Modeling Fault Tolerance Tactics with Reusable Aspects

Naif A. Mokhayesh Alzahrani , Dorina C. Petriu
Department of Systems and Computer Engineering
Carleton University
Canada K1S 5B6
nzahrani@sce.carleton.ca , petriu@sce.carleton.ca

ABSTRACT
This paper is part of a larger research project aiming to integrate dependability analysis in the early phases of the software development process, by generating and analyzing Stochastic Reward Net (SRN) models from UML software models. The paper is focused on adding fault tolerance to software designs by using Aspect-Oriented Modeling. More specifically, single-version fault tolerance tactics are modeled as generic reusable aspects annotated with dependability attributes. The paper describes how the generic aspects are instantiated, bound to the context and composed with the original UML software model. Since an SRN analysis model is generated from the UML model, the paper discusses what kind of transformation rules are necessary for translating fault tolerance tactics from UML to SRN, giving as an example the transformation rule for checkpoint synchronization. A case study illustrates the proposed approach.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability*; D.2.13 [**Software Engineering**]: Reusable Software – *reuse models*.

Keywords
Fault Tolerance, Dependability Analysis, Model Transformation, Aspect Modeling.

1. INTRODUCTION
The dependability of a system is defined as the ability to avoid failures that are more frequent and severe than is acceptable [3]. It encompasses a set of attributes such as availability and reliability. Performing early quantitative dependability assessment and reasoning based on annotated architectural and behavioral models will help modelers to take the right design decisions for meeting the requirements. Our ultimate objective is to integrate dependability analysis in the model-driven software development process, by deriving automatically analysis models (based on Petri Net or other formalism) from UML software architectural and behavioral models, and by using the analysis results to provide feedback to the developer about dependability improvements. In our previous work, we introduced an automated dependability analysis framework that considers erroneous behavior and failure propagation of component based systems, called Component Erroneous Behavioral Aspect Modeling (CeBAM) [1]; we also proposed a set of transformation rules [2] to automatically derive Stochastic Reward Net (SRN) analysis models from UML architecture and behavior models annotated with MARTE profile and the dependability profile (DAM) [5, 19]. This paper is continuing the work toward such an objective, with the focus on using Aspect-Oriented Modeling (AOM) for adding fault tolerance mechanisms to software designs. The paper presents the Single Version Fault Tolerance Aspect Modeling (SvFTAM) approach that capture architectural and behavioral models of single fault tolerance tactics as generic reusable aspects annotated with formal dependability attributes.

Fault avoidance and removal techniques help in designing and building systems that anticipate, recognize and correct faults at runtime. It is known that developing and deploying a software system free of faults is hard to achieve, even for the most experienced people using the best available tools [11, 14]. In order to prevent system failures during operation, fault tolerance techniques are introduced to tolerate erroneous states and recover the system by bringing it back to a correct state. According to [3], "fault tolerance means to avoid service failure in the presence of faults". Fault tolerance mechanisms applied to the software architecture and behavior help to improve the overall system dependability and to deal with unpredictable situations, ensuring that the system behavior remains acceptable during operation.

Software fault tolerance techniques can be categorized in three groups. The first is design diversity or multi-version techniques, where many replicas with different implementations are developed from the same specifications, but by different teams and different programming languages. Examples are the recovery block and N-version programming. The second group, data diversity, uses identical replication, but each replica will be executed with different data generated by data re-expression mechanisms from the original data. Example of this category is N-copy programming [14, 22]. Single-version fault tolerance (SV-FT) is the third category, based on redundant software modules that can detect faults and apply recovery actions, such as restarting or switching to a redundant spare module deployed on a different node. Exception handling, checkpoint, restart, and process pairs are examples of SV-FT techniques [23].

