Specification of Failure-Handling Requirements as Policy Rules on Self-Adaptive Systems

João Pimentel[1,2], Jaelson Castro[1], Xavier Franch[2]

[1] Universidade Federal de Pernambuco - UFPE, Centro de Informática, Recife, Brazil
{jhcp, jbc}@cin.ufpe.br
[2] Universitat Politècnica de Catalunya, Omega-122, CP: 08034, Barcelona, Spain
franch@essi.upc.edu

Abstract. Most adaptive systems have compensation mechanisms for recovering from or preventing failures. However, sometimes a compensation is not essential. Hence, diagnosing and compensating each and every one of their failures may be ineffective. Rather than polluting a requirements specification with fine grained definition of failure-handling conditions, this work aims to increase the flexibility of failure handling in self-adaptive systems using tolerance policies. We allow the expression of conditions in which certain failures may be ignored – i.e., conditions on which a failure will not be compensated. Such policies may lead to reduced costs and performance improvement. The FAST framework consists of the definition of a tolerance policy, the mechanisms to evaluate this policy and a tool to aid the creation of policies.

Keywords: Self-adaptive systems, Autonomic systems, Failure requirements, Requirements engineering, Policy specification

1 Introduction

Adaptive systems are systems that are able to change their behavior in response to changes on the environment and on the system itself [5]. Similarly to autonomics systems [12], these systems should be able to change their behavior at runtime with minimal human intervention [14][17], even in dynamic environments. In such a system, failures are handled with compensations – or recovery activities. At design time, possible failures are identified and responses to the respective failures are defined. However, these responses may have a significant impact on non-functional requirements, such as performance and cost. For instance, the failure of a free web-service may be compensated through the usage of a similar but paid service. Therefore, it is important to allow some flexibility on the definition of which failures are to be compensated and on which scenario.

The notion that different failures have different impacts on different users and contexts is widespread on the literature [4][11][21]. So, rather than defining this as static requirements, we propose the usage of policies defined during the system deployment or at run time. The concept of policies is used in Software Engineering to

allow users or system administrators to control some characteristics of a system, without having to deal with implementation details [8]. In particular, this concept has often been used by the network community [25][24]. In this work we are defining a policy to enable the customization of the way that a system handles its failures.

The FAST Framework – Failure hAndling for Autonomic Systems – comprises the policy specification, algorithms to process the policy and a supporting tool. This framework was initially aimed to provide this flexibility for autonomic systems, more specifically with a self-configuring architecture [19]. In this paper we are going to present a generic version of this framework. Hence, a large variety of systems could borrow the concepts and mechanisms presented here to enhance its failure handling.

This paper is organized as follows. Section 2 presents our approach for expressing conditions in which a failure may be ignored – namely, the Tolerance Policy. The algorithm for processing this policy is presented in Section 3. Section 4 describes an agent that implements the policy algorithms and the tool developed to support the policy. In section 5 we compare our research with related works. Finally, Section 6 summarizes our work and points out open issues.

2 Tolerance Policy

A policy may be seen as an set of policy rules [24], which is formed by a condition and its corresponding actions [24][18]. When the conditions apply, the respective actions are performed. The tolerance policy is concerned with the definition of conditions for failures to be ignored. An excerpt of the conceptual model for this policy is presented on Fig. 1.

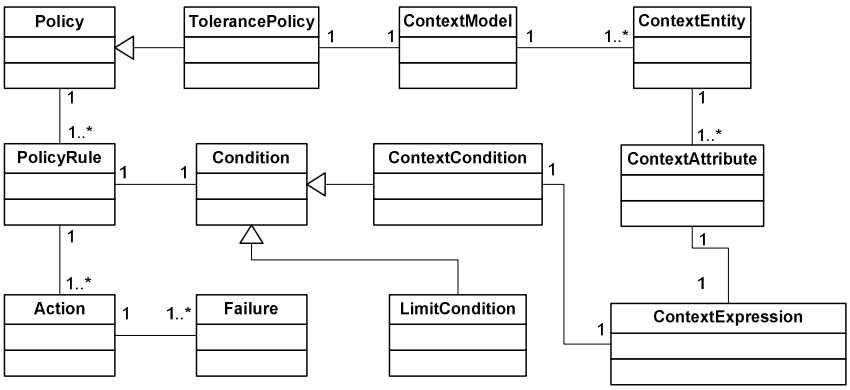

Fig. 1. Conceptual model excerpt of the Tolerance Policy

A failure may be a high-level failure – such as the non-achievement of a goal – or a low-level failure – such as an error reported by a software component. As default, for all failures that have a recovery action this action will be performed when the failure happens - only those failures explicitly mentioned in some rule of this policy will have its failures disregarded, i.e., will not trigger a compensation. Failures may be ignored depending on conditions that may be related to the system's context or to the

