Towards Self-Protecting Enterprise Applications

Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè
Università degli Studi di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy
{lorenzoli,mariani,pezze}@disco.unimib.it

Abstract

Enterprise systems must guarantee high availability and reliability to provide 24/7 services without interruptions and failures. Mechanisms for handling exceptional cases and implementing fault tolerance techniques can reduce failure occurrences, and increase dependability. Most of such mechanisms address major problems that lead to unexpected service termination or crashes, but do not deal with many subtle domain dependent failures that do not necessarily cause service termination or crashes, but result in incorrect results.

In this paper, we propose a technique for developing self-protecting systems. The technique proposed in this paper observes values at relevant program points. When the technique detects a software failure, it uses the collected information to identify the execution contexts that lead to the failure, and automatically enables mechanisms for preventing future occurrences of failures of the same type. Thus, failures do not occur again after the first detection of a failure of the same type.

1 Introduction

Enterprise systems are long living applications that integrate persistent and transaction-based services to offer core business functionalities to large populations of users who continuously access enterprise applications to meet relevant business objectives [8]. Dependability properties, such as availability, safety and reliability, are essential quality attributes, and enterprise systems are thoroughly tested throughout all development phases, from system design to deployment, to verify the satisfaction of such essential properties. However because of the complexity of these systems, faults cannot be completely eliminated from deployed applications [3]. Due to the continuous accesses from many users, failures can be experienced repeatedly before identifying the responsible faults and before developing, testing and deploying suitable patches. Failure prevention techniques aim at mitigating the problem of recurrent failures by protecting systems from failure occurrences, while waiting for faults to be identified and removed.

Failure prevention techniques can be roughly classified as *failure-specific* and *general* techniques. Failure specific techniques are based on design-time prediction of failures that are likely to occur at run-time, and on the design of mechanisms to protect the system from the occurrence of the predicted failures. Common failure specific techniques are exception handling and defensive programming [21, 29]. We can for example design exception handlers to manage accesses to non-existing file, even if such events should not happen. Failure-specific techniques can capture a limited subset of potential faults, but do not protect from problems that are not predicted and handled at design-time.

General techniques are based on mechanisms that handle catastrophic events that do not depend on the specific application, such as transactional services and fault-tolerance mechanisms [25, 1]. Fault tolerance mechanisms can for example automatically hide single algorithmic faults by replicating computations. General techniques do not require predicting the exact cause of failures, but are mostly limited to domain independent failures, such as system crashes and hardware failures.

In this paper, we propose a self-protection technique, called FFTV (From Failures To Vaccine), that increases reliability of enterprise applications by capturing the context of observed failures, and preventing failures of the same type from occurring again. Initially, FFTV monitors system executions, extracts information related to failures, and creates models that describe the context of failures, that is the sequence of events that led to failure. Successively, FFTV matches executions with failure context models. When FFTV identifies an execution that matches a failure context, it activates suitable protection mechanisms that either prevent the system from failing or heal the problem and enable the successful termination of a running functionality that

would fail otherwise. In this paper, we use transactional services to dynamically encapsulate dangerous executions into safe execution contexts, in order to prevent loss of data or reaching faulty states. The technique proposed in this paper resumes the application into a "safe" state, that is a state that allows users to continue interact with the system.

FFTV is not an alternative to failure-specific and general mechanisms, but it complements the set of problems dealt by existing techniques. Failure specific mechanisms prevent failures related to exceptional events that can be predicted at design-time. General mechanisms prevent failures that depend on general catastrophic events. FFTV prevents specific problems that are difficult to predict at design-time, but can be identified at run time by learning from program executions.

FFTV can address different classes of problems by capturing different information and distilling different models. In this paper, we exemplify the approach by referring to problems caused by unexpected data values passed between components or assigned to state variables.

The paper is organized as follows. Section 2 presents the overall approach. Section 3 illustrates the design-time activities required to develop FFTV-enabled systems. Section 4 describes the generation of models used to identify unexpected values. Section 5 shows how to build failure contexts from observed failures. Sections 6 and 7 detail the techniques for analyzing and detecting failures that comprise the building of failure contexts. Section 8 shows how transactional services can be used to implement protection mechanisms. Section 9 presents early experimental results obtained by applying FFTV to prevent a specific class of faults injected into the Sun PetStore enterprise system. Section 10 discusses related work, and Section 11 presents conclusions.

2 FFTV: From Failure to Vaccine

FFTV monitors system executions, identifies contexts associated with observed failures, that is the conditions under which the system failed, and activates mechanisms to protect the systems when executions match failure contexts.

Figure 1 illustrates FFTV. At *design-time*, developers define the fault types addressed by FFTV, and design oracles that detect these faults. Oracles are then compiled into the target system. Fault types and oracles can be specified manually, for instance, by specifying assertions [19], or derived automatically from specifications, for instance, by compiling high-level properties into code-level assertions [13]. Once oracles have been defined, FFTV uses static analysis techniques to automatically identify the program points that can affect the properties defined by the oracles. These program points are monitored at testing-time to extract information about successful executions, that is executions that pass the checks computed by oracles.

