

Model Driven Mutation Applied to Adaptative Systems Testing

Alexandre Bartel¹, Benoit Baudry², Freddy Munoz², Jacques Klein¹, Tejeddine Mouelhi¹ and Yves Le Traon¹

¹ Interdisciplinary Center for Security, Reliability and Trust
University of Luxembourg

L-1359 Luxembourg-Kirchberg, Luxembourg
{alexandre.bartel, jacques.klein, tejeddine.mouelhi, yves.lettraon}@uni.lu

² INRIA Centre Rennes - Bretagne Atlantique
Campus de Beaulieu

35042 Rennes, France
{benoit.baudry, freddy.munoz}@inria.fr

Abstract—Dynamically Adaptive Systems modify their behavior and structure in response to changes in their surrounding environment and according to an adaptation logic. Critical systems increasingly incorporate dynamic adaptation capabilities; examples include disaster relief and space exploration systems. In this paper, we focus on mutation testing of the adaptation logic. We propose a fault model for adaptation logics that classifies faults into environmental completeness and adaptation correctness. Since there are several adaptation logic languages relying on the same underlying concepts, the fault model is expressed independently from specific adaptation languages. Taking benefit from model-driven engineering technology, we express these common concepts in a metamodel and define the operational semantics of mutation operators at this level. Mutation is applied on model elements and model transformations are used to propagate these changes to a given adaptation policy in the chosen formalism. Preliminary results on an adaptive web server highlight the difficulty of killing mutants for adaptive systems, and thus the difficulty of generating efficient tests.

Index Terms—model driven engineering, MDE, mutation, testing, adaptative systems

I. INTRODUCTION

Dynamically Adaptive Systems (DAS) must adapt themselves to ongoing circumstances and find the way to continue accomplishing their functionalities. DAS play increasingly important role in society's infrastructures; the demand for DAS appears in application domains ranging from crisis management applications such as disaster management [8], space exploration [6], and transportation control to entertainment and business applications. This demand is intensified by the mobile and nomadic nature of many of these domains. The IDC¹ analysts forecast a global increase in the number of mobile workers to more than 850 million by 2009 [5].

DAS respond to environmental changes by modifying their internal configuration to continue meeting their functional and non-functional requirements. Designing a DAS involves

specifying environmental fluctuations that have an impact on the system, as well as the related strategies for performing the structural changes. This is captured by an adaptation logic that expresses the actions to be adopted when the environment changes [4], [7], [9], [15]. More precisely, adaptation logics drive the adaptation process and compute the right system configuration that should be adopted given an environmental condition.

This paper focuses on the issue of testing whether an adaptation logic is correctly implemented. More specifically, we focus on mutation of adaptation logic, considering that test cases should be able to distinguish between the original adaptation logic and the mutated one. Mutation thus provides a qualification criterion for test cases.

We use a Model-driven engineering (MDE) process to model adaptation formalisms/languages as well as adaptation policies defined according to these formalisms. A metamodel captures all the necessary concepts for representing action-based adaptation policies. From the metamodel, we derive mutation operators that can apply to several action-based adaptation formalisms.

We classify adaptation logic faults into two groups:

- 1) The possible environmental conditions the system will face, and
- 2) the complexity involved in producing a response to those conditions.

The first, environmental completeness (EC) faults embody faults due to gaps in the space covered by the adaptation logic, thereby missing adaptations for environmental changes. The second, adaptation correctness (AC) faults embody faults due to incorrect adaptations to environmental changes. Our hypothesis is that managing environmental changes involving a single property variation (simple) is easy, whereas managing several properties varying at the same time (complex) is error prone. We summarize the contributions of this paper as follows:

- 1) A generic metamodel capturing the concepts inherent to

¹IDC is an analyst company and a global provider of market intelligence, advisory services, and events for the information technology, telecommunications, and consumer technology markets.

adaptation logic, completed with model transformations from two different input formalisms.