amount of occurrences of a failure. For each of these types of conditions, there is a specific rule type: t.context (ContextCondition) and t.limit (LimitCondition). The 't' in these type names stands for 'tolerance'.

Besides the list of failures that have recovery actions, another input for this policy is the context model, or environmental model. The context model specifies the data that will be monitored by the system. In self-adaptive systems this data is used to identify when an adaptation should be performed and to identify the occurrence of failures themselves [7]. In the FAST framework we are considering a context model in the form of entities and their attributes [3], expressed in XML. When an attribute is an enumeration, this XML also define its possible values. In Fig. 2 we show an example of a context model. In this case, there are two context entities: Internet and Calendar. The Internet entity has the attribute Speed, which possible values are zero, low, average and high. The Weekday of the Calendar may be Sunday, Monday, and so on, while the Hour is a number.

In the following sub-sections it will be described the two tolerance rule types – t.context and t.limit. The regular expressions that precisely define the rules syntax are presented in Appendix A.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <root>
3      <ContextEntity name="internet">
4          <Attribute name="speed" type="enum">
5              <Value>zero</Value>
6              <Value>low</Value>
7              <Value>average</Value>
8              <Value>high</Value>
9          </Attribute>
10     </ContextEntity>
11     <ContextEntity name="calendar">
12         <Attribute name="weekday" type="enum">
13                 <Value>sunday</Value>
14                 <Value>monday</Value>
15                 <Value>tuesday</Value>
16                 <Value>wednesday</Value>
17                 <Value>thursday</Value>
18                 <Value>friday</Value>
19                 <Value>saturday</Value>
20         </Attribute>
21         <Attribute name="hour" type="number">
22         </Attribute>
23     </ContextEntity>
24 </root>
```

Fig. 2. A context model example. This XML excerpt shows two context entities – internet and calendar – and their attributes.

2.1 Tolerance Rule Type t.context

In order to express in which contexts certain failures may be ignored we use t.context rules. This rule type has the following structure:

failuresSet isAllowedToFailIf *contextExpression*

failuresSet is a set of failure identifiers divided by a colon (:), and that has at least one failure - i.e., it cannot be an empty set. The *allFailures* reserved word may be used to refer to all the recoverable failures of a system, without the need to name them one by one.

isAllowedToFailIf is a fixed string to identify the rule type. *contextExpression* is a logic expression, with the following structure:

contextEntity.attributeName operator attributeValue

contextEntity is any entity of the system's context model, and *attributeName* is the name of an attribute of that entity. *operator* is a logic comparator, among the following: equals (=), greater than (>), greater equals than (>=), lower than (<), lower equals than (<=) and different (<>). *AttributeValue* is any possible value that entity's attribute may have. During the system execution, this value will be compared with the actual value of that attribute, in order to evaluate if this context applies or not.

A rule of the t.context type has the following meaning: if a failure that is an element of the *failuresSet* occur and the *contextExpression* currently applies, then that failure will be ignored. In other words, no compensation will be performed for that failure.

Usual situations in which a failure can be ignored are those related to date and time, as in examples 1 and 2. Example 1 states that if a certain *failureX* occurs but it is before 8 am, this failure will be ignored. The same applies for *failureY*. In Example 2, we express that the occurrence of any failure of that system will be ignored on Sundays.