In this paper, we instantiate the technique on state-based faults, that is faults that depend on executions that erroneously change the state of the execution, and then use the corrupted state, leading to a failure. To capture these faults, we use JML assertions to write oracles that can identify unexpected values [16], and def-use analysis to identify the state variables that can affect the evaluation of existing assertions [12]. So far, we have generated assertions manually, but we are working on automatic generations of assertions from specifications.

At *testing time*, FFTV records the information flowing through relevant program points, and stores the data extracted from executions of successful test cases into a repository. After the testing sessions, FFTV uses these data to automatically derive models that represent in a compact and general way properties related to relevant program points.

In this paper, we apply FFTV by recording values of the state variables identified by data-flow analysis, and we use Daikon [7] to infer general relations, expressed as Boolean expressions, between these state variables. Column *Models* in Table 1 shows examples of properties that can be automatically inferred with Daikon.

At *run-time*, FFTV analyses failures, and detects executions that may lead to failures of the same type, that is, it identifies both the causes of failures (*failure analysis*), and the executions that can likely lead to failures similar to the identified ones (*failure detection*).

Failure analysis is based on failures detected by oracles. The *observer* analyzes the data monitored at relevant program points, and compares these data with the models built at testing time, in order to identify the causes of experienced failures. The analysis produces failure contexts, which are used to identify situations that will likely lead to failures of the same type of those experienced in the past. The types of failures that are addressed by failure contexts depend on inference engines. In this paper, we detect failures that derive both from state-based faults, that is unexpected values assigned to variables, and from interface problems, that is unexpected values assigned to parameters. In both cases, we detect unexpected values with Boolean expressions.

During normal system execution, FFTV compares the data monitored at relevant program points with the recorded failure contexts. If the execution matches a failure context, FFTV activates protection (or healing) mechanisms to prevent system failure, thus increasing the dependability of the software system.

In this paper, we design protection mechanisms for software systems that implement the MVC design pattern, a widespread adopted pattern for designing three layered enterprise systems [23]. In the MVC design pattern, each action that can be executed by users is clearly represented with an individual entity (usually a class or a method).

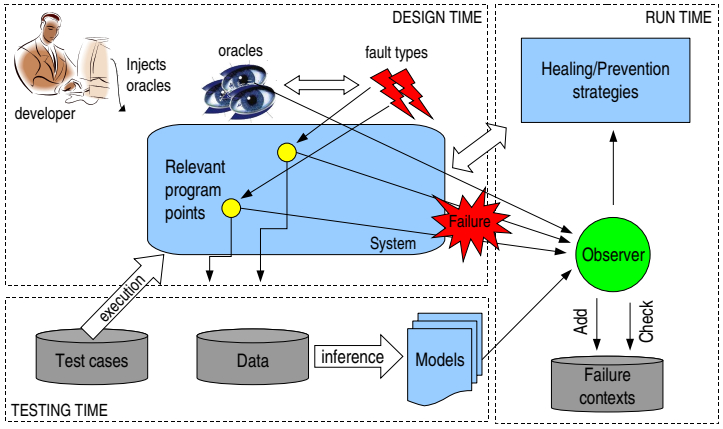


Figure 1. The FFTV approach

We protect these systems by encapsulating dangerous executions within transactions (transaction services are always available in J2EE compliant application servers and are widely adopted in enterprise systems in general [14]). According to MVC, we identify the stable states as the states of the system between the execution of consecutive actions. When FFTV identifies the execution of actions that can lead to system failures, it dynamically embeds potentially dangerous actions within a transaction. If the software system fails, FFTV resumes the system execution from the last saved stable state, thus missing the failing transactions but allowing users to continue interact with the system and execute other actions. Users who perform the same action again, will experience similar detect-resume interactions.

The FFTV approach can be instantiated with different techniques at design, testing and run-time, to capture and heal different types of failures. We are currently investigating finite state machines and temporal logic as oracles to dynamically capture component integration problems, inter-component invocations as relevant program points, finite state machines inferred with kBehavior [18] as model for interactions, and equivalent scenarios as healing technique [4].

