Conditions for ignoring failures based on a requirements model

João Pimentel[1], Emanuel Santos[1], Jaelson Castro[1]
[1]*Universidade Federal de Pernambuco, Centro de Informática, Recife, Brazil*
{jhcp, ebs, jbc}@cin.ufpe.br

Abstract

Ideally, all system failures should be compensated. In fact, most failure-prone systems try to compensate all their failures. However, sometimes a compensation is not essential. Hence, diagnosing and compensating each and every one of their failures may be ineffective.

Thus, this work aims to increase the flexibility of failure handling in self-configuring systems, using tolerance policies based on requirements models. We allow the expression of conditions in which certain failures may be ignored – i.e., conditions in which a failure will not be compensated. Such policies may lead to reduced costs and performance improvement.

The FAST[1] framework consists of the definition of a tolerance policy, the mechanisms to evaluate this policy and a tool to aid the creation and maintenance of policies. We use a Smart Office system to show the several types of policy rules in action.

1. Introduction

The increasingly complexity of software systems led to the proposal of Autonomic Computing, in which the systems are capable of maintaining its ideal behavior requiring only a minimal amount of human intervention, even in dynamic environments [1]. Autonomic systems present four basic characteristics: self-configuration, self-healing, self-protection and self-optimization [2][3]. In particular, self-configuration is seen as the main characteristic [4][5], partially because of the support it provides for implementing the other characteristics.

In this work we are considering a specific architecture for self-configurable systems [6], in which the system execution is monitored at the requirements level. This architecture performs a Monitoring – Diagnosing – Compensating (MDC) cycle. It monitors the execution of a system, diagnoses the failures that may happen and proposes reconfigurations in order to avoid these failures. This monitoring and diagnosing is performed regarding the system requirements, expressed with goal models. The goals are modeled using the Tropos notation [7], which captures the social and intentional relationships in the system organizational environment, as well as quality attributes and functionalities of the system.

Ideally, all system failures should be compensated. However, sometimes compensation is not essential - it depends on the failure's criticality. For instance, let us consider that a research group has weekly meetings every Wednesday. If one of the meetings happens to coincide with a holiday, the group may just cancel that meeting and gather together in the following Wednesday. On the other hand, if there are three consecutive cancelations of the meeting (due to holidays or other motives), the time gap between one meeting and the next one would be too large. Hence, it would probably be better to reschedule some meetings to an alternative day of the week (eg. Thursdays). In this scenario, the failure – cancellation of a meeting - does not always need to be compensated. For example it is allowed to happen two times in sequence, but no more than that.

In contrast, the MDC cycle expects that each system failure will lead to compensation. Thus, this work aims to increase the flexibility of failure handling in self-configuring systems, allowing the expression of conditions in which certain failures may be ignored – i.e., conditions in which a failure will not be compensated. To discover the types of conditions, we performed an extensive analysis of goal models presented in the academic literature.

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 [9][10]. In this work we are defining a policy to enable the customization of the way that a system handles its failures.

This paper is organized as follows. Section 2 presents the background of this research, which is a

[1] Failure handling for Autonomic Systems

context-enriched Tropos notation. Section 3 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 4. Section 5 illustrates the use of our approach and the tools developed to support the policy. In section 6 we compare our research with related works. Finally, Section 7 summarizes our work and points out open issues.

2. Background

Our architecture is based on some previous work [6] which considers the requirements model as a goal model and a context model. With this information and the data provided during a system execution a self-configuration component is able to monitor and diagnose failures at runtime.

A goal model is a model that depicts the intentions of actors in a system, along with the means - tasks - to achieve these goals and the interdependencies among the actors. In particular, the self-reconfiguration architecture [6] adopted uses a Tropos [7] goal model consisting of actors, their goals, goals and/or-decompositions, tasks, means-end links (from a task to a goal) and dependency links (from an actor to another actor). Below (Figure 1) we describe the example to be used throughout the paper.