Increasing redundancy by identical replication is a common approach for fault tolerance in hardware. According to [14] this approach is not applicable to software, which is deterministic and thus each replica receives and processes the same data; instead, design diversity needs to be used. However, the work in [10, 24] introduces a new thinking in software fault tolerance based on environmental diversity as opposed to design diversity. Software bugs are classified into two categories: Bohrbugs and

QoSA'15, May 04-08, 2015, Montreal, QC, Canada

http://dx.doi.org/10.1145/2737182.2737189

Mandelbugs. The former is manifested consistently under known conditions and should be fixed during testing, while the later is hard to reproduce and it has complex error propagation. In [24] it is shown that the failures caused by Mandelbugs are more predominant. Restart, reconfigure and reboot are techniques employed to recover from Mandelbugs. In practice, availability tactics presented in [4] depend on redundancy or retry of a single version. Indeed, design diversity is not widely adopted in practice due to its high cost and effort, being used only for mission critical systems.

Adding any kind of fault tolerance mechanism is expected to improve the system's reliability and availability, but the effects are non-trivial due to the dependency of such mechanisms on the software context [6]. In fact, each fault tolerance technique needs to be customized and tailored to the application using it. Our proposed approach aims to provide quantitative data for supporting an easy comparison of different fault tolerance tactics, in order to select the best solution for a given system.

In this paper, we address the above issues by introducing the SvFTAM approach that models single version fault tolerance tactics as generic reusable aspects. Our work has been inspired by the new vision of software fault tolerance introduced in [4, 24]. We illustrate our approach with three reusable fault tolerance tactics: spare with checkpoint, standby spare, and retry. A generic aspect is instantiated and then its parameters are bound to the application context. The resulting context-specific aspect models are then composed with the original design model.

Applying fault tolerance to software architecture usually requires adding new components and modifying the existing ones, therefore the overall architecture changes and becomes harder to maintain. To overcome this issue, we replace a component without fault tolerance with a single composite component which contains the original component (possibly replicated) and a fault tolerance manager, preserves the original interfaces and has fault tolerance capabilities to recover from internal manifested failures and to prevent failure propagation (as described in section 3).

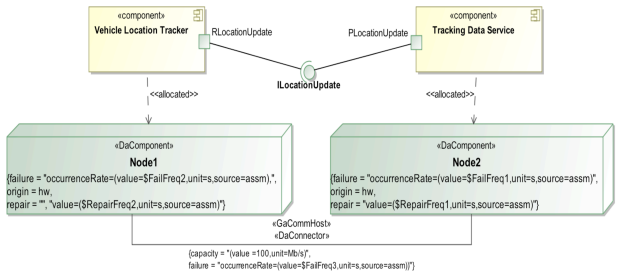
The main contribution of this paper is threefold. First, we introduce the SvFTAM approach that applies fault tolerance tactics to the UML software architecture and behavior using the AOM approach. In a simple case study, we show how to model SV-FT tactics and then present an automated process for aspect instantiation and composition with the basic UML model, which represents not only the normal behavior, but also the erroneous behavior and failure propagation following the CeBAM approach introduced in our previous work (see sections 3 and 4). Secondly, we discuss what new transformation rules are necessary to map the elements of a fault-tolerance tactic to SRN, in order to extend the original SRN analysis model with the respective fault-tolerance tactic, giving the checkpoint transformation as an example. Third, we illustrate how to approach the analysis by solving and comparing the derived SRN models before and after applying fault tolerance as explained in section five.

The paper is organized as follow. Section 2 presents the Vehicle Tracking System (VTS) case study modeled according to CeBAM. Section 3 describes SvFTAM by modeling three SV-FT tactics. Section 4 illustrates the process of aspect instantiation and composition, whose effect is to refactor the case study model by adding the spare checkpoint tactic to the original design. Section 5 discusses how to approach the analysis of the derived model and what transformation rules are necessary to map the elements of the fault-tolerance tactic to SRN, giving as example the checkpoint transformation. Section 6 discusses related works and section 7 concludes and summarizes the future work.