3 Oracles and Relevant Program Points

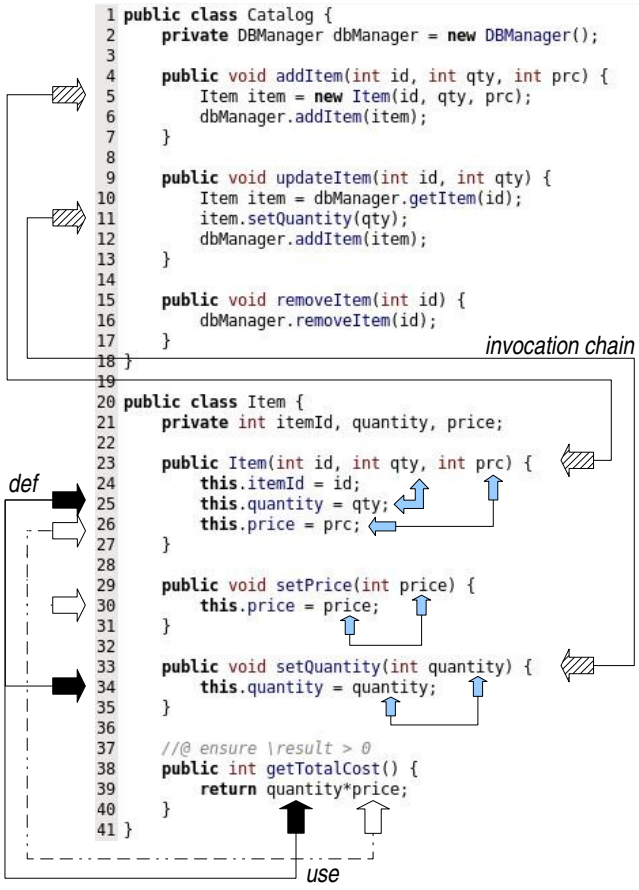
Oracles check the properties selected by developers according to the criticality of the components, the complexity of the implementations and the importance of the functionalities. For examples, in the case of a functionality that registers new users into the system, oracles can verify that at least user name and password are correctly recorded in the database.

In this paper, we consider oracles specified as code-level assertions [6], which are particularly well suited to reveal failures that depend on wrong values of state variables and parameters. We use the Java Modeling Language (JML) [17] to indicate the conditions over program variables that are expected to hold at run-time. JML annotations are Java comments prefixed with `@`. Line 37 of Figure 2 shows an example of JML assertion that specifies that the return value of method `getTotalCost` must be greater than 0 (`@ensure /result` $>$ 0). When an assertion is violated, a generic `AssertionException` is thrown.

We detect failures that depend on both violations of user-specified oracles, and uncaught exceptions. Since we expect that no exceptions should be observed by the controller of a MVC-based system, we treat all exceptions observed by the controller as failures.

Given a set of JML assertions, we automatically analyze the application to identify the relevant program points that can influence the oracles. We then instrument the identified points to extract the information needed for identifying failure contexts. We identify the program variables that can impact the evaluation of the expressions computed in JML assertions by means of data-flow analysis [12]. If a statement s assigns a value to a variable v, we say that s defines v; if a statement s references to a variable v, we say that s uses v. If there exists an execution path p from a statement s_1 that defines variable v, to a statement s_2 that uses the same variable v, and no statements in p (with the exception of s_1) define v, we say that p is a definition-clear path with respect to variable v and (s_1^v, s_2) is a definition-use pair for v. A sequence $(s_1^{v_1}, s_2^{v_2}, s_3^{v_3}, \ldots, s_k)$, where each pair $(s_i^{v_i}, s_{i+1})$ is a definition use pair, and in node s_{i+1} the use of variable v_i influences the definition of variable v_{i+1}, is a definition-use chain of length k [11, 10].

Given a variable v and a statement s where v is used, we denote with $chain_k^v(s)$ the set of all definition-use chains $(s_1^{v_1}, s_2^{v_2}, \ldots, s_{k-1}^v, s)$. Given a definition-use chain c =




```
1  public class Catalog {
2      private DBManager dbManager = new DBManager();
3
4      public void addItem(int id, int qty, int prc) {
5          Item item = new Item(id, qty, prc);
6          dbManager.addItem(item);
7      }
8
9      public void updateItem(int id, int qty) {
10         Item item = dbManager.getItem(id);
11         item.setQuantity(qty);
12         dbManager.addItem(item);
13     }
14
15     public void removeItem(int id) {
16         dbManager.removeItem(id);
17     }
18 }
19
20 public class Item {
21     private int itemId, quantity, price;
22
23     public Item(int id, int qty, int prc) {
24         this.itemId = id;
25         this.quantity = qty;
26         this.price = prc;
27     }
28
29     public void setPrice(int price) {
30         this.price = price;
31     }
32
33     public void setQuantity(int quantity) {
34         this.quantity = quantity;
35     }
36
37     //@ ensure \result > 0
38     public int getTotalCost() {
39         return quantity*price;
40     }
41 }
```

Legend
Hyperedges relate definitions and uses. Hyperedges are graphically identified with arrowhead patterns. Def-use chains can be obtained by recursively following hyperedges.

Figure 2. An example of program with an assertion and a def-use chain.

$(s_1^{v_1}, s_2^{v_2}, \ldots, s_k)$, we use $stat(c)$ to denote the set of all the statements included in c, that is $stat(c) = \{s_1, s_2, \ldots, s_k\}$. Given v used in statement s, $stat_k^v(s)$ is the set of all the statements, up to a depth k, that may influence the value of v in s, that is $stat_k^v(s) = \bigcup_{c \in chain_k^v(s)} stat(c)$.

Given a JML assertion J and variables v_i used in J, the set of statements that may influence the value of v_i is given by $stat_k^{v_i}(J)$. The set of relevant program points associated with J is given by $\bigcup_{v\ used\ in\ J} stat_k^v(J)$, where k is a parameter. Large values of k provide higher coverage of relevant program points.

