezing and unfreezing some components.

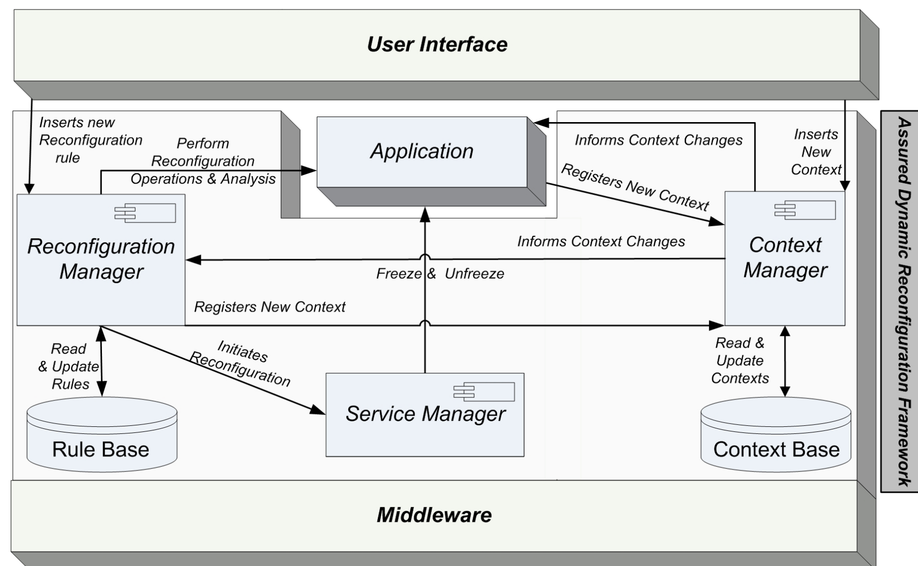

Fig. 1. The Architecture of ADRF

2.2 Reconfiguration Process in ARDF

In the ADRF component model, components are interconnected through messages and messages are buffered in the source and destination components. Therefore, connectors are just some pointers and do not have significant role in our view of the software architecture. A UML-like state-chart is used for specifying component behavior. Software configuration in ADRF is represented by a graph of components. The system behavior is characterized using the components behavior in addition to its architectural configuration. The reconfiguration process in ADRF consists of four steps:

1. Detecting the need for a reconfiguration (or reconfiguration initiation)
2. Selecting a reconfiguration map
3. Performing the reconfiguration map
4. Analyzing the architecture after reconfiguration

The first step starts when a change in the system or environment occurs which satisfies a reconfiguration's pre-conditions. These pre-conditions in PCSs are context-aware. It means that they are triggered by changes in the system, user, or environmental contexts. In fact, system designers or architects use these pre-conditions to define situations where system needs reconfiguring its architecture. The followings are some examples of these situations in PCSs: the need for tolerating faults, using new services, adapting to existing resources, automatic evolution, and supporting change in user intents.

In the second step the corresponding map for the pre-condition is fired. A reconfiguration map is a set of reconfiguration operations which should be performed sequentially. Each reconfiguration has a rule, which contain preconditions and corresponding map. In ADRF, reconfiguration rules are specified in the design time by the system designer or architect. Reconfiguration operations are:

- *Add*(C_i) which adds component C_i to an architecture,
- *Delete*(C_i) which deletes component C_i from an architecture,
- *Attach*(C_i,C_j) which attaches two components C_i and C_j to each other,
- *Detach*(C_i,C_j) which detaches two components C_i and C_j from each other, and
- *Replace*(C_i,C_j) which replaces two components C_i and C_j with each other.

The definition of reconfiguration rules can be defined in EBNF (Extended Backus-Naur Form) as:

- <RecRule> ::= <Precond>, <Map>
- <Precond> ::= <Cond> {Λ <Cond>}*
- <Cond> ::= **Context** <Op> **Context** | **Context** <Op> **Const**
- <Op> ::= < | > | <= | => | == | ≠ | in | not in
- <Map> ::= <RecOp>+
- <RecOP> ::= **Add(Ci)** | **Delete(Ci)** | **Attach(Ci, Cj)** | **Detach(Ci, Cj)** | **Replace(Ci, Cj)**

Where terminals are shown in bold and non-terminals are located between "<" and ">". The terminal *Context* can be one of the pre-defined contexts in the system or environment. The terminal *const* is a constant value. In ADRF, software components are attached together without specific connectors. Therefore, the connector role in the reconfiguration operations is omitted.