- 2) A generic set of mutation operators for adaptation logics as well as a specialization of this model to action-based adaptation logics.
- 3) A first proof of concepts through an adaptive web server case study.

It has to be noted that we do not deal with efficient test cases generation in this paper, and for the experiments we simply create test sequences randomly (sequence of events issued by the environment).

The remainder of this paper proceeds as follows. Section 2 provides a background on dynamically adaptive systems. Section 3 introduces model driven engineering techniques and explains how they can be used with testing adaptation logics. Section 4 describes the first mutation operators we used. Section 5 presents our first experiments. Section 6 presents the related work. Finally, we conclude and present our perspectives in section 7.

II. DYNAMICALLY ADAPTIVE SYSTEMS

Consider an adaptive web server, which processes file requests over the HTTP protocol. It answers these requests as fast as possible while optimizing the resources it consumes, e.g. memory, CPU time, etc. Additionally, it provides non-stop service, thus it needs to adapt its internal structure to respond to a changing working environment. This environment is characterized by the variable amount of requests over time.

A. Environment and configurations

Dynamically adaptive systems (DAS) encode the environment into an abstraction called context.

Definition 1 (context): A context consists of a n-tuple of fields $\langle p_0, p_1, \dots, p_n \rangle$, where each field p_i represents an environmental property. The type of each field is defined by the encoding chosen for the property it represents.

In our adaptive web server example, the environment is modeled as a context with two properties:

- p_1 : number of request per second (server load);
- p_2 : the percentage of request (request density).

The last one corresponds to the number of requested files. The domain or type of each property has a lower and an upper bound. For instance, we associate the type integer with request density and server load, a lower bound 0 and upper bound 100 for both. The server load domain indicates that the minimum amount of request in one second is 0 (no request) and the maximal is 100. Analogous, request density indicates the number of requested files.

Definition 2: Specific environmental conditions at an instant t are drawn by an instance I of the context representing the environment. Such an instance is an n-tuple of values corresponding to the punctual value of a particular property. The context instance $\langle 12, 3 \rangle$ designates a particular environmental condition with 12 requests per second requesting 3 different files. A sequence of context instances $I_0, I_1, I_2, \dots, I_n$ ordered by their occurrence over time is called a

context flow (CF). A context generates a space containing all the possible instances that can produce the combination of property values. The context of the adaptive web server generates a space containing all its possible context instances.

B. Adaptation logic

Adaptation in DAS is driven by an adaptation logic (adaptation model) that uses a specific strategy to describe the configuration to adopt given a context change.

Definition 3: An adaptation logic defines a relation between contexts and system configuration. It receives a context instance (current environmental condition), a context flow (history of the environment, and the history of the system configuration, and gives the next configuration the system must adopt.

There exist several strategies to describe adaptation logics, a few examples are: action-based adaptation [9], where adaptations are triggered when a condition is satisfied; goal based adaptation [7], where adaptations are performed to reach a specific goal; and utility function based adaptation [15], where adaptations are calculated according to a cost function based on environmental conditions and variation point value.

An action-based strategy describes the adaptation logic of the adaptive web server [9]. In this case the adaptation logic is a set of rules (adaptation policies) that, whenever an event occurs (environmental change) evaluate if a set of conditions are satisfied, and if it is so, they perform a series of adaptation actions.

Table I
EXCERPT OF THE ADAPTIVE WEB SERVER ADAPTATION LOGIC.

```

1: when requestdensity is 'high' or 'medium'
2: if cacheHandler.size == 0
3: then utility of addCache is 'high'

5: when requestdensity is 'low'
6: if cacheHandler.size == 0
7: then utility of addCache is 'low'

9: when LOAD is 'high'
10: if FileServers.size <= 10
11: then utility of addFileServer is 'high'

12: when LOAD is 'LOW'
13: if FileServers.size <= 10
14: then utility of addFileServer is 'low'

```

Table I presents an excerpt of the adaptive web server adaptation logic. The first two rules manage the system cache. The first rule (lines 1-3) enables the cache mechanism when the request density is high or medium (line 1) and there is no cache (line 2). The second rule (lines 5-6) is analogous to the first. It reflects the fact that when the dispersion is high, adding a cache is not very useful. The remaining rules (lines 9-14) handle the variations of the server load property.