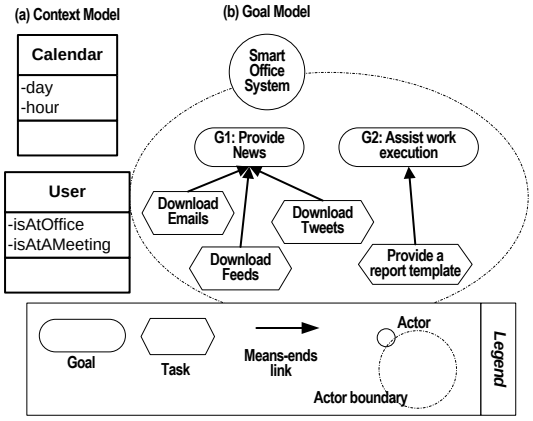

Figure 1 - Goal and context model for a Smart Office System

The example in Figure 1 (b), of a Smart Office system, shows some of these concepts. This system has two goals: provide news (to its user) and assist (a user) when performing his work. The *Download Emails*, *Download Feeds* and *Download Tweets* tasks are means to the *Provide News* end. Similarly, the *Provide a report template* task is a means to the *Assist work execution* end. We are considering that when downloading e-mail an Ethernet connection is used. When this connection is down, the reconfiguration strategy is to connect to the Internet through a mobile phone - which is more expensive than using the Ethernet connection.

The context model defines the data that context sensors will have to monitor, in order to assess the tasks execution. In Figure 1 (a) we show the context model for the Smart Office system. In this example, the context sensors would need to know the day and current hour of the calendar and if the user is at his/her office or at a meeting.

3. Tolerance Policy

Given the overview of the models we are taking into consideration, in this section we are going to describe our tolerance policy. It is concerned with the definition of conditions for task failures to be ignored. A failure, in this case, is the unsuccessful execution of a task. By default, on the MDC cycle, all failures must be compensated through some reconfiguration - only those tasks explicitly mentioned in some rule of this policy will have its failures disregarded. Failures will be ignored depending on conditions that may be related to the system's context, to the system goals or to the amount of time elapsed since the occurrence of a failure. For each of these types of conditions, there is a specific rule type: t.context, t.goal and t.limit. The 't' in these type names stands for 'tolerance'. In the following sub-sections we describe each one of all types in detail.

3.1 Tolerance Rule Type 1 (t.context)

In order to express in which contexts the failure of certain tasks may be ignored we use *t.context* rules. It has the following structure:

`tasksSet` isAllowedToFailIf `contextExpression`

`tasksSet` is a set of tasks divided by a colon (:), and that has at least one task - i.e., it can not be an empty set. The `allTasks` reserved word may be used to refer to all the tasks of the goal model, without needing to name them one by one. This definition of `tasksSet` and of the `allTasks` reserved word is shared with all the remaining types.

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 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 task that is an element of the `taksSet` fails 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. Considering the Smart Office System, that periodically download the e-mails and feeds for its user, Example 1 states that the failure of the *downloadEmail* task will be ignored if it happens before 8a.m., as well as the failure of the *downloadFeeds* task. Example 2 defines that the failure of any task of the system will be ignored if it occurs on a Sunday. Example 3 illustrates that any context entity of the context model can be part of the context expression - the failure of the *downloadEmail* task will be ignored whenever the user is away from his office.

Ex.1: *downloadEmail:downloadFeeds isAllowedToFailIf calendar.hour<=8*
Ex.2: *allTasks isAllowedToFailIf calendar.day=Sunday*
Ex.3: *downloadEmail isAllowedToFailIf user.isAtOffice=false*

3.2 Tolerance Rule Type 2 (t.goal)

This type of rule uses the goal model to define when a task failure will be ignored. The status of a goal, or a set of goals will be examined. Its structure is similar to the structure of t.goal:

`tasksSet` isAllowedToFailWhen `goalExpressions`

`tasksSet` is defined similarly to the t.context. `goalExpressions` is a non-empty set of logic expressions, separated by a colon (:). Each expression is an equality comparison: `goalName=satisfied` or `goalName=unsatisfied`. In the first case, the expression will apply if the given goal is currently satisfied, and in the second case if the given goal is currently unsatisfied. The rule will apply if and only if all its `goalExpressions` apply.

The string isAllowedToFailWhen is the identifier for this rule type. The t.goal rule states that whenever a task in the `tasksSet` fails, if all expressions of `goalExpressions` apply then this failure will be ignored.

In Example 4 we have that the failure of the *downloadFeeds* task will not be compensated if the system did not help the user in executing his/her work. In Example 5 the failure of the *downloadEmail* task will be ignored if both the *assistWorkExecution* goal is satisfied and the goal *provideNews* is satisfied.