Let us consider for example the assertion `result`> 0 associated with method `getTotalCost` shown in Figure 2. The definition-use chains of length 3 for this assertion are indicated by hyperedges that relate definitions and uses of variables. The set of relevant program points can be obtained starting from the assertion `result`> 0 and selecting the program points reachable by following the hyperedges. In this example, the set of relevant program points is the set of statements at lines $4, 5, 9, 11, 23, 25, 26, 29, 30, 33, 34, 39$.

4 Model Generation

At *testing-time*, test designers execute test suites and collect data from relevant program points. To generate models of successful behavior, we consider only the data obtained from successful executions of test cases. The type and amount of information extracted by FFTV depend from both the considered types of failures and the technologies used to prevent the failures or heal the related faults. In this paper, we extract information from relevant program points by recording the values of the variables when they are defined both before and after the execution of a relevant program point, and the values of the variables when they are used before a relevant program point.

FFTV automatically produces general models of the observed behaviors from the data collected during testing. When executing the deployed application, FFTV uses these models to identify executions that differ from the behavior observed at testing-time, and thus deserve further attention. In this paper, we produce general models of the observed behavior with the Daikon inference engine [7], which can identify relations between sets of variables at given program points. Properties are expressed as Boolean expressions. The models produced by FFTV represent relations of values observed during execution, thus the accuracy of the models depend on the quality of the test cases, which roughly correspond to the coverage of the execution space. According to our early experience, a well design set of test cases generates accurate models of the software system.

Let us consider, for instance, the program point at line 25 in the class `Item` shown in Figure 2. Daikon can infer properties like `item.quantity'` ≥ 1 and `item.quantity'` $= qty$. The former property specifies that the final value of the quantity is always greater or equal than 1, while the latter indicates that the final value of quantity corresponds to the value of variable qty.

5 Failure Contexts

When running in the field, FFTV monitors the values of the variables at relevant program points, identifies *anomalous values*, which are values that violate models derived at testing-time, detects failures by means of oracles, and indicates when actions start and finish.

When a new type of failure is observed, FFTV computes its failure context, that is the sequence of events that led to the failure. The failure context cannot prevent the first occurrence of the failure, but can be used to detect future occurrences of events that may lead to other failures of the same type, and prevent further failure occurrences.

6 Failure Analysis

When oracles reveal failures, the information extracted from controllers at relevant program points is used to automatically build failure contexts. Failure contexts capture the anomalous run-time conditions that occurred before the observed failures. In this paper, we generate failure contexts by using both the sequence of the last $k+1$ actions (action $k+1$ is the one that cannot be successfully completed because it generates a failure) and the anomalous values that are detected at relevant program points (k is a parameter of the technique).

The rationale is that many failures occur within executions that are characterized by particular combinations of actions and data values. Thus, by examining the last $k+1$ actions, we can detect a set of unexpected events related to a type of failure experienced in previous executions. Aim of FFTV is not to capture either the exact source of failures or the complete sequence of events that lead to a failure, but to identify few anomalous values that are usually observed before the occurrence of a failure. These values are used as failure predictors.

As illustrated by the example in the next subsection, we use non-disruptive prevention and healing strategies. Thus, false positives, that is sequences of events that are wrongly identified as potentially leading to failures, do not introduce additional failures. However, false positive introduces overhead caused by the unnecessary activation of prevention and healing mechanisms, thus it is important to reduce the amount of occurrences of false positive, to avoid performance problems. We solve the problem by introducing a confidence threshold $conf$ that estimates the quality of failure contexts and allows us to discard low quality contexts. When the percentage of false positive identified by a given failure context exceeds $conf$, we consider the context not relevant and we discard it. We record deleted failure contexts to avoid their re-instantiation.

The use of a limited window of size $k+1$ to identify events that can be included into failure contexts allows for several optimizations. In particular, the observer needs only to preserve information about the last k actions, and each time a new action is observed, information about the oldest can be discarded. In presence of a reasonable amount of available memory and with a limited k, the observer can work by keeping run-time data on primary memory only, without accessing secondary memories, and thus with small overhead.

FFTV generates failure contexts from failures, the last $k+1$ actions and corresponding model violations in three steps. In the *first step*, FFTV summarizes the sequence of the first k actions (the action that produces the failure is not considered) and the corresponding model violations with an annotated FSM. For example, Figure 3 shows the annotated FSM corresponding to the sample execution presented in Table 1. Transitions represent actions and annotations indicate the models that have been violated by the corresponding action. Note that only actions that refer to program points occur in failure contexts. Actions that do not refer to program points are excluded since they do not contribute to the failure.

Annotated FSMs represent executions at the granularity of actions, and detect anomalous events at the granularity of program points. Specifying actions as labels of FSM transitions abstracts from low level details, such as sequences of method executions, that would introduce useless details in the model. Using program points as indicators of violations, such as unexpected variable values, captures low level faults, even if the model focuses on high-level information.