Ex.1: *failureX: failureY isAllowedToFailIf calendar.hour<=8*
Ex.2: *allFailures isAllowedToFailIf calendar.day=Sunday*

2.2 Tolerance Rule Type t.limit

In this rule type we are not concerned in defining specific conditions in which a failure will be ignored. Instead, the concern is to define a maximum number of times that some failure will occur without being compensated. This type has the following structure:

failuresSet isAllowedToFailAtMost *limit*

failuresSet is defined as in t.context. The isAllowedToFailAtMost uniquely identifies this rule type. *limit* is a positive integer number that indicates how many consecutive occurrences of each failure of the *failuresSet* will be ignored, before a compensation is triggered.

A rule of this type means that each failure of the *failuresSet* will have a limit number of occurrences ignored. The failure number *limit + 1* will be compensated,

and the occurrence counting of that failure will be reset. However, the failures that are ignored due to a t.context rule are not included on this counting, as it will be explaind in Section 3.

Note that, when using more than one failure in the *failuresSet*, we do not define a limit of occurrences for a set of failures, but the limit for each failure of the *failuresSet*. For instance, in Example 3 the limit of 4 occurrences is not for the two failures altogether, it is for each failure separately (*failureX* and *failureY*). The rule in Example 3 can be split in other two rules (examples 4 and 5), keeping the same meaning.

Ex.3: *failureX: failureY isAllowedToFailAtMost 4*
Ex.4: *failureX isAllowedToFailAtMost 4*
Ex.5: *failureY isAllowedToFailAtMost 4*

3 Policy Processing Algorithm

The goal of the Tolerance Policy processing is to define all failures that will be ignored. For that, the procedure described in Fig. 3 is used. The parameters are the failure itself - i.e., a failed that actually occurred -, a list of tolerance rules, from the policy, and a list of context entities, from which we can get the current attribute values of that entities. The result of this procedure is the status of the current failure occurrence: ignored or not ignored.

The first step is to check if there is a rule of the type t.context which *failuresSet* contains that failure (line 1). If there is such a rule, we need to analyze each one of these rules (line 2). If the rule is of the type t.context and its context expression applies, we will label that element as *ignored* (lines 3 to 9). The analysis of the context expressions is performed by the procedure EvaluateContext. The EvaluateContext procedure trivially checks if the rules conditions apply [20]. After analyzing all t.context rules for the failure occurrence, if it is not yet marked as *ignored* (line 12), we will check if there is a rule of the type t.limit which *failuresSet* contains that element (line 13). If there is such a rule, we will check if the limit for that failure has already been reached (line 14). If the limit has not been reached yet, we will increase the occurrence counter of that failure and mark it as *ignored* (lines 15 and 16). If the limit has been reached, we cannot ignore that occurrence - i.e, the compensation will be required - and we reset the failure counter (line 18) for that failure. As a result we return the status of the failure occurrence, indicating if it should be ignored or not (line 22).

In summary, the t.context rules define conditions when the occurrence of a given failure may be ignored, and t.limit rules define the maximum number of consecutive occurrences of a given failure that can be ignored. However, the amount of occurrences defined with a t.limit rule does not take into account the occurrences already ignored by the t.context rules. In this sense, we can state that the rule type t.context prevails upon the type t.limit. Given a t.context rule, the occurrence of a failure in its failuresSet will always be ignored if its context expression is satisfied, in despite of how many times this failure had been ignored before.