Ex.4: *downloadFeeds isAllowedToFailWhen assistWorkExecution=unsatisfied*
Ex.5: *downloadEmail isAllowedToFailWhen assistWorkExecution=satisfied:provideNews=satisfied*

3.3 Tolerance Rule Type 3 (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 task will fail without being compensated. This type has the following structure:

`tasksSet` isAllowedToFailAtMost `limit`

`tasksSet` is defined similarly to the t.context and t.goal cases. The `isAllowedToFailAtMost` name uniquely identifies this rule type. `limit` is a positive integer number that indicates how many times the failures of each task of the `tasksSet` will be ignored, before a compensation is required.

A rule of this type means that each task of the `tasksSet` will have a `limit` number of failures ignored. The failure number `limit + 1` will be compensated, and the failure counting of that task will be reset.

Note that we do not define a limit of failures for a set of tasks, but the limit for each task of the `tasksSet`. For instance, in Example 6 the limit of 5 failures is not for the two tasks altogether, it is for each task separately (*downloadEmail* and *downloadFeeds*). The rule of the Example 6 can be split in other two rules (examples 7 and 8), keeping the same meaning.

Ex.6: *downloadEmail:downloadFeeds isAllowedToFailAtMost 5*
Ex.7: *downloadEmail isAllowedToFailAtMost 5*
Ex.8: *downloadFeeds isAllowedToFailAtMost 5*

4. Policy Processing

The goal of the Tolerance Policy processing is to define all failures that will be ignored. For that, the

procedure described in Figure 2 is used. Initially, there is a list of failed elements - i.e., the tasks that were not successfully completed. There is also a list of tolerance rules, extracted from the policy file, and a list of context entities, from which we can get the current attribute values of that entities. The result of this procedure is a list of failed elements without those which failure will be ignored.

```
   Data: FE : FailedElement [ ], TR : ToleranceRule [ ], CE : ContextEntity [ ]
 1 foreach fe_i in FE do
 2     if ∃tr_1 ∈ TR (tr_1.type = t1 or tr_1.type = t2) and tr_1.elementsSet.contains(fe_i)
       then
 3         foreach tr_j in TR do
 4             if tr_j.elementsSet.contains(fe_i) then
 5                 if tr_j.type = t1 then
 6                     if EvaluateContext(tr_j.expression, CE) then
 7                         FE.removeFailedElement(fe_i)
 8                         fe_i.status ← ignored
 9                     end
10                 else
11                     if tr_j.type = t2 then
12                         if EvaluateGoals(tr_j.expression, FE) then
13                             FE.removeFailedElement(fe_i)
14                             fe_i.status ← ignored
15                         end
16                     end
17                 end
18             end
19         end
20     end
21     if (fe_i.status ≠ ignored) then
22         if ∃tr_2 ∈ TR tr_2.type = t3 and tr_2.elementsSet.contains(fe_i) then
23             if fe_i.failureCounter < tr_2.limit then
24                 FE.removeFailedElement(fe_i)
25                 fe_i.status ← ignored
26                 fe_i.failureCounter ← fe_i.failureCounter + 1
27             else
28                 fe_i.failureCounter ← 0
29             end
30         end
31     end
32 end
33 return FE
```

Figure 2 - Algorithm for failure ignoring evaluation

For each failed element (line 1), we check if there is a rule of the type t.context (t1) or t.goal (t2) which `tasksSet` contains that element (line 2). If there is such a rule, we are going to analyze each one of these rules (lines 3 and 4). If the rule is of the type t.context (t1) and its context expression applies, we will remove this element from the list of failed elements and mark that element as *ignored* (lines 5 to 9). If the rule is of the type t.goal (t2) and its goal expressions apply, we also remove this element from the list of failed elements and mark that element as *ignored* (lines 10 to 17). The analysis of the context expressions and of the goal expressions are performed, respectively, by the procedures EvaluateContext and EvaluateGoals. After analyzing all t.context and t.goal rules for the element, if it is not yet marked as *ignored* (line 21), we will check if there is a rule of the type t.limit (t3) which `tasksSet` contains that element (line 22). If there is such a rule, we will check if the failure limit for that element was reached (line 23). If the limit was not reached yet, we will increase the failure counter of that element and mark it as *ignored* (lines 24 to 26). If the limit was reached, we will not ignore that failure - i.e, the compensation will be required - but we will reset the failure counter (line 28). As a result we return the list of failed elements (line 33), from which we removed all elements which failures were supposed to be ignored.

The EvaluateContext and EvaluateGoals procedures simply check if the rules conditions apply [14]. These procedures will not be detailed here for the sake of space.