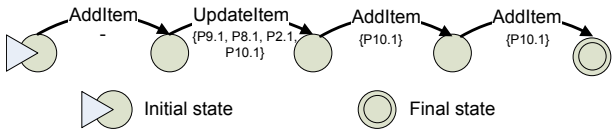


Figure 3. The annotated FSM associated with the failure shown in Table 1

In the *second step*, FFTV augments the annotated FSM with weights that indicate the relevance of the transitions (actions and corresponding violations) with respect to the identification of a failure. We assign weights by following these rules:

- actions associated with no violations are likely not to contribute to failure contexts, thus they are assigned a low weight equals to 1.
- in general, anomalous values do not generate a single model violation, but cause several model violations across different actions. Actions associated with model violations that occur early in the sequence are usually more important than actions associated with model violations that occur late in the sequence, because late anomalous values are often caused by anomalous values generated early in the execution sequence. The following formula assigns to actions with anomalies a weight proportional to the occurrence of the actions in the sequence of anomalous actions, by considering both homogeneous and heterogeneous actions:

Action	Method	Variables traced at program points	Models		Violations
			Boolean expression	ID	
Add item	GUI.addItem	P6.quantity P7.price	P6.quantity > 0 0 < P7.price <= 2000	P6.1 P7.1	-
	AddItemAction.<init>	P4.quantity P5.price	P4.quantity > 0 0 < P5.price <= 2000	P4.1 P5.1	-
	Item.setQuantity	P2.quantity	P2.quantity > 0	P2.1	-
	Item.setPrice	P3.price	0 < P3.price <= 2000	P3.1	-
Remove item	-	-	-		-
Update item	GUI.updateItem	P9.quantity	1 <= P9.quantity <= 13	P9.1	P9.quantity = -1
	UpdateItemAction.<init>	P8.quantity	1 <= P8.quantity <= 13	P8.1	P8.quantity = -1
	Item.setQuantity	P2.quantity	1 <= P2.quantity <= 13	P2.1	P2.quantity = -1
	Item.getQuanity	P10.quantity	1 <= P10.quantity <= 13	P10.1	P10.quantity = -1
Add item	GUI.addItem	P6.quantity P7.price	P6.quantity > 0 0 < P7.price <= 2000	P6.1 P7.1	-
	AddItemAction.<init>	P6.quantity P7.price	P4.quantity > 0 0 < P5.price <= 2000	P4.1 P5.1	-
	Item.setQuantity	P4.quantity	P2.quantity > 0	P2.1	-
	Item.setPrice	P5.price	0 < P3.price <= 2000	P3.1	-
	Item.getQuanity	P10.quantity	1 <= P10.quantity <= 13	P10.1	P10.quantity = -1
Add item	GUI.addItem	P6.quantity P7.price	P6.quantity > 0 0 < P7.price <= 2000	P6.1 P7.1	-
	AddItemAction.<init>	P6.quantity P7.price	P4.quantity > 0 0 < P5.price <= 2000	P4.1 P5.1	-
	Item.setQuantity	P4.quantity	P2.quantity > 0	P2.1	-
	Item.setPrice	P5.price	0 < P3.price <= 2000	P3.1	-
	Item.getQuanity	P10.quantity	1 <= P10.quantity <= 13	P10.1	P10.quantity = -1
Purchase item	ShoppingCart.getTotalCost	P1.totalCost P1.quantity P1.price P1.totalCost'	P1.totalCost < P1.totalCost' P1.totalCost' >= P1.quantity*P1.price ...	P1.1 P1.2 ...	P1.quantity = -1 P1.totalCost' = -320

Legend

The table shows an observed execution that reveals a failure. A user executes the sequence of actions indicated in the first column of the table: starting with `AddItem`, and terminating with `PurchaseItem`. Column `Method` indicates the methods with relevant program points that are called while executing the corresponding actions (the methods are either directly executed by the action, or invoked by the View after the last action has been completed and before the next action is executed.) The program points considered in this example are obtained from a single assertion placed in method `getTotalCost`, as shown in Figure 2. The third column indicates the program points and variables related to methods shown in column 2. For instance, the method `GUI.addItem` includes two program points $P6$ and $P7$. Program point $P6$ traces the value of the variable `quantity` and program point $P7$ traces the value of the variable `price`. Column `models` indicates the models that have been derived from variables traced at program points. Column `ID` provides a unique name to each model. The last column `Violations` indicates the models that are violated by the considered execution, and the values that violate the models.

In summary, this table shows an execution of the sequence of actions `AddItem`, `RemoveItem`, `UpdateItem`, `AddItem`, `AddItem`, `PurchaseItem`, where action `UpdateItem` violates models $P9.1$, $P8.1$, $P2.1$ and $P10.1$, the last two `addItem` actions violate model $P10.1$ and action `PurchaseItem` violates both models $P1.1$ and $P1.2$ and the assertion in method `getTotalCost`.

