Contribution of the Formal Verification of Self-adaptive Distributed Systems: A Case Study

Muhammad Nauman, Nadeem Akhtar
The Islamia University of Bahawalpur
Pakistan
{mhmmd.nauman, nadeem.akhtar.phd}@gmail.com

ABSTRACT: *Self-adaptive systems have been proposed as an effective approach to tackle the challenges associated with the engineering and management of modern-day complex distributed software systems. There has been a substantial enhancement in the approaches to build such systems that are designed, organized, deployed and formally verified for previous decade; there is still disagreement among the researchers on some of the essential fundamental concepts. In this paper, we present a case study in which we use model checking to verify behavioural properties of a decentralized self-adaptive system. Concretely, we contribute with a formalized architecture model of a Distributed Internet Marketing System and prove a number of self-adaptation properties for flexibility and robustness. To model the main processes in the system we use timed automata, and for the specification of the required properties we use timed computation tree logic. We use the Uppaal tool to specify the system and verify the safety property.*

Keywords: Self-adaptive Systems, Software Verification, Formal Verification, Model Checking, Distributed Software Systems

Received: 1 March 2014, Revised 8 April 2014, Accepted 14 April 2014

1. Introduction

With the emergence of new technologies for developing, organizing deploying and verifying the ever-increasing complex software systems, there is a need for mechanisms that simplify the development and formal verification of self-adaptive systems. However, the formal verification of such systems has been shown to be significantly more challenging than static and conventional software systems.

One important challenge in self-adaptive systems, in particular those with decentralized control of adaptation, is to provide guarantees about the required runtime quality properties. Researchers have defined formally founded design models for decentralized self-adaptive systems that cover structural aspects of self-adaptation [1]. These models support engineers with reasoning about structural properties, such as types and interface relations of different parts of the decentralized system. However, in order to provide guarantees about qualities, we need to complement this work with an approach to validate behavioral properties of decentralized self-adaptive systems. The need for research on formal verification of behavioral properties of self-adaptive systems is broadly recognized by the community [2] [3]. We have verified the safety and flexibility properties of a Distributed Internet Marketing System Using the Uppaal Model Checker tool.

With flexibility we refer to the ability of the system to adapt dynamically with changing conditions in the environment, and robustness is the ability of the system to cope autonomously with errors during execution. We model the main system processes with timed automata and specify the required properties using timed computation tree logic (TCTL).We use the Uppaal tool that offers an integrated environment for modelling, simulation and verification, based on automata and a subset of TCTL.

The remainder of this paper is structured as follows. In Section 2, we introduce the Distributed Internet Marketing System and explain a number of adaptation scenarios. In Section 3, we give a brief background on formal modelling. In Section 4 we provide introduction to Uppaal and the properties backgrounds that can be verified using this tool. Section 5 presents the design model of the distributed internet marketing system, and how we verified key properties and discusses potential uses of the study results both as input for model based testing and as a starting point for the definition of a reusable behaviour model for self-adaptive systems. We discuss related work in Section 6, and conclude with a summary and challenges ahead in Section 7.

2. State of the Art

We discuss related work on formal modelling of self-adaptation in three parts: fault-tolerance and self-repair, verification of various properties, and integrated approaches. We conclude with a brief discussion of the position of the work presented in this paper.

2.1 Fault-tolerance and Self-repair [5]

Introduces an approach to create formal models for the behaviour of adaptive programs. The authors combine Petri Nets modelling with LTL for property checking, including correctness of adaptations and robustness properties. [6] presents a case study in formal modelling and verification of a robotic system with self-x properties that deal with failures and changing goals. The system is modelled as transition automata and correctness is checked using LTL and CTL (computational tree logic). [7] outlines an approach for modelling and analysing fault tolerance and self-adaptive mechanisms in distributed systems. The authors use modal action logic formalism, augmented with deontic operators, to describe normal and abnormal behaviour. [8] models a program as a transition system, and present an approach that ensures that, once faults occur, the fault-intolerant program is upgraded to its fault-tolerant version at run-time.