While table 1 presents a textual action-based adaptation logic, the Diva framework² allows to express the adaption policy in the form of a set of tables which are directly

²<http://www.ict-diva.eu/DiVA/>

manipulated as models elements. Thus the connection with our MDE process is natural.

III. MDE AND ADAPTATION LOGICS

This section will introduce the MDE concepts which are required to understand how we create and use mutation operators later in the paper.

A. Metamodeling, Kermeta and Sintaks

This section summarizes the intents of metamodeling and how the *Kermeta* environment fits in this modelling activity.

1) *Metamodelling*: Metamodelling consists in building a metamodel that defines a modeling language for a particular domain. The metamodel defines the concepts and relationships that describe the domain. A metamodel is a model itself that is expressed with a modeling language called the meta-metamodel.

2) *Kermeta* : Kermeta [14] is an open source metamodeling environment developed by the Triskell team at IRISA (Research Institute in Information technology and rAndom Systems) that is fully integrated with Eclipse. This action language is imperative and object-oriented and is used to provide an implementation of operations defined in metamodels.

3) *Sintaks*: Sintaks [13] is a tool to defines bridges between plain text files and models.

B. Action-based adaptation logic metamodel

Figure 1 represents the metamodel we propose to represent the abstraction of action-based adaptation logics. An action-based adaptation logic always consists of a set of rules (Rule-Set in the figure) called Event-Condition-Action, or ECA, rules. One ECA rule (Rule) features one event (Property), one condition (Condition) and one action (Action). An event is bound to a context property. When the bounded context property changes and its new value matches the event condition (propertyCondition), then the rule is executed. When the rule is executed the condition (Condition->BoolExpression) has to be true to perform the rule's action. This condition usually refers to internal states of the adaptation system. The action consists of assigning a new value (newValue) to a property (actionProperty).

In short, a rule performs an Action if the bounded Event property in the new context matches the Event condition and if the rule Condition is true. For instance, the first rule of the adaptation logic represented table I, is bounded to the property "requestdensity". The rule will be executed only after specific context changes in which the property became "high" or "medium". The action of assigning "high" to "addCache" is only performed if the internal variable cacheHandle.size equals zero.

The metamodel ecore file was created using EMF (Eclipse Modeling Framework) and GEE (Graphical Ecore Editor).

We will describe the process of mutant generation as well as the genericity of the metamodel in the following section.

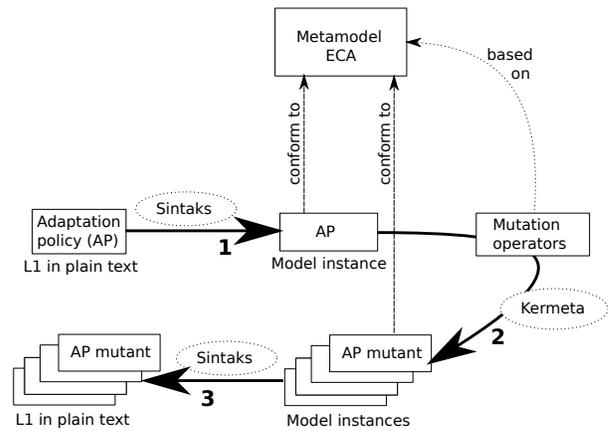


Figure 2. Mutants generation process.

C. Generic process for mutant generation

Figure 2 represents the mutants creation process. The process start by selecting an adaptation policy (AP) expressed in an action-based language (L1 on the figure). The first step (1) is to transform the adaptation policy into a model conforming to the metamodel. The second step consists in applying the mutation operators to the policy. Mutation operators are generic and work on models, not plain text files. Once the mutant models have been generated, they are transformed in plain text files. The Sintaks tool was used to do a mapping between a rule set written in plain text and its model representation.