```
Data: f : Failure, TR : ToleranceRule[], CE : ContextEntity[]
1  if ∃ tr1 ∈ TR tr1.type = tcontext and tr1.failuresSet.contains(f) then
2      foreach trj in TR do
3          if trj.failuresSet.contains(f) then
4              if trj.type = tcontext then
5                  if EvaluateContext(trj.expression, CE) then
6                      f.status ← ignored
7                  end
8              end
9          end
10     end
11 end
12 if f.status ≠ ignored then
13     if ∃ tr2 ∈ TR tr2.type = tlimit and tr2.failuresSet.contains(f) then
14         if f.failureCounter < tr2.limit then
15             f.status ← ignored
16             f.failureCounter ← f.failureCounter + 1
17         else
18             f.failureCounter ← 0
19         end
20     end
21 end
22 return f.status
```

Fig. 3. Algorithm for processing the Tolerance Policy at runtime

The t.limit rules are concerned only with the failures that were not ignored during the evaluation of the t.context rules. Note that the failures ignored due to a t.context rule will not change the occurrence counting of a failure.

Rules can interact. Table 1 for example shows a log of occurrences for the failure *failureX*, considering the two rule types expressed in examples 6 (a t.context rule) and 7 (a t.limit rule). That table shows the number of each failure occurrence and the value of the *calendar.day* attribute, which is required to assess if any of these rules apply. It also indicates if the failure occurrence was ignored as well as the rationale for ignoring it – i.e., the rule that made the failure be ignored.

Ex.6: *failureX isAllowedToFailIf calendar.day=sunday*
Ex.7: *failureX isAllowedToFailAtMost 3*

In this example, the failure occurrences for which the rule of the example 6 applies are failures number 2 and 3. However it is not applicable for occurrences number 1, 4, 5, 6 and 7, hence we have to evaluate the rule of the example 7. The occurrences 1, 4 and 5 were ignored, since they were below the limit of 3 failure occurrences expressed in the rule. The occurrence number 6, being the fourth occurrence of that failure that were not ignored by a t.context, shall be compensated, and the occurrence counter for that failure shall be reset. Since the occurrences counter was reset, the occurrence 7 was also ignored for being below the limit of three occurrences.

Table 1. Occurrence log of the failure *failureX*

Occurence number	Calendar.day	Ignore failure?	Rationale
1	Saturday	Yes	Ex. 7 (1st occurrence)
2	Sunday	Yes	Ex. 6
3	Sunday	Yes	Ex. 6
4	Monday	Yes	Ex. 7 (2nd occurrence)
5	Monday	Yes	Ex. 7 (3rd occurrence)
6	Monday	No	
7	Tuesday	Yes	Ex. 7 (1st occurrence)

4 Application

In order to use our approach we developed a policy manager component that implements the algorithm presented in Section 3. This component is responsible for loading the policy rules, presented in Section 2, and the context model. Besides, it receives updates on the context and assess if a given failure should be ignored or not, upon requests of other components.

For illustration purpose, on this paper we are presenting the Policy Manager component encapsulated as an agent - the FAST Agent. This is a way of showing the generic characteristic of this framework. We also envision the usage of the FAST implementation as a crosscutting aspect [10], in synergy with works about aspectual modeling on multi-agent systems [2][23].

The exchange of messages between a system using the FAST framework and the FAST Agent itself is depicted in Fig. 4. The first two messages are related to the initialization of the FAST Agent, by providing the Uniform Resource Locator (URL) of the files that contain the policy and the context model for that system. Then it is expected to occur some messages of the third kind, in order to inform an initial state of the context. During the rest of the execution of the system, there will be an exchange of messages to update the context (message 3) and to check if a failure shall be ignored (message 4). Therefore, the agent is not responsible for identifying context changes or the occurrence of failures - it receives this information from the system itself, or from a monitor system.