In summary, the t.context and t.goal rules define conditions when the failure of a given task may be ignored, and t.limit rules define an amount of failures of a given task that will be ignored. However, the amount of failures defined with a t.limit rule does not take into account the failures already ignored by the t.context and t.goal rules.

In this sense, we can state that the rule types t.context and t.goal prevails upon the type t.limit. Given a t.context rule, the failure of a task in its `tasksSet` will always be ignored if its context expression is satisfied, despite how many times this failure had been ignored before. In a similar way, given a t.goal rule, the failure of a task in its `tasksSet` will always be ignored if its goal expressions hold.

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

Rules can interact. For example three rule types, from examples 9 (a t.context rule), 10 (a t.goal rule) and 11 (a t.limit rule) and the failure log depicted in Table 1. That table shows a log of failures of the *downloadEmail* task, together with the number of the failure, the value of the *calendar.day* attribute and the status of the *assistWorkExecution* goal at the moment of the failure. It also indicates if the failure was ignored as well as the rationale (the rule used for ignoring the failure).

Ex.9: *downloadEmail isAllowedToFailIf calendar.day=sunday*
Ex.10: *downloadEmail isAllowedToFailWhen assistWorkExecution=satisfied*
Ex.11: *downloadEmail isAllowedToFailAtMost 3*

In this example, the failures for which the rule of the example 9 applies were ignored: 2 and 3. In the same way, the failures 1, 3 and 6 were ignored due to the rule of the example 10. These rules do not apply for failures 4, 5, 7, 8 and 9, so we may evaluate the rule of the example 11 for these failures. The failures 4, 5 and 7 were ignored, since they were below the limit of 3 failures expressed in the rule. The failure 8, being the fourth failure of that task that were not ignored by a t.context or t.goal rule, shall be compensated, and the failure counter for that task shall be reset. Since the failure counter was reset, the failure 9 was also ignored for being below the limit of three failures.

5. Application

In order to use our approach we implemented all algorithms needed to evaluate the proposed policy. They were integrated with a simulator of our chosen self-configuration architecture [6]. We can provide a goal model, a context model and a log of context events, from which the simulator will run the Monitor - Diagnoses - Compensation cycle. We added to the simulator the ability to receive the Tolerance Policy as input as well.

We also developed wizards for making it easier to create the policy rules. Figure 3 shows an example of the creation of a t.context rule. The user selects tasks, which are extracted from the goal model, and then defines in which context that task can fail without compensation. As an example, the following rule is defined: *downloadEmail isAllowedToFailIf user.isAtAMeeting=true*. With these wizards we prevent some syntax errors that could otherwise occur.

We applied the policy rules in the example of a Smart Office system introduced in Section 2. In order to avoid the cost of downloading e-mails through a mobile phone connection at times when the e-mails are unnecessary, we defined some tolerance rules, as follows.

Assuming that when the user is at a meeting he may not need to check his e-mails, we define the following t.context rule: *downloadEmail isAllowedToFailIf user.isAtAMeeting=true*. Assuming also that it is not required to have his e-mails updated when the system has already finished assistang the user in performing his work, we define this t.goal rule: *downloadEmail isAllowedToFailWhen assistWorkExecution=satisfied*. Finally, accepting that the e-mail downloading can fail at most three times consecutively, the following t.limit rule is stated: *downloadEmail isAllowedToFailAtMost 3*.

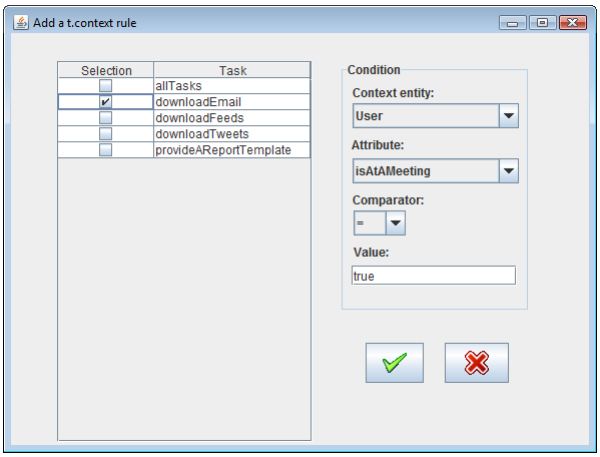

Figure 3 - Screenshot of the wizard for creating a t.context rule

We ran the simulator applying only one rule at a time, considering the scenario of one typical day of work, but on which the Ethernet connection was always down. The average result was a decrease of 46% on the number of required compensations. This result shows that, in some situations, the use of a tolerance policy can reduce the overall cost of using a