Since mutation operators are defined from the metamodel and are working on models, they are independent of the action-based language used to write the adaptation policy: a bridge between textual files and models must be defined for each language. This is achieved by defining one bridge for each language with Sintaks.

As a result we got a set of Resulting plaintext mutants will be used to test the adaptation logic's test suites. We consider that test suites must be able to distinguish a correct adaptation logic from the incorrect ones.

In the following section we introduce the first mutation operators.

IV. MUTATION OPERATORS FOR ADAPTATION LOGICS

Definition 3 introduces the concept of adaptation logic as the driver of the adaptation. Testing the realization of such driver means verifying whether the system is capable of adapting to environmental changes, and whether such adaptations proceed as expected. This section presents the challenges associated with testing adaptation logics, as well as a fault model for adaptation logics.

A. Testing challenges

Testing adaptation logics involves generating context instances, and evaluating the results of exposing the system to such context instances.

Three steps compose the testing process:

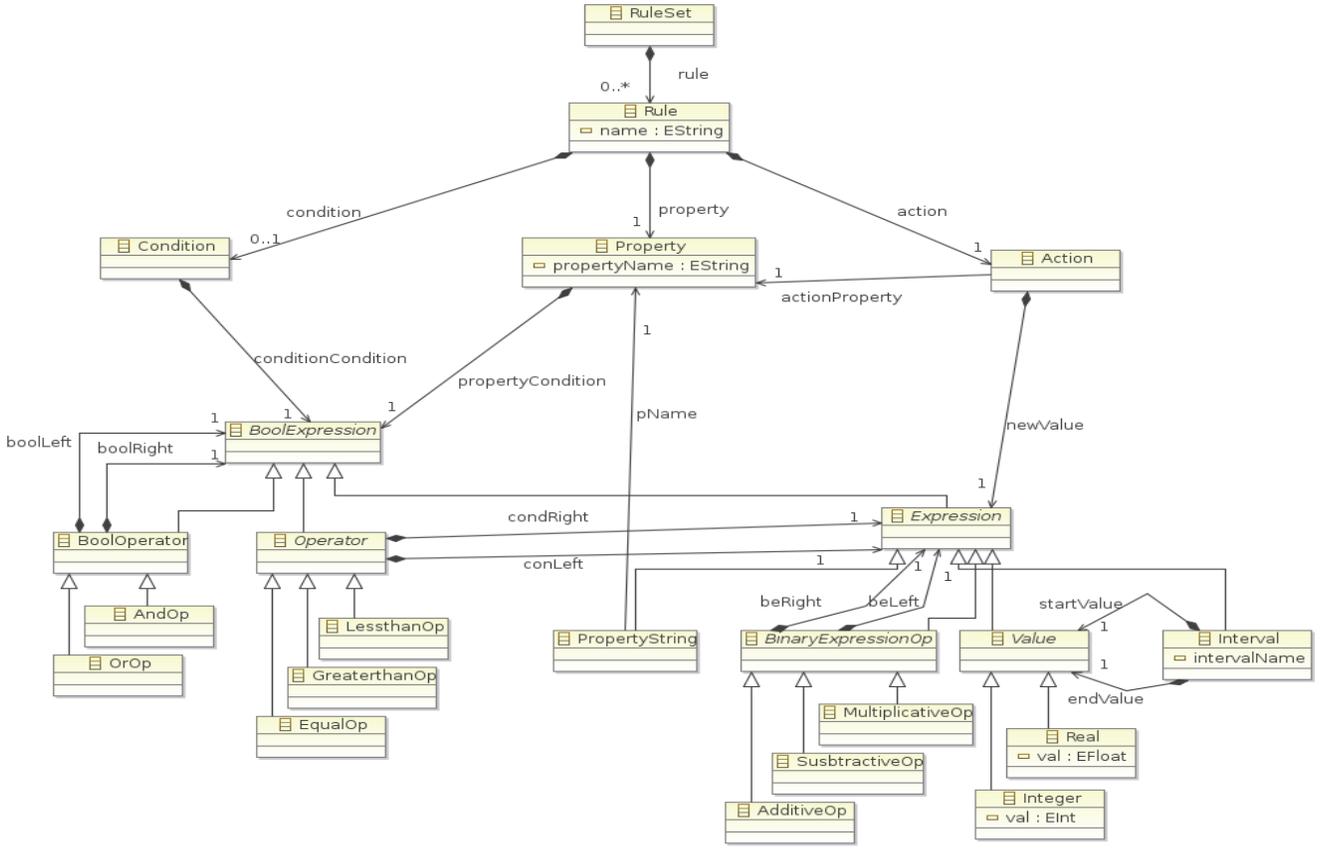


Figure 1. Metamodel of action-based adaptation logics

- 1) Initially, testers synthesize a context flow from a series of context instances.
- 2) Then, they execute and expose the system to the generated context flow. Testers evaluate whether the configurations adopted by the system (configuration flow) when exposed to environmental changes are as expected. If not, the adaptation logic contains a fault.
- 3) The process may start again until a qualification criterion is reached.

Note that (1) and (2) are not the object of the paper. Thus we generate test cases randomly. We rather focus on (3).

A *test suite* is a set of *test cases*. In this paper a test case is defined as a *context flow* of a certain length, L . L represents the number of *context instances* in the flow. Given a flow f containing L context instances I_i $i \in (1, 2, \dots, L)$, I_i and I_{i+1} differs by one or more of their properties' values. For each I_i the adaptive system will generate one or more events corresponding to the properties that have changed. Those events are then handled by the adaptation logic (rules) which generates a new configuration for the system.

A test case is said to kill a mutant if the result (new configuration) generated by the mutant adaptation logic differs from the result given by the original adaptation logic.