2.2 Integrated Approaches [13]

Uses architectural constraints specified in Alloy as the basis for the specification, design and implementation of self-adaptive architectures for distributed systems. [14] proposes a model-based framework for developing robotic systems, with a focus on performance and failure handling.

The systems behaviour is modelled as hybrid automata, and a dedicated language is proposed to specify reconfiguration requirements. The K-Components Framework [15] offers an integrated design approach for decentralized self-adaption in which the system's architecture is represented as a graph.

A configuration manager monitors events, plans the adaptations, validates them, rewrites the graph and adapts the underlying system. [16] presents the PLASTIC approach that supports context-aware adaptive services. PLASTIC uses Chameleon, a formal framework for adaptive Java applications.

2.3 Various Properties [9]

It Presents a verification technique for multi-agent systems from the mechatronic domain that exploits locality. The approach is based on graph, and graph transformations, and safety properties of the system are encoded as inductive invariants. [10] PobSAM is a flexible actor-based model that uses policies to control and adapt the system behavior. The authors use actor-based language Rebeca to check correctness and stability of adaptations. [11] presents a formal verification procedure to check for correct component refinements, which preserves properties verified for the abstract protocol definition. A reachability analysis is performed using timed story charts. [12] considers self-adaptive systems as a subclass of reactive systems. CSP (Communicating Sequential Processes) is used for the specification, verification and implementation of self-adaptive systems.

3. Problem Statement

Self-Adaptive software systems are increasingly built in practice. There has been much research in the area in recent years and a number of contributions have already been produced. Despite these achievements, much remains to be done to support the development of predictable self-adaptive software via a formal, systematic and disciplined approach. We are elaborating on the main research areas in which significant work is required to better integrate formal verification techniques into software adaptation.

4. Formal Methods and Verification

Formal methods are based on solid mathematical foundations. Formal verification is the act of proving or disproving the correctness of underlying system algorithms with respect to certain formal specifications using formal methods of mathematics.

5. Model Checking

Model checking is a verification procedure that has been used for hardware verification of industrial applications and verification of communications protocol specifications [4]. All the possible states of the systems and execution paths are examined in an organized and exhaustive way in order to check if one or more properties hold. Model Checking can lead to state space explosion problem. The goal of model checking is to improve the quality of verification and validation (V&V) by specifying and checking properties that wrap all requirements of the application.

6. Uppaal

Uppaal is a model-checking tool for verification of behavioural properties. In Uppaal, a system is modelled as a network of timed automata, called processes. A timed automaton is a finite-state machine extended with clock variables. A clock variable evaluates to a real number, and clocks progress synchronously. It is important to note that fulfilled constraints for the clock values only enable state transitions but do not force them to be taken. A process is an instance of a parameterized template. A template can have local declared variables, functions, and labeled locations. The syntax to declare functions is similar to that of the C language. State of the system is defined by locations of the automata, clocks, and variables values.

Uppaal uses a subset of TCTL for defining requirements, called the query language. The query language consists of state formulae and path formulae. State formulae describe individual states with regular expressions such as $x >= 0$. State formulae can also be used to test whether a process is in a given location, e.g., Proc.loc, where Proc is a process and loc is a location. Path formulae quantify over paths of the model and can be classified into reachability, safety, and liveness properties:

• **Reachability** properties are used to check whether a given state formula f can be satisfied by some reachable state. The syntax for writing this property is *E <> f*.

• **Safety** properties are used to verify that "*something bad will never happen*". There are two path formulae for checking safety properties. *A* [] f expresses that a given state formula f should be true in all reachable states, and *E* [] f means that there should exist a path that is either infinite, or the last state has no outgoing transitions, called maximal path, such that f is always true.

• **Liveness** properties are used to verify that something eventually will hold, which is expressed as *A <> f*.