In the third step of the reconfiguration process, the reconfiguration map is performed by sequentially executing its reconfiguration operations on the software. Executing these operations, it is necessary to block (freeze) some parts of the software which are participated in the reconfiguration. It is due to the fact that in most cases components can not be reconfigured, when they are being executed through a running process. During freezing period, services which are provided by frozen components are not available. Freezing has two problems which should be solved in the reconfiguration process: finding the best time to freeze, and finding the minimum components which should be frozen.

After freezing, the reconfiguration operations are performed and then the frozen components are unfrozen. In the fourth step, some run-time analyses are carried out before unfreezing the modified software architecture to check its conformance with the architect anticipation.

3 Monitoring and Analysis of Reconfiguration Process in ARDF

Run-time assurance analysis in ADRF is performed in three phases: before, during, and after reconfiguration. In the initialization phase of ADRF, the context-aware application and each reconfiguration rule register themselves in CM. Each rule may include some contexts in its pre-condition. Such pre-conditions are registered in CM in the initialization phase as well as new rules insertion time. If all preconditions of a reconfiguration are satisfied, CM will trigger RM to fire the reconfiguration. In the following sub-sections, we explain the details of the three reconfiguration analysis phases.

3.1 Freezing the Affected Area

The first phase of analyzing a reconfiguration, which is done before reconfiguration execution, is freezing the affected area by SM. Affected area in a specific reconfiguration, is the set of components affected by the reconfiguration. It consists of all components which have been given as parameters to the reconfiguration operations. For example, if a reconfiguration attempts to replace c1 with c2; c1 should be frozen and added to the affected area of this reconfiguration. Unfrozen components continue their execution regardless of the frozen part. If a running component sends a message to one of the frozen components, the message will remain in the destination component buffer, until the component is unfrozen.

After recognizing the affected area, SM should find the best time to freeze. When components of the affected area are in their Safe Reconfiguration Points (SRPs) is the best time. SRPs are states in the component state-chart where the component state can be correctly transferred. In fact, components that are not in SRP states can not be reconfigured. Recognizing SRPs in the state-chart can not be done completely automatic due to the lack of some semantic information which should be given by the architect. In ADRF we assume that the architect specifies SRPs in the component state-chart. SRPs are defined per component without taking into account the difference between reconfigurations. Therefore, we need additional restriction on SRPs to find allowed starting states per reconfiguration. In ADRF this is done by a Transfer Function, which corresponds some SRPs of the component to new states after a specific reconfiguration. Transfer Function of a reconfiguration is given in a table called T-Table. This table is a list of following pair states <Permissible SRP from the reconfiguration point of view, Corresponding state after reconfiguration>.

In ADRF, each component is executed in a separate execution process and each user instantiates the component in a separate execution thread. A reconfiguration execution reaches to a break-point when all its threads are in their permissible SRPs regarding T-Table. When the freeze instruction is invoked, execution threads will be stopped by SM in the first break-point. If the affected area components can not reach

to a break-point in a defined time, the reconfiguration is regarded as unsafe and ADRF will reject it.

3.2 Structural Analysis

The second assurance analysis is structural correctness checking after performing the reconfiguration. In ADRF this analysis is done by Assurance Automata. The automaton is created during reconfiguration to model the intermediate architectures, from the initial to the expected target architecture. In Assurance Automata, each state represents the anticipated architectures during reconfiguration. ADRF continually monitors current system configuration and compares it with the states of Assurance Automata. There are some techniques and methods for capturing the current system architecture such as [12]. Assurance Automata is defined more formally as $(S, S', \delta, F, \Sigma)$ where:

$\sum$ is the automaton alphabet and includes reconfiguration operations:

$\sum$ = {Add(C_i), Delete(C_i), Attach(C_i, C_j), Detach(C_i, C_j), Replace(C_i, C_j)} $\cup$ {Er, Hld}, Where Er indicates the incorrect execution and Hld shows the unexecuted operation.

S_i represents a configuration (valid or invalid) of an architecure. The configuration is shown by G(V,E). G is a directed graph, where its nodes are the architecture components and its edges are connectores (links between attached components).

δ is the transition function defined by either correct execution of an operation (destination: the *next* state) or incorrect execution in the case of run-time errors (destination: *other* states or one *trap* state in online and offline methods respectively) or unexecution, for any reason (destination: the *current* state).