This process enables us to detect:

- duplicate rules or useless rules (the mutant is not killed in this case)
- errors in the adaptation logic

- either an event is not handled properly or
- an incorrect action is performed leading to an incorrect new configuration

B. Fault model for adaptation logics

Managing the scenarios to which a system adapts is complex due to their large number and the difficulty to foresee the interactions between them.

In this section we introduce generic mutation operators for the adaptation logics metamodel. Those operators will mutate adaptation logic models conforming to the metamodel and thus are independent of any adaptation logic language.

1) *Environmental completeness faults* : Definition 1 defines a context as a tuple of fields representing environmental properties. The adaptation logic interprets these fields' values, and decides the system configuration that best fits the environmental conditions. It is possible, however, that the adaptation logic neglects some property values, or a complete property. We call faults of this type environmental completeness (EC) faults.

In the following, we describe three different types of EC faults represented as mutation operators.

- 1) ICP - Ignore Context Property

For a given property p , delete each rule that can be executed on p .

For instance, when ignoring property “requestdensity” the two last rules in table I (lines 9-14) are deleted.

2) ISV - Ignore Specific Context Property Value

For a given couple (property p , value v), delete each rule that can be executed when p equals v .

When ignoring value “high” for property “LOAD” one rule (lines 9-11) is deleted.

3) IMV - Ignore Multiple Context Property Values

For a given set of couples (property p_i , value v_i), delete each rule that can be executed when any p_i ($i \in \{1,2,\dots,N\}$) equals v_i . (At least two rules with different properties are modified/deleted).

When ignoring value “high” for property “LOAD” and “low” for property “requestdensity”, two rules (lines 5-7 and lines 9-11) are deleted.

2) *Adaptation correctness faults* : The observable behavior produced by the adaptation logic is the adaptation it produces facing an environmental change. Some times such adaptation does not change the system in the expected way. We call this kind of faults adaptation corrected (AC) faults, because they lead directly to incorrect adaptations. Notice that the observable behavior of EC faults is manifested in at least one of the following AC faults.

1) SRA - Swap Rule Action

The action values from two rules modifying the same property are swapped.