Processes communicate with each other *thro*ugh channels. Binary channels are declared as chan *x*. The sender *x*! can synchronize with the receiver *x*? through an edge. If there are multiple receivers *x*? then a single receiver will be chosen non-deterministically. The sender *x*! will be blocked if there is no receiver. Broadcast channels are declared as broadcast chan *x*. The syntax for sender *x*! and receiver *x*? is the same as for binary channels. However, a broadcast channel sends a signal to all the receivers, and if there is no receiver, the sender will not be blocked. Uppaal also supports arrays of channels. The syntax to declare them is chan *x* [*N*] or broadcast chan *x* [*N*], and sending and receiving signals are specified as *x* [*id*]! and *x* [*id*]?. Note that processes cannot pass data through signals. If a process wants to send data to another process then the sender has to put the data in a shared variable before sending a signal and the receiver will get the data from shared variable after receiving the signal.

Uppaal offers a graphical user interface (GUI) and model checking engine. The GUI consists of three parts: the editor, the simulator and the verifier. The editor is used to create the templates. The simulator is similar to a debugger, which can be used to manually run the system, showing running process, their current location and the values of the variables and clocks. The verifier is used to check properties of the model as described above.

7. Distributed Internet Marketing System

7.1 Introduction

The Distributed Internet Marketing System (DIMS) is a client/server system for internet marketers to provide online business marketing services and products. DIMS is designed to helps people in becoming online entrepreneurs. The main challenges for this system are

• Inform the servers with the dynamic changes in the system.

• Realize this functionality in a distributed way, avoiding the bottleneck of a centralized control centre.

• Make the system robust to Server failures

DIMS consist of Servers and Web Clients distributed across the world. Web-clients present a an administrations interface to the marketers to create marketing pages to market their services and collect the viewer name, email and phone tahrough an interface.

Once the internet user browses the form at web-client, web-client sends that information to server which stores it into the database. There is an administration interface at the servers from where systems administrators can use that information and provide more information about the service to lead.

The system consists of a set of intelligent web clients, which are distributed over the internet. Each we client is connected to a server. Web clients send the lead information to server so that the further information can be processed. To form a decentralized solution, web clients collaborate in organizations: if one of the servers goes down, they form an organization by creating a server from themselves that provides information to other web clients that was connected to the server so that they can join the organization. The first scenario concerns the dynamic adaptation of the configuring clients to a new server, when one of the servers goes down, here systems shifts all the clients connected to down server to the nearby server. The second scenario concerns when the other nearby server goes down too, now one of the clients behaves as a server and all the clients are connected to it until system finds a server up again.

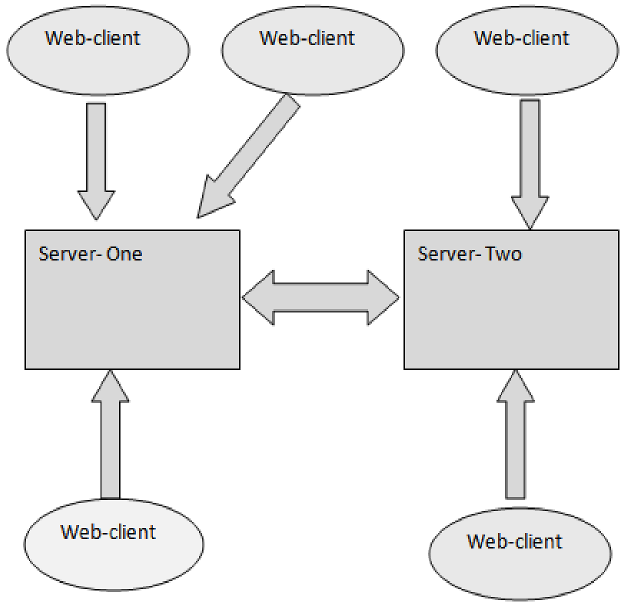

Figure 1. distributed internet marketing system

7.2 Scenarios

7.2.1 Dynamic Client-Server Organization for Flexibility

Figure 1 shows the primary components of the software deployed for internet marketing system.