S' is the initial state, equivalent to the system architecture just before reconfiguration.

F is the final state, equivalent to the target architecture.

Structural analysis by Assurance Automata can be done in offline or online methods:

Offline Method: In the offline method, a snapshot of the system is captured and then reconfiguration starts. After a predefined time, which depends on the number of reconfiguration operations, RM compares the system state (current configuration) with the target state in Assurance Automata. The reconfiguration execution is structurally correct if those states match. Otherwise, the system state is compared to the all intermediate states in Assurance Automata in the reverse order, until finding an equal state. Afterward, the reconfiguration is re-executed from the discovered state with remained operations. If none of the states are equal to the system state, the system is in a *trap* state and it should be recovered from *initial* state, which is stored in the captured snapshot, and then the reconfiguration is re-executed.

In this method, besides the timeout, the number of executed operations is another stopping criterion for reconfiguration process. RM restricts the number of performed operations to the number specified in the map.

Online Method: In the online method, when the expected time to execute an operation is passed, RM compares the current system state with the expected state in Assurance Automata. The expected states represent correct execution of each

reconfiguration operations. If the system state is equal to the *next* state of the automaton, the execution has been performed correctly. If the system state is not equal to the *next* state, but equal to the “before transition” (*current*) state, the last operation must be re-executed. If the system state is not equal to either the *next* or *current* state, system has gone to the *other* state. In this case, system must be recovered from the *current* state and then the last operation should be re-executed.

The main advantage of this monitoring and control mechanism is online error detection that is suitable when some repair mechanism is available.

3.3 Behavioral Analysis

The behavioral analysis is the third phase in assurance analysis which is performed at the end of the reconfiguration process and before unfreezing the affected area. RM checks component states which should match with the T-Table information. If Assurance Automata passes the reconfiguration but a component is found in the affected area which is not in its expected state, the behavioral assurance is not satisfied and the state transfer should be repeated. In ADRF, the current state transfer algorithm is simple but can be replaced without any change in the core of the framework.

Finally, if the three assurance analysis phases are passed successfully, the reconfiguration process is regarded as assured and SM can unfreeze the affected area.

4 Availability Issues of Reconfigurable Software in PCSs

The term availability is defined as the ratio of the total time a functional unit is capable of being used during a given interval to the length of the interval. The most simple representation for availability is as a ratio of the expected value of the uptime of a system to the aggregate of the expected values of up and down time, or *MTTF/(MTTF+MTTR)*. Where MTTF is the mean time to failure and MTTR declares the mean time to repair.

Although a successful reconfiguration can improve system availability by 1) replacing faulty components with the debugged version and 2) adding extra components to reply requests of overloaded components, but it has the possibility of freezing some components at run-time, causing them to be unavailable for a while. Replication of components seems to be a solution. However, it has problems such as the overhead of reconfiguration of all replicas. In addition, in cases where the new component functionality is not valid anymore, replicated components are inapplicable. ADRF tries to minimize the mentioned unavailability time of the affected area components.

4.1 Availability Definition in Reconfigurable Software

To define the system availability in ADRF, we assume the importance of all services in the system is the same and freezing a component results in unavailability of only that component services. Accordingly, we can define the system availability as the simple average (instead of weighted) of its services or components availability. The availability of each component itself is the average of all its instances availability.

Component instances are instantiated from a base component for each user session where the component is invoked. Reconfiguration process is performed on the base components. By reconfiguring a base component all its instances should also be reconfigured accordingly. When all instances of a component are frozen, the component is frozen and ready to be reconfigured.

Putting all together, the system availability is the average availability of all system component instances. Let call j^{th} instance of ith component, C_{ij}, so in a system having n components and m_i instances for each component C_i, the system availability is defined as equation 1.a.