Table 1 - Failures log of the task *downloadEmail*

# failure	calendar.day	assistWorkExecution	Ignore failure?	Rationale
1	Saturday	Satisfied	Yes	Ex. 10
2	Sunday	Not satisfied	Yes	Ex. 9
3	Sunday	Satisfied	Yes	Ex. 9, Ex. 10
4	Monday	Not satisfied	Yes	Ex. 11 (1st failure)
5	Monday	Not satisfied	Yes	Ex. 11 (2nd failure)
6	Monday	Satisfied	Yes	Ex. 10
7	Tuesday	Not satisfied	Yes	Ex. 11 (3rd failure)
8	Tuesday	Not satisfied	No	
9	Tuesday	Not satisfied	Yes	Ex. 11 (1st failure)

system, without a significant impact on the system behavior.

6. Related Work

In this work we applied the Tolerance Policy in connection with Dalpiaz architecture [6]. Despite the existence of a Tolerance Policy component in the original architecture, their work did not define a set of rule types, neither how they could be applied. Thus, in our work we have provided a more fine-grained control on the failure handling mechanism of that architecture, which results in a smaller amount of compensations to be performed during a system execution.

There are some other architectures for self-configuring, self-managing and autonomic systems [11][12][13]. However, for the best of our knowledge, none of them provide this level of failure control.

7. Conclusions

In this paper it was presented a tolerance policy that deals with failure occurrences. The objective was to increase the flexibility of failure handling in self-configuring systems, using tolerance policies based on requirements models. In particular we can express conditions in which a failure may be ignored. These conditions are related to requirements models – more specifically, a goal model and a context model. In order to make a proof of the concept, we defined algorithms and proposed a Policy Editor tool. This editor makes it easier for the user to create and maintain the rules of a policy. A simple example was used to explain our approach.

For the future, we 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 handle more complex rules, which can mix different types of a rule. We also need to apply these policies mechanisms in a real-world software system, analyzing the usefulness and the effectiveness of our approach. Lastly, we are interested in investigating how our policy could be used in different architectures, i.e. moving towards a more generic tolerance policy.

8. 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 sponsored by FACEPE, CNPQ and CAPES.

9. References

[1] Horn, P. Autonomic computing: IBM's Perspective on the State of Information Technology. [S.l.]: IBM, 2001.

[2] 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.

[3] Müller, H. A.; O'Brien, L.; Klein, M.; Wood, B. Autonomic Computing. CMU/SEI-2006-TN-006. [S.l.], 2006.

[4] Salehie, M.; Tahvildari, L. Autonomic computing: emerging trends and open problems. SIGSOFT Softw. Eng. Notes, ACM, New York, NY, USA, v. 30, n. 4, p. 1-7, Julho 2005. ISSN 0163-5948. Available in http://dx.doi.org/10.1145/1082983.1083082

[5] Parekh, J.; Kaiser, G.; Gross, P.; Valetto, G. Retrofitting autonomic capabilities onto legacy systems. Cluster Computing, Kluwer Academic Publishers, Hingham, MA, USA, v. 9, n. 2, p. 141-159, 2006. ISSN 1386-7857.

[6] Dalpiaz, F.; Giorgini, P.; Mylopoulos, J. An architecture for requirements-driven self-reconfiguration. In: ECK, P. van; GORDIJN, J.; WIERINGA, R. (Ed.). CAiSE. [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5565), p. 246-260. ISBN 978-3-642-02143-5.

[7] Giorgini, P.; Mylopoulos,, J.; Perini, A.; Susi, A. The Tropos Metamodel and its Use. In: Informatica journal, 2005.

[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] 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.

[10] 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.

[11] Anthony, R.; Pelc, M.; Ward, P.; Hawthorne, J.; Pulnah, K. A run-time configurable software architecture for self-managing systems. Autonomic Computing, International Conference on, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 207-208, 2008.

[12] 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.

[13] Subramanian, L.; Katz, R. H. An architecture for building self-configurable systems. In: MobiHoc '00: Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing. Piscataway, NJ, USA: IEEE Press, 2000. p. 63-73. ISBN 0-7803-6534-8.

[14] Pimentel, J.H.C. 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.