The local monitoring system provides the functionality to detect connection down and inform other clients. The local distrib

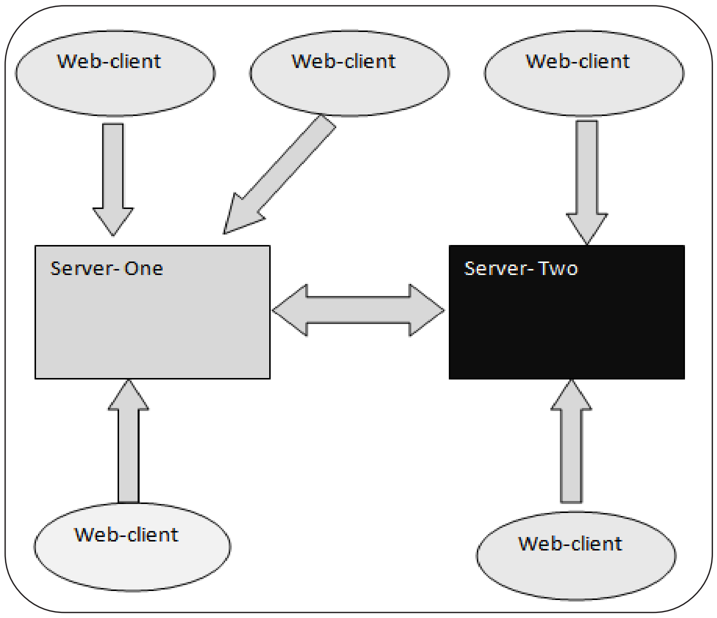

Figure 3. DIMS with self-adaptive layer

uted internet marketing system is conceived as an system consisting of two components. The client is responsible for monitoring the server connection and collaborating with other clients to report a possible server down to other clients. The organization middleware offers life cycle management services to set up and maintain organizations. We employ dynamic organizations of clients to support flexibility in the system, that is, client's organizations dynamically adapt them with changing system conditions.

To access the hardware and communication facilities on the clients, the local monitoring system can rely on the services provided by the distributed communication and host infrastructure. In normal system conditions, each client is connected to a nearby server. However, when a server down is detected that self management system tries to connect it with other server.

7.2.2 Self-Healing Sub-system for Robustness

To recover from server failures, a self-healing sub-system is added to the server and client system. The self-healing subsystem maintains a model of the current dependencies of the components of the local monitoring system with other active client servers. To recover from system failures, the subsystem contains repair actions for failure scenarios in different roles. Examples of actions are: halt the communication with the failed server, elect a new server, and exchange the current monitored system state with other clients. To detect failures, the self-healing subsystem clients in the dependency model using a ping-echo mechanism. Clients send periodically ping messages to other clients and a failure is detected when a client does not respond with an echo after a certain time

8. Model Design in Uppaal

Each server has two states. In normal operation, the server can be in working state. Additionally, the camera can be in the ServerDown state, representing the status of the server after a silent node failure. There is an instance with a unique id for each server.

Web-client has only one state and it uses Sendlead channel to synchronies on servers.

We have verified only the lead process from client to server using Uppaal Model Checker for safety property.

Self-Healing Controller is used to detect failures of other servers based on a ping-echo mechanism. A self-healing controller process runs on each server. The self-healing controller sends periodically isLive[ping] signals to the self-healing controllers of the dependent servers. If a server does not respond in a certain time it adapts the organizational controller.

Figure 4 and Figure 5 shows web-client template and server template for the Uppaal model checker.