The policy file is a text file in which each line contains a policy rule. The syntax of the rules is described in Appendix A as Java regular expressions. The context model is a XML file containing the context entities and their attributes. The context model defines the data that will be monitored by the system, and that will be informed to the FAST agent. This way it will be possible to assess if a given t.context rule applies on a specific moment during the execution.

Besides the FAST Agent, we developed a tool for making it simpler to create the policy rules. With this tool we are able to prevent syntax errors that could otherwise occur. Fig. 5 shows an example of the creation of a t.context rule. The user selects failures, from the list of failures that have recovery actions, and then defines in which context that failure can occur without compensation. In this example, the failures are regarding the updating of data on a movie system.

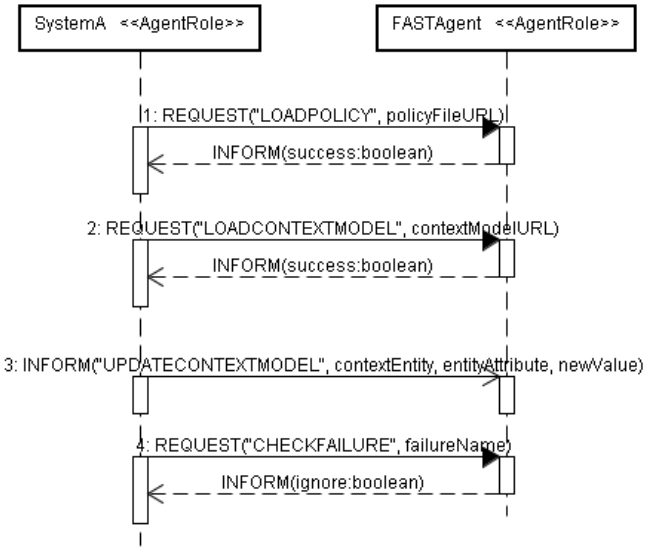

Fig. 4. Communication between a system and the FAST Agent

In order to prevent the user from deciding to ignore a critical failure, the list of possible failures informed for this tool may be a partial model. Therefore, the reserved word *allFailures* will not include the omitted failures.

The rule defined in the example of Fig. 5 is *downloadPictures* isAllowedToFailIf *calendar.weekday=saturday*. This policy editor tool makes it easier for the user to create and maintain the rules of a policy.

We performed a simulation of the execution of this system, considering two variables: the amount of failures occurrence (low, medium and high occurrence) and the context on which the failures occur. All simulations were performed considering one t.context rule and one t.limit rule. The average result was a decrease of approximately 41% on the number of required compensations, preventing the computational resources waste of performing these unnecessary compensations. This gives a general idea of the suitability of this approach. However, these results cannot be generalized to every system. Hence, an analysis of the adoption of this framework needs to be performed system-by-system.

5 Related Work

There is a series of languages for policy definition in the communication networks domain. The CIM-SPL language [1] is a standard proposed by the Distributed Management Task Force to specify network policies. Rei [13] is a policy definition language based on deontic logic, on the same domain. Other languages include Ponder [9], ACPL [22] and PDL [16]. These policies, besides targeting specific domains, are far more computationally costly and complex than it was required for the framework, motivating the creation of a language of our own.

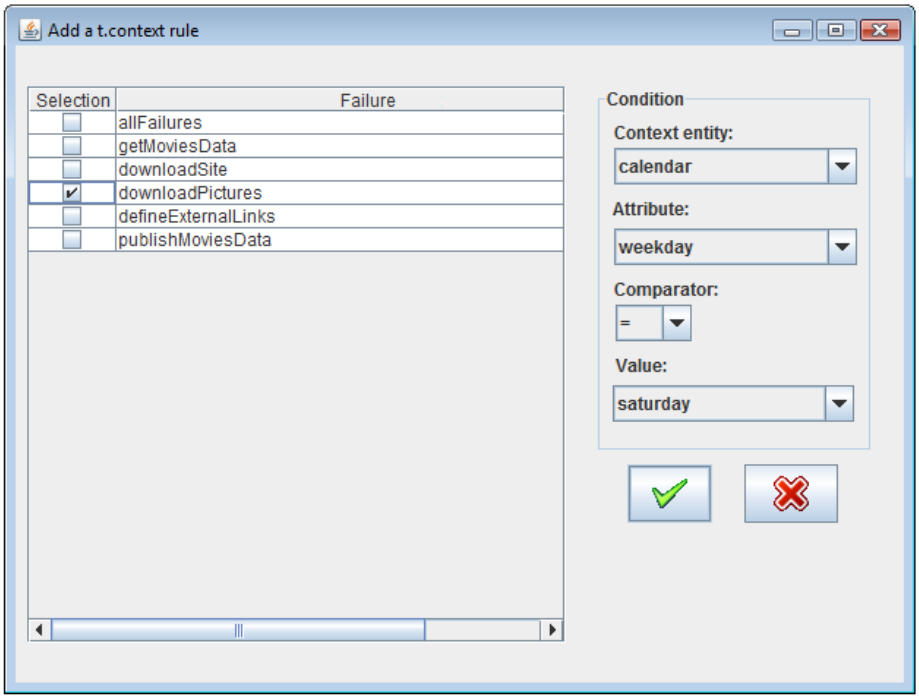

Fig. 5. Wizard for creating a t.context rule

A more strongly related kind of policies are the policies for deteriorating systems. They basically define conditions on which a software component should be repaired, based on their age, failure rate [16] and their technological obsolescence [17]. Our tolerance policy complements these policies, in the sense that we deal with another aspect of failures.

Another way of providing the flexibility to the user would be by including the failure handling in an options or a settings menu. This approach is potentially more user-friendly, however it lacks in generalization, since not every system has a graphical interface and, in those that have one, the user interfaces are usually specifically designed for each system. Moreover, the inclusion of a new category of options in the already overloaded options menu [15] could harm the usability of the software as a whole.

6 Conclusion and Future Work