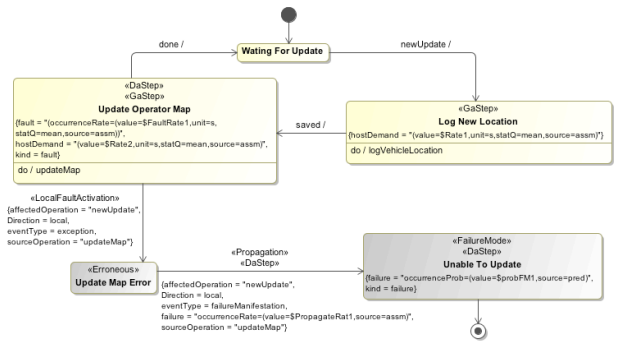
2. CASE STUDY SYSTEM

Vehicle Tracking System (VTS) is used as an example throughout this paper to illustrate the process of applying reusable single version fault tolerance tactics to a software architecture and behavior. The long-term objective is to automatically refactor the initial software design by adding different fault tolerance solutions, to generate the corresponding SRN analysis models and to compare quantitatively their effects on dependability attributes, such as availability and reliability.

The paper focuses on VTS systems used in vehicles to send periodic status update to the central monitoring system, providing vehicle location and other basic information. A system with similar functionality can be installed in taxi, police patrol cars and cargo trucks to provide the central control system with information about the vehicle such as current location. We model the VTS system according to the CeBAM approach introduced in our previous work [1], which considers failure propagation between components. We focus on periodically sending vehicle location update as the most critical scenario.



(a) Components involved in update location scenario



(b) Internal behavior of Tracking Data Service Component

Figure 1. VTS case study

Any failure in this scenario will affect the system availability and reliability. Figure 1(a) shows the components involved in the scenario, as well as their deployment. *Vehicle Location Tracker* is the active component that will periodically report the vehicle location to the *Tracking Data Service* component by calling the *newUpdate()* operation. Once *Tracking Data Service* component receives the location update, it will perform a set of actions depicted in Figure 1(b). For the sake of simplicity, we model only one failure mode (represented by the states *Update Map Error* and *Unable to update*). We annotate the model with MARTE+DAM profiles with information to be used in the derived SRN model, as

explained in the last section of this paper. Note that we do not show here the port behavior state machines as recommended in CeBAM due to limited space. However, applying SvFTAM will not modify the port behavior of the original components, that have already passed the conformance and compatibility verification phase [1, 2]. The VTS system reliability and availability is an important non-functional property. Therefore, to achieve high reliability and availability, we have to avoid any single point of failure (such as *Tracking Data Service* component) by adding a fault tolerance mechanism to the initial design, as described in the following section.

3. SINGLE VERSION FAULT TOLERANCE ASPECT MODELING

Single version fault tolerance tactics [4, 11] have been widely used as best practices to improve system reliability and availability. However, there is a lack of modeling approaches able to represent these tactics in a generic reusable form, along with dependability annotations, which can be used in early software design phases. One of the main challenges is to customize and tailor the generic models to the software context in order to get accurate results in terms of dependability improvements. In order to model SV-FT tactics as reusable models we use AOM [25] to describe the structure and behavior of the selected tactics. In SvFTAM the generic structure and behavior of fault tolerance tactics is captured according to the CeBAM approach [1]. In order to reuse these generic tactics, we employ model transformations to automate the instantiation of context-specific aspects and to compose them with the basic model.

In any fault tolerance mechanism, four basic actions are taken to tolerate an erroneous state. First, the error detection action determines the erroneous state. Next, during the processing phase, two actions focus on assessing the damage, identifying the cause of the error and restoring the system to its normal state. The last action uses the recovered state to continue the service and the normal operation [11, 14, 22]. All reusable single version fault tolerance tactics in our approach are modeled according to CeBAM [1] that captures normal and erroneous behavior. The detecting mechanism in our approach is started by the primary component, which sends a notification message (i.e., raising an exception), to the fault tolerance manager component. In such a case, a recovery action will be taken by switching the control to the redundant component, which then resumes the request.