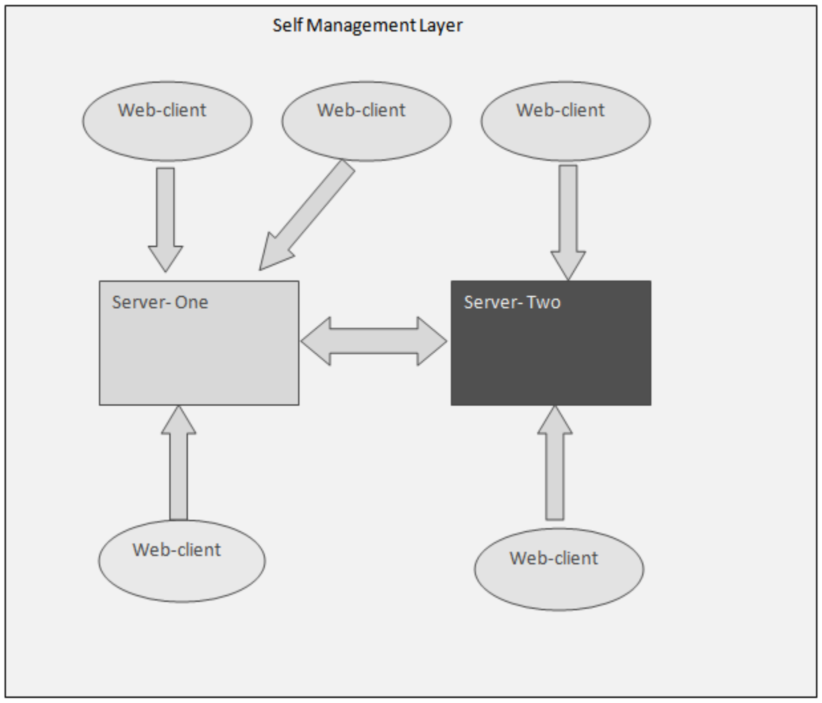

Figure 3. DIMS with self-adaptive layer

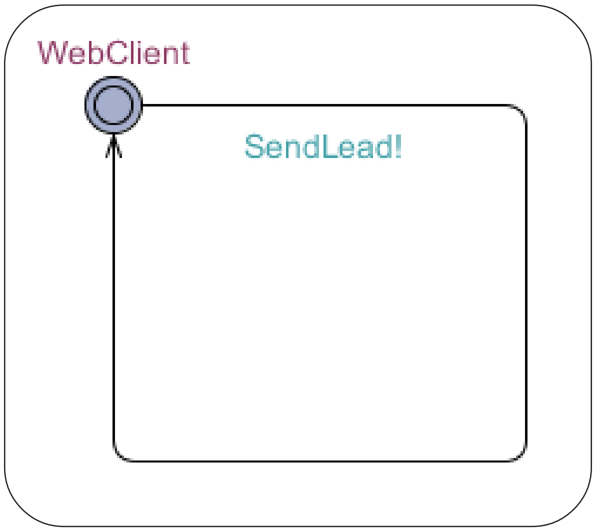

Figure 4. web-client template

There are three web-clients sending leads to the two server. We have created a channel SendLead! to verify the safety property. We have run the Uppaal simulator for 3 concurrent web clients sending leads to the server. As soon as web-client initiates the send lead process the model verifies that the server receive the leads at his end.

8.1 Model Verification

The following properties are verified:

1. A [] not deadlock

This statement verifies that system will not be in a deadlock at any time.