In this paper we present a generic version of the FAST framework, which provides system users and administrators with the capability of defining conditions on which a failure may be ignored. The contribution of this framework is twofold:

a) It enhances the failure handling on software systems by including a degree of flexibility. This way the impact of a failure is not defined only by software engineers, but also by users or system administrators;

b) It reduces the resources wasted when compensating failures, by reducing the amount of failures that require compensation.

In this paper the policy itself was described, with its two rule types, as well as the algorithm used to assess the rules at runtime. Using a policy is far different from expressing these conditions directly on the requirements model, in the sense that the model is designed by software engineers, whereas the policy is possibly designed by the user.

The feasibility of the algorithm was shown by coding it as the behavior of a software agent. A Policy Editor tool was also developed, making it easier for the user to create and maintain the rules of a policy. Despite initial experiments showing an overall suitability, further experiments on real-world software system need to be performed, to analyze the usefulness and the effectiveness of our approach.

We also plan to increase the expressiveness of the policy rules, allowing the usage of logic operators like AND, OR and XOR to create more complex conditions. Furthermore, we want to be able to handle more complex rules, which can mix different types of a rule. Lastly, we intend to incorporate a priority policy, to define the priorities for the compensation of each failure.

Acknowledgements

We are thankful to Fabiano Dalpiaz, Paolo Giorgini and John Mylopoulos, for inspiring this work and for their continuous feedback. This work was partially supported by FACEPE, CNPQ, CAPES, UPV PAID-02-10, Erasmus Mundus External Cooperation Window - Lot 15 Brasil and the Spanish research project TIN2007-64753.

References

1. Agrawal, D.; Calo, S.; Lee, K.; Lobo, J. Issues in Designing a Policy Language for Distributed Management of IT Infrastructures. In: Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on. [S.l.: s.n.], 2007. p. 30-39.
2. Alencar, F.; Castro, J.; Moreira, A.; Araujo, J.; Silva, C.; Ramos, R.; Mylopoulos, J. Integration of Aspects with i* Models. In: Agent-Oriented Information Systems IV, LNCS 4898, Springer-Verlag, 2008, pp. 183-201.
3. Ali, R.; Dalpiaz, F.; Giorgini, P. A goal modeling framework for self-contextualizable software. In: Lecture Notes in Business Information Processing, v. 29. p. 326- 338, . Berlin Heidelberg: Springer, 2009.
4. Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, IEEE Computer Society, Los Alamitos, CA, USA, v. 1, p. 11-33, 2004. ISSN 1545-5971.

5. Cheng, Betty H.C., de Lemos, Rogério, Giese, Holger, Inverardi, Paola, and Magee, Jeff. Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, v. 5525. Berlin Heidelberg: Springer, 2009.
6. Clavareau, J.; Labeau, P. Maintenance and replacement policies under technological obsolescence. Reliability Engineering and System Safety, Elsevier, v. 94, n. 2, p. 370-381, 2009.
7. Dalpiaz, F.; Giorgini, P.; Mylopoulos, J. An architecture for requirements-driven self-reconfiguration. In: ECK, P. van; GORDIJN, J.; WIERINGA, R. (Ed.). CAiSE 2009. Lecture Notes in Computer Science, v. 5565, p. 246-260. ISBN 978-3-642-02143-5.
8. Damianou, N. Dulay, N.; Lupu, E.; Sloman, M.; Tonouchi, T. Tools for domain-based policy management of distributed systems. In: Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2002), pages 203–217, 2002.
9. Damianou, N.; Dulay, N.; Lupu, E.; Sloman, M. The ponder policy specification language. In: LECTURE NOTES IN COMPUTER SCIENCE. [S.l.]: Springer-Verlag, 2001. p. 18-38.
10. E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena. Applying and Evaluating Concern-Sensitive Design Heuristics. XXIII Brazilian Symposium on Software Engineering, 2009, pp. 83-93.