For instance “high” and “low” swap lines 11 and 14 in table I for property “addFileServer”.

2) Modify Rule Condition Value

The condition value (always on the right part of the condition), for a condition which uses operator $>$ or $<$, in a rule is decreased or increased, respectively.

For instance in table I line 10, the value “10” is increased to “100”.

V. EXPERIMENTS

In this section we present a preliminary proof of concept based on the adaptive web server system.

A. Test subject

To validate our hypothesis about the ability of AST to uncover faults in adaptation logics, we use the adaptive web server presented in section 2 as a test subject.

Figure 3 illustrates the architectural realization of the adaptation logic presented in section 2. It is composed of a sensor component, which is aware of the environment and collects the data produced by environmental changes. It encodes the data into values representing the environmental properties of interest (context instance) and passes them to a reconfiguration engine. Finally, the reconfiguration engine loads the adaptation rules and matches the values against the adaptation rules. If an

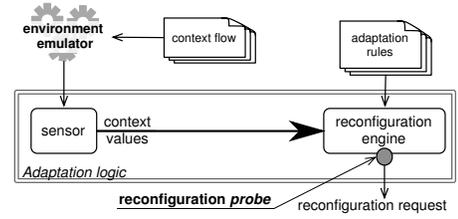


Figure 3. Instrumented architecture of the adaptive web server adaptation logic

adaptation rule matches the values, then it requests the system implementation to reconfigure as described by the rule.

To inject context instances and collect reconfiguration data we have instrumented the adaptation logic. Figure 3 presents the instrumented architecture. We have modified the source code of the sensor component and replaced the environment sensing mechanism with an environment emulator. This emulator reads context flows from a text file and injects them into the system provoking the instrumented sensor to respond identically to the non-instrumented one. We have also added a reconfiguration probe that records the reconfiguration requests produced by the reconfiguration engine.

B. Experiment set up, results and analysis

Table II
EXPERIMENT SET-UP AND EXECUTION

# of test suites	30
# of context flows per test suite	10
# of context instances per flow	20
# of mutants of the adaptation logic	130
Total number of simulations	39.000 (30 · 10 · 20 · 130)

1) *Experiment*: We prepared and executed our experiment as described in table II. We generated 30 test suites. Each of them contains 10 test cases (context flows). A flow is created by uniformly selecting a sequence of context instances among all the possible context instances.

Table III
EXPERIMENT RESULTS

Test suite	Random
minimum mutation score	91/130 \approx 70%
maximum mutation score	96/130 \approx 74%
average mutation score	93/130 \approx 71%

2) *Results and analysis*: Table III presents the global mutation score (number of unique killed mutants).

What we notice is that 30% of the mutants are not killed with random-generation. Even if we take longer test cases the results are similar. This first shows that other techniques should be studied.

C. Threats to validity

There exists no perfect data, or perfectly trustable analysis results, and this study is not an exception. For this reason we identify the construction, internal and external threats to validity for this study.

Internal threats lie on the source and nature of the empirical data. We recognize that we have only studied a small adaptive system realizing the adaptation logic through action-based reasoning. The limited number of environmental properties, and the size of the space represent a threat since it is easy to achieve a uniform coverage with few context instances.

External threats lie on the statistical significance of our study. We are aware that since the adaptive system is small and only one, it does not represent the industrial trends. To make more general statements it is necessary to try the presented technique on large system. However, DAS are an emergent technology still paving its adoption.