Table 1. Example of run-time data for a shopping cart application

$$w = \frac{200}{k_{diff}^{n_{diff}} + k_{same}^{n_{same}}} \quad (1)$$

where

- k_{diff} is a reduction factor that indicates the relevance of an action based on the early presence in the sequence of actions of different type associated with model violations
- k_{same} is a reduction factor that indicates the relevance of an action based on the early presence in the sequence of actions of the same type associated with model violations
- n_{diff} is the number of actions associated with model violations that both differ from the current action and have been executed before the current action
- n_{same} is the number of actions associated with model violations both of the same type of the current action and have been executed before the current action.

Assigning large values to parameters k_{diff} and k_{same} increases the relative relevance of actions associated with model violations depending on how early they occur in the sequence compared to other actions associated with model violations, while values close to 1 do not discriminate the relevance of the violations. In general, $k_{same} > k_{diff}$, because repeated observations of actions associated with the same model violations are likely to be repeated observations of the same problem, which usually have a small impact on the whole failure context.

For instance, with values $k_{diff} = 2.5$ and $k_{same} = 5$, we generate the weights shown in Figure 4 for the FSM shown in Figure 3.

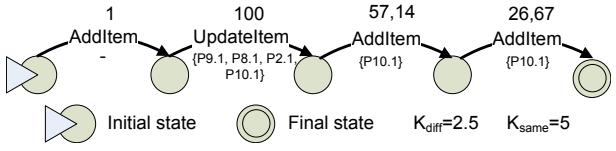


Figure 4. The weighted FSM associated with the failure shown in Table 1

In the *third step*, FFTV generates the failure context from the weighted FSM. A failure context is composed of two elements: the activating action and the contextual events. The activating action is the $k + 1$ action that generated the failure. For instance, in the case shown in Table 1, the activating action is `PurchaseItem`. The contextual events are the events indicated in the weighted FSM, including the set of model violations and the weights associated to the corresponding actions. For instance, Figure 5 shows the failure context associated with the weighted FSM shown in Figure 4 .

Activating Action:	PurchaseItem		
Contextual Events:	AddItem	-	1
	UpdateItem	P9.1,P8.1,P2.1, P10.1	100
	AddItem	P10.1	57,14
	AddItem	P10.1	26,67

Figure 5. The failure context associated with the failure shown in Table 1

7 Failure Detection

Failure contexts include sets of actions, either associated with anomalous values or not, that have been recorded before a failure and are likely to be related to the failure occurrence. Each failure context is associated with an activating action. Whenever FFTV detects an activating action a during the execution of the system, it compares the last k actions that have been executed before a with all failures contexts that include a as activating action. If a failure context indicates that the execution of a can cause problems, suitable healing or prevention strategies are activated. FFTV checks for the presence of a potentially failing execution when the execution of an activating action is requested by the user, but before the action itself is executed, and thus before the possible failure.

Failure contexts can be checked by measuring the number of actions in the contextual events that have been executed within the last k actions. We consider an action in the contextual events as executed when both the name of the action and at least cov of its anomalies, i.e., the set of violated models, match ($0 \leq cov \leq 1$ indicates the minimum values for the ratio between the number of executed anomalies and the total number of anomalies to be executed). In our early experiments, we set cov to 0.8. For instance, if we consider the failure context shown in Figure 5, and we consider the execution `AddItem AddItem AddItem UpdateItem PurchaseItem`, and only `UpdateItem` generates the anomalies $P2.1$, $P8.1$, $P9.1$ and $P10.1$, we executed 2 out of 4 actions. The weights of the 2 executed actions is $1 + 100 = 101$.

We do not require the complete matching of a failure context because failure contexts include both useful information and "noise", that is actions that are not directly related to the failure. We usually activate a failure context even in presence of the execution of a small fraction of the actions occurring in the context. This approach is facilitated

by the use of non-disruptive healing/prevention strategies, which do not cause additional problems in presence of false positives. The matching of a failure context is measured by weighting the actions as follows:

$$coverage = \frac{\sum_{a\ covered\ action} w(a)}{\sum_{a \in failurecontext} w(a)} \quad (2)$$

where $w(action)$ indicates the weight of the action in the considered failure context. We match a failure context when the matching measured with the formula above is beyond an $activationrate$ threshold. We usually set $activationrate = 0.5$, because executing half of the weighted actions occurring in a failure context indicates that either actions that correspond the most important anomalies or many anomalies with low weights have already occurred. For instance, the 2 actions executed by the example execution above provide a matching measure of $\frac{101}{184.81} = 0.55 > 0.5$; thus, FFTV activates the failure context. While, an execution sequence `addItem`, `addItem`, `addItem`, `removeItem`, `PurchaseItem`, with actions that generate no violations, includes 1 action only with a total weight of 1. The matching level is $\frac{1}{184.81} = 0.01 < 0.5$, thus FFTV would not activate the failure context.

Our technique depends on 5 parameters: k, k_{diff}, k_{same}, cov and $activationrate$. We provided default values for all parameters. Fine tuning the parameters according to empirical evidences is part of our ongoing research work.