2. A <> SelfHealingController (0). Failed imply SelfHealingController (0). FailureDetected

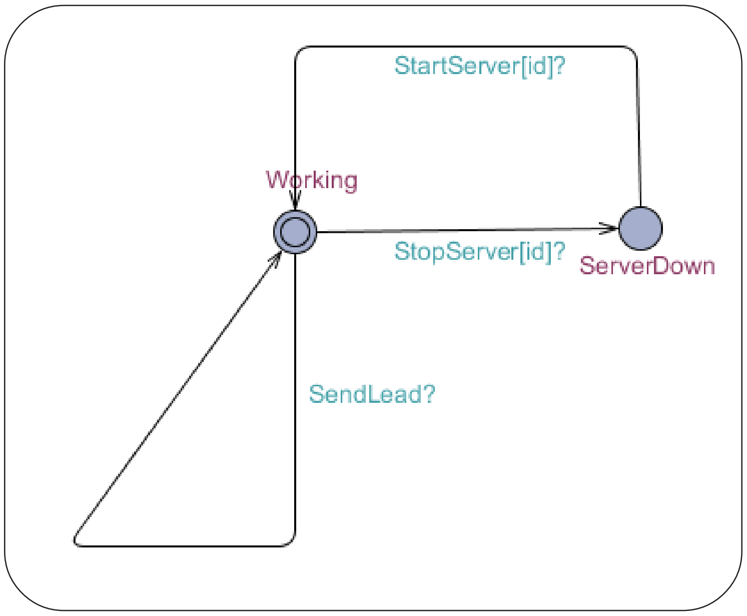

Figure 5. Server template

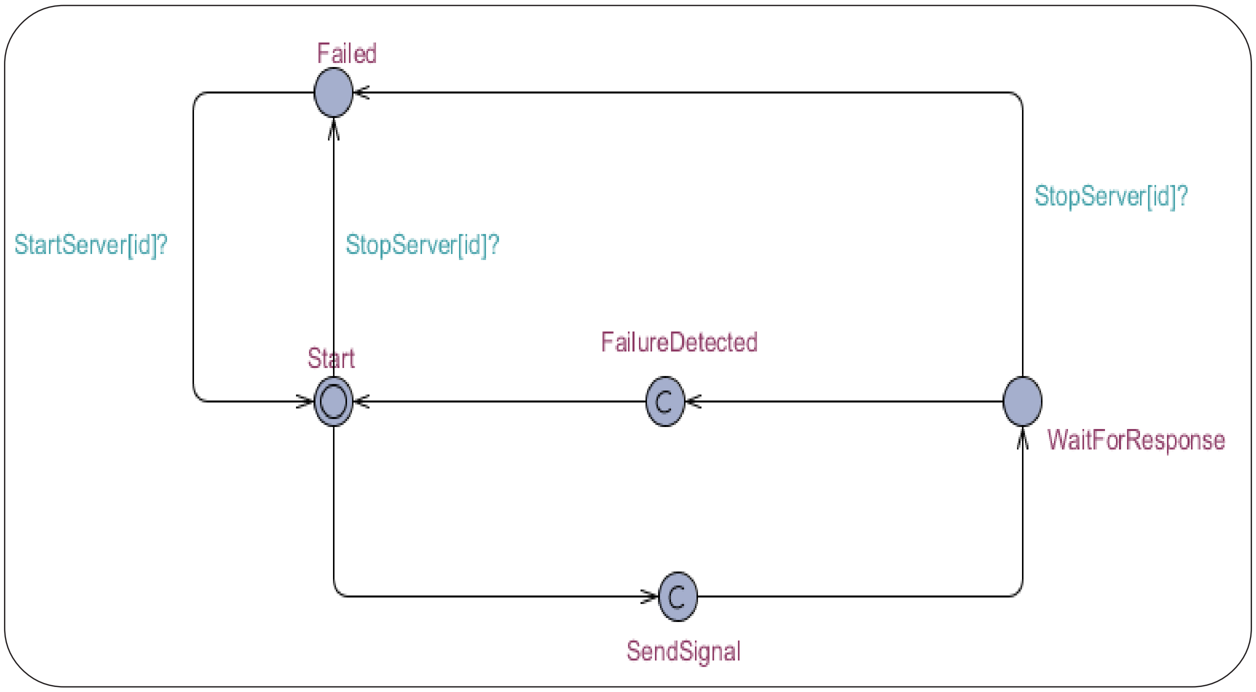

Figure 6. Self healing controller template

This statement verifies self-healing controller will detect server failures.

3. E <> Server (1). Working

This statement verifies the liveness property of the system.