VI. RELATED WORK

As far as we know there is no other work that uses mutation to measure the quality of adaptation logics' tests. However, a large number of researchers have addressed the validation and testing problem of adaptive systems. Zhang et al. [17] address the verification of dynamically adaptive systems through modular model checking. They model the adaptive system as finite state machine in which states represent different system variants. Zhang and Cheng [16] introduce a model-based development process for adaptive software that uses Petri-nets. Biyani and Kulkarni [3] use predicate detection for testing adaptive systems during adaptation. They extend existing algorithms based of global predicate evaluation [2] for testing distributed systems to the system during adaptation. Kulkarni and Biyani[11] introduce an approach using proof-lattice to verify that all possible adaptation paths do not violate global constraints. Allen et al [1] used the Wright ADL to integrate the specifications of both architectural and behavioral aspects of dynamically reconfigurable systems. Kramer and Magee [10] use property automata and labeled transition systems to specify and verify adaptive program's properties. The main difference between these verification approaches and ours is the focus of attention. We are interested in verifying through testing the adaptation driver, and not the adaptation process itself. Furthermore, these approaches require computing the entire system configurations and the transitions between them, however sometimes this is not possible.

Lu et al. [12] study the testing of pervasive context-aware software. They propose a family of test adequacy criteria that measure the quality of test sets with respect to the context variability.

Since very different testing techniques exist, we hope that mutation will reveal itself as a good way to compare them.

VII. CONCLUSIONS AND PERSPECTIVES

The mutation operators presented in this paper are a first proposal to offer a qualification environment for comparing testing techniques applied to action-based adaptative systems. The use of MDE makes it possible to derive mutants for most action-based logics, thus providing a common framework for such test cases qualification. The case study shows the feasibility of the approach and confirms that, for killing mutants, other testing techniques should be considered rather than random test generation. Due to the size of the case

study and the number of environmental properties it contains, it is not possible to generalize to larger DAS. Future work will thus consist of completing the set of mutation operators, and will exhibit experimental results on other case studies comparing several test generation techniques. We plan to experiment with a much larger case study, which comprises several environmental properties and interactions. Furthermore we plan studying and specializing our fault model to other adaptation logic technologies, such as goal oriented.

REFERENCES

- [1] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. pages 21–37.
- [2] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, University of Bologna, Department of Computer Science, January 1993.
- [3] Karun N. Biyani and Sandeep S. Kulkarni. Testing dynamic adaptation in distributed systems. In Hong Zhu, W. Eric Wong, and Amit M. Paradkar, editors, *AST*, pages 51–54. IEEE, 2007.
- [4] Franck Chauvel, Olivier Barais, Isabelle Borne, and Jean-Marc Jézéquel. Composition of qualitative adaptation policies. In *Automated Software Engineering Conference (ASE 2008)*, pages 455–458, 2008. Short paper.
- [5] Drake-Giusto, Boggs, Sandler, and Burden. Worldwide mobile worken population 2005-2009 forecast and analysis, 2005.
- [6] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in JPL's mission data system. In *AIAA Space Technology Conference and Exposition, Albuquerque, NM.*, 1999.
- [7] Frank Eliassen, Eli Gjorven, Viktor S. Wold Eide, and Jørgen Andreas Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. In Nalini Venkatasubramanian Geoff Coulson, editor, *The 5th annual Workshop on Adaptive and Reflective Middleware (ARM 2006)*, pages 1–6. ACM Press, 2006.
- [8] Danny Hughes, Phil Greenwood, Gordon Blair, Geoff Coulson, Paul Smith, and Keith Beven. An intelligent and adaptable grid-based flood monitoring and warning system. In *Proceedings of the UK eScience All Hands Meeting*, pages 53–60, 2005.
- [9] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, pages 3–14. IEEE, June 2003.
- [10] Jeff Kramer and Jeff Magee. Analysing dynamic change in distributed software architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
- [11] Sandeep S. Kulkarni and Karun N. Biyani. Correctness of component-based adaptation. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 2004.
- [12] Heng Lu, W. K. Chan, and T. H. Tse. Testing pervasive software in the presence of context inconsistency resolution services. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 61–70. ACM, 2008.
- [13] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondeur, Michel Hasenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *Proceedings of the MoDELS/UML 2006*, Genova, Italy, October 2006.
- [14] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.
- [15] William E. Walsh, Gerald Tesaro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems, 2004.
- [16] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 371–380. ACM, 2006.
- [17] Ji Zhang, Heather Goldsby, and Betty H. C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, pages 161–172. ACM, 2009.