In addition to matching-based identification of failure contexts, which considers the number of actions with anomalies that are executed, as illustrated above, but does not consider the type of violations, we use an identification technique based on the type of anomalies. In particular, given a failure detected by a violation or a caught exception, we identify a set of violations that are strictly related to the failure. For instance, in the case of failures revealed by capturing `NullPointerExceptions`, relevant violations are given by models violated with `null` values. In the case that any of the actions in a failure context is violated with a relevant violation, the failure context is activated, independently from the matching. This mechanism supports identification of specific problems that are usually caused by particular values assigned to variables. The definition of relevant violations can be generalized to other exceptions and assertions. For instance, it is possible to relate the `OutOfBoundException` to assertions that limit the range of values assigned to a variable. We have currently defined this mechanism only for the `NullPointerException`. Generalizing this strategy to other exceptions and assertions is part of future work.

8 Prevention Mechanisms

In this paper, we implemented a non-disruptive prevention mechanism by using the transaction service available in J2EE application servers. We extended the controllers of enterprise systems in the following way: When actions are executed, controllers ask the observer to identify potentially dangerous actions, that is actions that activate failure contexts. If the observer does not identify the action as potentially dangerous, the action is executed normally. If the action is identified as potentially dangerous, the controller initiates a transaction before executing the action normally. If the action is executed successfully, the transaction is simply committed, with limited impact on on system performance. If an oracle identifies a failure, the system propagates an exception of type `AssertionException` to the controller, which rollbacks the execution to a correct state, and suitably warn the user. The scope of transactions activated to prevent failures matches execution of single actions identified as dangerous. The mechanism recover from failures, and users can continue working with the system.

9 Early Validation

We validate the FFTV approach by measuring the ability of the technique to generate failure contexts for a specific class of problems: failures caused by unexpected values that are processed by enterprise systems, either because the application erroneously accepts incorrect inputs from users or because of corrupted data in databases. We focused on this class of problems for the initial validation because enterprise systems with complex user interfaces and input from external sources often suffer from such problems.

We conducted the first experiments with the Sun Petstore enterprise system version 1.4 [24], a sample application developed by Sun Microsystems to demonstrate the features of J2EE 1.4 application servers. The Sun Petstore implements a classic web shop that includes functionalities like user authentication, cart management, catalog browsing and administration. We designed 8 assertions that check for the correctness of the data used to ship orders. To evaluate the capability of FFTV to create failure contexts for the considered class of faults, we removed all consistency checks on both input values and data extracted from the database. We executed the Sun PetStore with test cases that focus on boundary or incorrect inputs, for instance, test cases that add users with incorrect shipping addresses, buy negative quantities of items, and insert incorrect values in the database. We revealed a total of 9 failures. FFTV captured all these problems by creating 9 suitable failure contexts. In one case, FFTV recovered from a failure not easily revealed with classic approaches, in fact the total cost of items in the cart is always displayed as a positive value, even when is a

negative number. Thus, users cannot distinguish carts with negative prices from carts with positive ones, by simply inspecting the output.

The early validation also highlighted the effectiveness of def-use chains to select relevant program points. In fact, def-use chains automatically identify actions that may appear unrelated but can interfere during system execution, for instance, action `addUser` generates information relevant for the successful execution of action `PurchaseOrder`. Moreover, def-use chains automatically discard actions that are not related from the execution viewpoint, even if they apparently working on similar components, for instance, action `getItem` is not relevant for action `PurchaseOrder`, even if they work on similar data structures. This information is extremely useful when creating failure contexts, because irrelevant actions that can hide important actions are automatically filtered. For example, the execution of several `getItem` actions will never affect an `addItem` action in failure contexts associated with a problem in action `PurchaseOrder`.

The experience gained so far with the Sun Petstore indicates that the extra statements added by our technique introduce a limited overhead, not perceivable by end-users. In fact at each program point, FFTV evaluates only simple Boolean expressions; FFTV evaluates failure contexts only when the observer detects activating actions. Moreover, evaluating failure contexts consists of matching two sets of a maximum size K, which is fast, especially if the available memory allows for the operation to be performed without accessing secondary memory.

In this early experience with FFTV, we studied effectiveness of the technique when testers specify a limited set of focused assertions (in our investigation we focused on correctly shipping user orders). We are currently extending the validation to a wider class of faults, and we are studying the scalability of the technique to a large number of assertions.

10 Related Work

Self-protecting techniques have been widely studied for hardware systems [15, 2]. The many techniques for design for testability, Built-In Self Test and Built-In Self Repair are effective for hardware devices, but focus mostly on manufacturing faults, and are based on fault models not shared with software systems, and do not apply well to many relevant software problems.

Classic failure prevention techniques for software systems can be classified in two main groups: failure specific and general techniques. Failure specific techniques support developers in defining proper procedures to handle problems that can be predicted at design-time. Main approaches are based on the development of defensive code and design of checking mechanisms, e.g., defensive programming [29], assertions [6], self-checking systems [27] and exception handlers [21].

General techniques aim to preserve system functionalities in case of catastrophic events. The main techniques have been studied for safety critical applications by the fault-tolerance community [1]. Common fault tolerance approaches include redundancy, service relocation, and transactional services [25], to add dynamic recovery mechanisms, and rejuvenation features [26], to prevent aging problems, that is systems with a state that degrades over time potentially leading to failures.