The availability of a component instance is equivalent to its up time (CIUT) divided by its life time (CILT). CILT is the time between the instantiation of an instance and its destruction. With respect to the reconfiguration process, CIUT is a part of CILT that the component instance is not frozen, multiply by α . α is the component's normal availability without considering the reconfiguration process. The freeze time of a component instance depends on the number of reconfigurations performed on the instance during its life time (p) and the instance freeze time during each reconfiguration (CIFT(R_k)). CIUT is obtained by subtracting the sum of all freeze times of an instance from the instance life time, multiplying by α . The system availability is obtained by the average value of Availability(C_{ij}) for all component instances in the system (replacing Availability(C_{ij}) in equation 1.a by Availability(C_{ij}) from equation 1.b).

$$\text{a) } \text{Availability(system)} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m_i} \text{Availability}(C_{ij})}{\sum_{i=1}^{n} m_i} \tag{1}$$

$$\text{b) } \text{Availability}(C_{ij}) = \frac{\text{CIUT}}{\text{CILT}} = \frac{\alpha * (\text{CILT} - \sum_{k=1}^{p} \text{CIFT}(R_k))}{\text{CILT}}$$

4.2 Availability Factors for Reconfigurable Software in PCSs

We extracted factors that affect the system availability based on the above discussion. The effective factors are defined as follows (concentrating on the reconfiguration effects on availability, we assume that CILT and α are constants):

- **Number of reconfigurations:** As the number of reconfigurations is increased, the component freeze time is increased. Therefore, the component availability and consequently the system availability are decreased.
- **Number of involved components:** Since all involved components should be frozen during reconfiguration, the more components involved in a reconfiguration the less system availability. If a component instance does not participate in any reconfiguration during its life time, its availability has the

maximum value (α). For each participation, an unavailability time (CIFT) is added to the components down time and so decreases the system availability.

- **Number of users:** If the number of system users is increased, the number of component instances involved in the reconfiguration is increased. Therefore, available instances and the system availability will be decreased.
- **Number of reconfiguration operations:** The number of reconfiguration operations directly affects the total execution time of the reconfiguration process. Accordingly, long reconfigurations (including many operations) decrease system availability.
- **Error Rate:** The more errors occurrence the more validity checks and recovery done.

5 Availability Evaluation in ADRF

In our study, a simple PCS simulator was implemented to fill the absence of a real pervasive system. The simulator takes an XML file describing a context-aware application through its architectural component-diagram plus the state-chart of each component. Contexts can be changed randomly in the simulator. A sample of context is location and its change demonstrates the user mobility. Application execution is simulated by transferring messages among components. A prototype of ADRF has also been embedded in the PCS simulator implementation in order to manage reconfiguring the applications running in the simulator.

To evaluate ADRF and its impacts on availability, a smart library case study has been studied. Smart library provides a map-based guidance to books and collections on a Smart Digital Assistants. Main components in the system architecture which are distributed in the environment servers, gadgets, and user mobile devices are User Profile Manager, User Interface, Library Books Manager, Search Engine, Positioning Engine, Location Manager, Path Finder, and Map Manager. The reconfiguration which is used here is replacing Location Manager (LM) component with a new location manager (ULM) which supports updating the user current position while he is walking towards a selected rack. The study focuses on comparing availability of the system when using one of the following reconfiguration mechanisms:

- **BASE:** This mechanism ignores occurrence of errors. The system administrator is responsible for recovering the system and re-executing the reconfiguration. Involving the external human administrator in the reconfiguration process makes this mechanism unsuitable for PCSs. Evaluating this mechanism, the average recovery time by the external administrator is added to the unavailability time of all involved instances in the unsuccessful reconfiguration.
- **OPTIMISTIC:** This mechanism uses offline method of ADRF for structurally analyzing the reconfiguration process. If the system falls into an error state, it is automatically recovered after reconfiguration process and repeats the reconfiguration with the hope of correct execution. In this approach, capturing the system snapshot is done in the background while the system is available, but automatic recovery time is considered as a part of component's downtime.

- **ADRF:** This mechanism uses online method of ADRF in the structural analysis phase. The check and repair time for each operation, is important in the online method. In ADRF, each reconfiguration operation should be performed correctly. In the case of any error in the operation execution, it should be detected and repaired on-line. The smaller check and repair time, the quicker reconfiguration process and more available systems.

Our first observation is the effect of *number of users* on system availability. The number of library users in this case varies from 1 to 36. As shown in Figure 2.a, the availability in BASE mechanism is decreased more rapidly due to its long external recovery time. While the system availability in ADRF and OPTIMISTIC mechanisms are close to each other, ADRF provides more availability as the number of users increases. The reason is dependence between the number of users and the number of component instances. Therefore, OPTIMISTIC mechanism makes systems with many users more unavailable.