11. Hiltunen, M. A.; Immanuel, V.; Schlichting, R. D. Supporting customized failure models for distributed software. Distributed Systems Engineering, v. 6, 1999.
12. Horn, P. Autonomic computing: IBM's Perspective on the State of Information Technology. IBM, 2001.
13. Kagal, L.; Finin, T.; Joshi, A. A policy language for a pervasive computing environment. In: POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks. Washington, DC, USA: IEEE Computer Society, 2003. p. 63. ISBN 0-7695-1933-4.
14. Kephart, J. O.; Chess, D. M. The vision of autonomic computing. Computer, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 1, p. 41-50, 2003. ISSN 0018-9162.
15. Liaskos, S.; Lapouchnian, A.; Wang, Y.; Yu, Y.; Easterbrook, S. M. Configuring Common Personal Software: a Requirements-Driven Approach. In Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE), 2005.
16. Lobo, J.; Bhatia, R.; Naqvi, S. A policy description language. In: AAAI '99/IAAI'99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999. p. 291-298. ISBN 0-262-51106-1.
17. Müller, H. A.; O'Brien, L.; Klein, M.; Wood, B. Autonomic Computing. CMU/SEI-2006-TN-006. [S.l.], 2006.
18. Ouda, A.; Lutfiyya, H.; Bauer, M. Towards self-configuring policy-based management systems. In: POLICY '08: Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks. Washington, DC, USA: IEEE Computer Society, 2008. p. 215-218. ISBN 978-0-7695-3133-5.
19. Pimentel, J.; Santos, E.; Castro, J. Conditions for ignoring failures based on a requirements model. In: Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE). p. 48-53, 2010.
20. Pimentel, J. High Level Failure Treatment for Self-Configuring Systems: The FAST Approach (In Portuguese: Tratamento de Falhas de Alto-Nível para Sistemas Auto-Configuráveis: A abordagem FAST). MSc Dissertation. Federal University of Pernambuco, 2010.
21. Rinner, B. Detecting and Diagnosing Faults. In Telematik, 8(2), p. 6-8, 2002.

22. Scarlett, K. Solutions in action: Creating policy for action-based decisions in PMAC. Agosto 2006. Web-site. Last Access: 28th of November, 2010. Available on: http://www.ibm.com/developerworks/library/ac-naction1.html
23. Silva, C.; Lucena, M.; Castro, J.; Araujo, J.; Moreira, A.; Alencar, F. Support for aspectual modeling to Multiagent system architecture. In: ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, 2009 (EA´09), 2009, Vancouver. Procedings of ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, 2009 (EA´09), 2009. p. 38-43.
24. Stone, G.; Lundy, B.; Xie, G. Network policy languages: A survey and a new approach. Technical report, Defense Technical Information Center OAI-PMH Repository, 2003.
25. Strassner, J.; Samudrala, S.; Cox, G.; Liu, Y.; Jiang, M.; Zhang, J.; Meer, S.; Foghl´u, M.; Donnelly, W. The design of a new context-aware policy model for autonomic networking. In ICAC '08: Proceedings of the 2008 International Conference on Autonomic Computing, pages 119–128, Washington, DC, USA, 2008. IEEE Computer Society.
26. Wang, H. A survey of maintenance policies of deteriorating systems. European Journal of Operational Research, v. 139, p. 469-489(21), 2002.

Appendix A: Regular expressions

This Appendix presents the Java regular expressions used to define the syntax of the policy rules.

BASIC EXPRESSIONS:

Failure identifier: `[a-z][a-zA-Z]*`

Failures set: `[a-z][a-zA-Z]*(:[a-z][a-zA-Z]*)*`

Positive integer: `[0-9]*[1-9][0-9]*`

Undefined amount of whitespaces: `\\s+`

Context expression (structure): *ContextEntity.AttributeName operator AttributeValue*

Context expression (regular expression):
```
[a-zA-Z]+\\.[a-zA-Z]+
(=|>|>=|<|<=|<>)[a-zA-Z0-9]+
```

RULE TYPES:

T.context (structure): *failuresSet isAllowedToFailIf contextExpression*

T.context (regular expression):
```
^[a-z][a-zA-Z]*(:[a-z][a-zA-Z]*)*
\\s+isAllowedToFailIf\\s+[a-zA-Z]+
\\.[a-zA-Z]+(=|>|>=|<|<=|<>)[a-zA-Z0-9]+$
```

T.limit (structure): *failuresSet isAllowedToFailAtMost limit*

T.limit (regular expression):
```
^[a-z][a-zA-Z]*(:[a-z][a-zA-Z]*)*
\\s+isAllowedToFailAtMost\\s+[0-9]*[1-9][0-9]*$
```