Failure specific techniques can handle problems that can be predicted at design-time, but they cannot deal with failures difficult to predicted before deployment, such as environmental problems, for instance problems that derive from specific context of use, or configuration problems, for instance problems that derive from the integration of the application with other systems, or domain dependent problems, for instance unexpected ranges of values in a specific domain.

General techniques can deal with general catastrophic events, but cannot effectively cope with application specific problems that do not produce catastrophic effects.

Several techniques to prevent failures and design healing strategies complement the aforementioned techniques. For instance, Zachariadis, Mascolo and Emmerich defined a framework for designing mobile systems that can automatically adapt to different environmental conditions [28]. Cheng, Garlan and Schmerl defined a language that supports the definition of domain-level objectives dynamically enforced on a target system [5]. Modafferi, Mussi and Pernici defined a plug-in to add recovery mechanisms to WsBPEL engines [20]. Several researchers outlined the idea of using external models to support adaptation and healing mechanisms, and implementing healing and prevention procedures by changing architectures and using advanced services such as transactions [22, 9].

These approaches provide ideas or framework to support definition of complex techniques, but do not propose complete prevention or healing techniques. In this paper, we present a complete technique to detect failure conditions at run time, and to prevent repeated failures of the same type in enterprise systems. Our work represents an initial step towards the definition of a complete self-prevention technique for many types of functional faults in enterprise systems.

11 Conclusions

Since Enterprise systems usually offer $24/7$ services, reliability and high availability are extremely important. Classic fault-tolerance approaches can cope with catastrophic events that can be predicted at design-time, but are not effective with unpredictable environmental, configuration,

and domain dependent failures.

In this paper, we present a self-prevention technique that automatically generates failure context models, which capture execution conditions that may lead to failures, and uses these models to prevent future occurrences of recurrent failures.

Early validation shows that the technique works with limited overhead, and can effectively prevent some classes of failures - preliminary experiments have been conducted with failures that depend on incorrect users inputs, or corrupted data stored in databases.

We are currently conducting a large set of experiments to fully validate the technique and to empirically identify the scope of the approach. We are also extending the technique in several directions: Use of false positives, that is failure context models triggered by correct executions, to incrementally refine failure context models, for instance, by tuning actions, weights and threshold values; Automatic generation of oracles from system specification [13]; Instantiation of the general FFTV approach to other classes of problems and systems.

Acknowledgment This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

References

[1] R. Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys*, 22(1), 1990.

[2] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.*, pages 463–488. Kluwer Academic Publishers, 2000.

[3] Business Internet Group. The black friday report on web application integrity. Report, BIG-SF, 2004.

[4] H. Chen, T. Tse, and T. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.

[5] S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*. ACM Press, 2006.

[6] L. Clarke and D. Rosemblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3), 2006.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003.

[9] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *proceedings of the first workshop on Self-healing systems*, pages 27–32. ACM Press, 2002.

[10] A. Hajnal and I. Forgacs. A precise demand-driven def-use chaining algorithm. In *proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2002.

[11] M. Harrold and M. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, 1994.

[12] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994.

[13] R. Heckel and M. Lohmann. Model-driven development of reactive informations systems: From graph transformation rules to JML contracts. *International Journal on Software Tools for Technology Transfer*, 9(2):193–207, 2006.

[14] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase. The java EE 5 tutorial. Technical report, Sun Microsystems, 2007.

[15] N. K. Jha and S. Gupta. *Testing of Digital Systems*, pages 560–679. Cambridge University Press, 2002.

[16] G. Leavens and Y. Cheon. Design by contract with JML. http://www.cs.iastate.edu/ leavens/JML/documentation.shtml, 2006. draft tutorial.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[18] L. Mariani and M. Pezzè. Dynamic detection of COTS component incompatibility. *IEEE Software*, 24(5), 2007.

[19] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.

[20] S. Modafferi, E. Mussi, and B. Pernici. SH-BPEL: a self-healing plug-in for ws-bpel engines. In *proceedings of the 1st workshop on Middleware for Service Oriented Computing*, pages 48–53. ACM Press, 2006.

[21] T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in java. In *proceedings of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 83–95. ACM Press, 2001.

[22] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[23] I. Singh, B. Stearns, M. Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, 2002.

[24] Sun. Petstore v1.4. http://java.sun.com/j2ee/1.4/.

[25] Sun Microsystems Inc. Java transaction service (JTS). http://java.sun.com/j2ee/transactions/, 1999.

[26] K. Vaidyanathan and K. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2), 2005.

[27] H. Wasserman and M. Blum. Software reliability via runtime result checking. *Journal of the ACM*, 44(6), 1997.

[28] S. Zachariadis, C. Mascolo, and W. Emmerich. The SATIN component system-a metamodel for engineering adaptable mobile systems. *IEEE Transactions on Software Engineering*, 32(11):910–927, November 2006.

[29] M. Zaidman. Teaching defensive programming in java. *Journal of Computing Sciences in Colleges*, 19(3):33–43, 2004.