4. E <> Server (1). ServerDown imply Server (1). Working

This statement verifies that all servers cannot be down at the same time.

```
const int N = 3;          //# Web Client
typedef  int [0, N –1] client_id;
chan SendLead;

const int C = 2;    // # Server
typedef  int [0, C – 1] server_id;

broadcast chan ServerGone [C];

//Self-Healing Subsystem channels
broadcast chan isLive [N], imLive [N];

broadcast  chan  networkJoined;

broadcast chan StartServer [C], StopServer [C], NetworkJoined;
```

7. Conclusion and Challenges Ahead

In this paper, we presented a case study on formal modelling and verification of a distributed internet marketing self-adaptive software system. The Uppaal tool allowed us to model the system and verify the required flexibility and safety properties. We defined a dedicated environment model both to verify specific adaptation scenarios and manage the state space problem. This work fits in our long term research objective to develop an integrated approach for formal analysis of distributed self-adaptive software systems that combines verification of architectural models with model-based testing of applications to guarantee the required runtime qualities. Our next goal is to verify the safety property for server down scenarios. As distributed internet marketing system have more than one servers so our next target is to verify the systems self management behaviors for server down cases.

References

[1] Weyns, D., Malek, S., Andersson, J. (2012). FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7 (1).

[2] Magee, J., Maibaum, T. (2006). Towards Specification, Modelling and Analysis of Fault Tolerance in Self Managed Systems. *In*: Self-adaptation and Self-managing Systems, ACM.

[3] Vassev, E., Hinchey, M. (2009). ASSL: A Software Engineering Approach to Autonomic Computing. *Computer*, 42 (6) 90–93.

[4] Markosian, L., Mansouri-Samani, M., Mehlitz, P., Pressburger. T. (2007). Program model checking using Design-for-Verification: NASA flight software case study. *In*: Proceedings of the 2007 IEEE Aerospace Conference.

[5] Zhang, J., Cheng, B. (2006). Model-based development of dynamically adaptive software. *In*: 28th International Conference on Software Engineering, ACM.

[6] Gudemann, M., Ortmeier, F., Reif, W. (2006). Formal Modeling and Verification of Systems with Self-xProperties. Lecture Notes in Computer Science, Springer.

[7] Magee, J., Maibaum, T. (2006). Towards Specification, Modelling and Analysis of Fault Tolerance in Selfmanaged Systems. *In*: Self-adaptation and Self-managing Systems, ACM.

[8] Ebnenasir, A. (2007). Designing Run-Time Fault-Tolerance Using Dynamic Updates. *In*: Software Engineering for Adaptive and Self-Managing Systems, IEEE.

[9] Becker, B., Beyer, D., Giese, H., Klein Schilling, D. (2006). Symbolic invariant verification for systems with dynamic structural adaptation. *In*: 28th International Conference on Software Engineering.

[10] Khakpour, N., Khosravi, R., Sirjani, M., Jalili, S. (2010). Formal analysis of policy-based self-adaptive systems. *In*: Symposium on Applied Computing, ACM.

[11] Heinzemann, C., Henkler, S. (2011). Reusing dynamic communication protocols in self-adaptive embedded component architectures. *In*: 14th Symposium on Component-based Software Engineering, ACM.

[12] Bartels, B., Kleine, M. (2011). A CSP-based framework for the specification, verification, and implementation of adaptive systems. *In*: Software Engineering for Adaptive and Self-Managing Systems, ACM.

[13] Georgiadis, I., Magee, J., Kramer, J. (2002). Self-organising software architectures for distributed systems. *In*: 1st Workshop on Self-healing Systems, ACM.

[14] Tan, L. (2006). Model-Based Self-Adaptive Embedded Programs with Temporal Logic Specifications. *In*: 6th International Conference on Quality Software.

[15] Dowling, J. (2004). Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems. Ph.D. thesis, University of Dublin, Trinity College.

[16] Autili, M.,Di Benedetto, P., Inverardi, P. (2009). Context-Aware Adaptive Services: The PLASTIC Approach. In: Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science 5503, Springer.