The next experiment focuses on the number of reconfigurations. In this case, another reconfiguration which replaces ULM with LM is defined. These two reconfigurations are applied repeatedly on the software architecture by required context changes. Figure 2.b depicts the effect of varying the number of reconfigurations on the system availability. As shown, decline of the system availability in the BASE mechanism is very sharp, because of the huge external overhead per reconfiguration. As the number of reconfigurations increases, the re-execution overhead in the OPTIMISTIC mechanism results in less availability in comparison to the ADRF mechanism. This experiment recommends ADRF for adaptive context-aware systems which have many reconfigurations during their life-cycle.

The effect of the number of reconfiguration operations on the system availability is evaluated, as the next experiment. We increase the number of operations by replacing more components and adding new components. As depicted in Figure 2.c, by increasing the number of reconfiguration operations, the availability of ADRF against BASE and OPTIMISTIC mechanism is less decreased. The BASE mechanism is the worst, because of the higher chance of failure (when the number of operations increases) as well as the high external recovery time. For long reconfigurations OPTIMISTIC approach decreases the system availability more than ADRF, because of re-executing the reconfiguration process from the beginning.

Our concentration in Figure 2.d, is the effect of error rate on availability. Generally, when a fault happens it means that an error has happened before. This error could be in communication links, computation, storage, or anywhere else. We assume no difference between errors. Therefore, error rate is assumed as the rate of fault occurrence. By changing the average error rate between 0.05 and 0.5, the difference among availabilities gained in the three mechanisms is specified in error-prone environments. As expected, the OPTIMISTIC mechanism is not suitable in such environments even worse than the BASE mechanism because of its optimistic view on error occurrence and its huge re-execution overhead. According to Figure 2.d, when the risk of falling into error states in each operation execution is high, online detection and repair in ADRF is the best.

Based on the level of decline on availability, the number of reconfigurations is the most effective factor. Therefore, the mechanism tolerating this effect is more suitable for PCSs. Our above-mentioned evaluations determined that ADRF is an appropriate

framework for reconfiguration in PCSs especially when the environment is error-prone, the software is complex, context-aware, very adaptive with long reconfigurations, and lots of users. This is because of ADRF assurance mechanism which provides not only the correct and complete reconfiguration but also a highly available reconfiguration process, in comparison to performing reconfigurations without any assurance checks or simple offline optimistic validation mechanisms. Additionally, ADRF demonstrates itself scalable in terms of the number of users, reconfiguration operations, and the error rate.

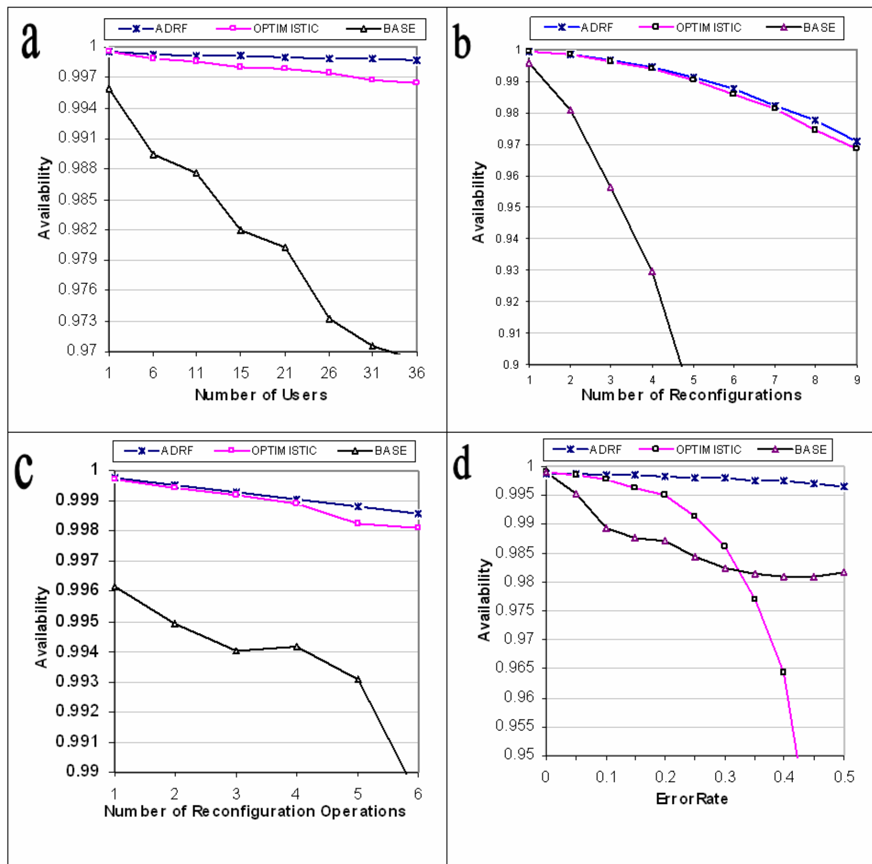

Fig. 2. The Effect of a) Number of Users, b) Number of Reconfigurations, c) Number of Reconfiguration Operations, and d) Error Rate on system availability

6 Conclusions and Future Work

Software reconfiguration will play an important role in the future computing environments. Most research on this domain and especially on the reconfiguration in PCSs are restricted to finding the best change strategies. However, applying these

reconfiguration strategies in a running system has problems such as reconfiguration failure which affects service availability. Without monitoring and validating the reconfiguration process at run-time, system invisibility and adaptability can be damaged.

In this paper, we proposed an Assured Dynamic Reconfiguration Framework, ADRF, capable of performing run-time reconfiguration on PCSs. Achieving three assurance analysis phases (before, during, and after reconfiguration process), ADRF ensures architect that his defined reconfigurations will be performed correctly and completely. In addition, we defined the system availability with respect to reconfiguration process and identify the effective factors on the PCSs availability. We evaluated our developed framework, which uses an online assurance mechanism in the reconfiguration process, based on the defined factors. Results confirmed that, our framework provides more system availability especially for complex PCSs in error-prone environments which perform long reconfigurations.

As future work, effects of other factors on reconfigurable software availability can be investigated. Enhancing ADRF to perform secure and dependable reconfiguration is also among the other topics of interest for future research.

References

1. Saha, D.: Pervasive Computing: A Paradigm for the 21st Century. IEEE Computer Society, Los Alamitos (2003)
2. Cheng, S., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P., Hu, N.: Software Architecture-based Adaptation for Pervasive Systems. In: Schmeck, H., Ungerer, T., Wolf, L. (eds.) ARCS 2002. LNCS, vol. 2299, Springer, Heidelberg (2002)
3. Oreizy, P., Taylor, R.N.: on the role of software architectures in runtime system reconfiguration. In: International Conference on Configurable Distributed Systems (1998)
4. Hemmati, H., Aliakbarian, S., Niamanesh, M., Jalili, R.: Structural and Behavioral Run-Time Validation of Dynamic Reconfiguration in Pervasive Computing Environments. In: 4th Asian International Mobile Computing Conference (AMOC), Calcutta, India (2006)
5. Nicoara, A., Alonso, G.: Dynamic AOP with PROSE. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, Springer, Heidelberg (2005)
6. Chen, F., Rosu, G.: Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. Electronic Notes in Theoretical Computer Science, vol. 89. Elsevier, Amsterdam (2003)
7. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-Based Runtime Software Evolution. In: The 20th International Conference on Software Engineering (ICSE 1998), Kyoto, Japan, pp. 177–186 (April 1998)
8. Roshandel, R., Hoek, A.V., Mikic, M., Medvidovic, N.: Mae – A System Model and Environment for Managing Architectural Evaluation. ACM Transactions on Software Engineering and Methodology (April 2004)
9. Diaconescu, A., Murphy, J.: A Framework for Using Component Redundancy for self-Optimising and self-Healing Component Based Systems. In: ICSE 2003 Workshop on Software Architectures for Dependable Systems, Portland, Oregon, USA (May 3 2003)
10. Niamanesh, M., Jalili, R.: A Dynamic-Reconfigurable Architecture for Protocol Stacks of Networked Systems. In: 31st Annual International Computer Software and Applications Conference, Beijing, China (July 2007)

11. Hemmati, H., Niamanesh, M., Jalili, R.: A Framework to Support Run-Time Assured Dynamic Reconfiguration for Pervasive Computing Environments. In: The first IEEE International symposium on wireless pervasive computing ISWPC, Thailand (2006)
12. Hamou-Lhadj, A., Braun, E., Amyot, D., Lethbridge, T.: Recovering Behavioral Design Models from Execution Traces. In: Ninth European Conference on Software Maintenance and Reengineering (CSMR 2005), pp. 112–121 (2005)