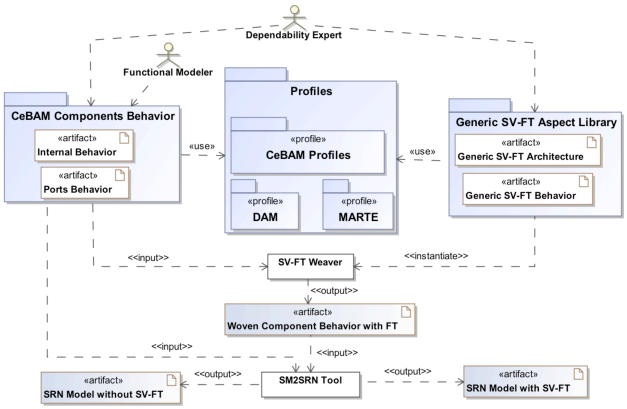


Figure 2. SvFTAM overall approach

We have created the *AspectComponent* UML2 profile to model fault tolerance tactics as generic reusable aspects annotated with dependability attributes. We consider single version fault tolerance tactics as crosscutting concerns, which can be applied to different components in the system. A point cut is a query that identifies the join point(s) in the base model where the aspects should be composed. To fully automate the process of selecting and composing aspects, we specify a point cut as an OCL query string in *AspectComponent* profile. During the composition process, the OCL query will be passed to an OCL parser implemented as a QVTO black-box module, for parsing and executing the OCL query [20]. As shown in Figure 2, the generic aspect template of a selected tactic will be instantiated and the template parameters (parameters name start with "|" symbol) will be bound to context values, thus obtaining an application-specific aspect that will be composed with the base model.

As illustrated in Figure 2, the original design (architecture and behavior) is built by the developer according to the user requirements. A dependability expert will augment the initial design in two phases. First, he/she will capture the erroneous behavior and will apply the required dependability attributes using the CeBAM, MARTE, and DAM profiles [1, 5, 19]. As shown in our previous work, erroneous behavior will be modeled as a context specific aspect, which is composed with the normal behavior using model transformation techniques. The next phase – which is the focus of this paper - is to refactor the software architecture and behavior by applying fault tolerance tactics to the most critical components. Different fault tolerance reusable aspects can be selected from a predefined library. A SV-FT weaver implemented in QVTO automates the process of obtaining the context-specific aspect and composing it with the basic model. The composed model (architecture and behavior) will be passed to another transformation chain (SM2SRN) to derive the SRN analysis model.

Adding fault tolerance will introduce more complexity in the software models. To mitigate this issue, SvFTAM will replace the most critical component in the scenario with a composite component, preserving the original interfaces; the new component embeds a set of components working together to provide fault tolerance capabilities. In the following examples, it contains two identical replicated components offering functional services, which are copies of the replaced simple component. These replicas are deployed on different nodes and their internal behavior is refactored to support fault tolerance actions (i.e., failure notification). In addition, the new component contains a fault tolerance manager component dedicated to managing fault tolerance behavior by detecting failure notifications and switching between replicas in case of hardware and software failure.

3.1 Spare with Checkpoint Tactic

This tactic, also known as *warm spare,* was originally described in [4]. It has an active component that periodically updates the state of the redundant spare using a checkpoint mechanism. The architecture changes include a new auxiliary component for detecting failure, as well as modifications to the component internal behavior to send checkpoint synchronization to the other replica. Our proposed SvFTAM approach will limit the changes to just one component and keep all the other dependent components unchanged. Figure 3(a) captures the structure of this tactic as a generic reusable aspect model.

As already mentioned, the aspect model is a single composite component including two replicas that provide the functional services specific to the application. The fault tolerance manager component manages fault the tolerance behavior, detecting

failures and switching requests between replicas. The primary replica will send a checkpoint update to the secondary replica. In case of a failure, the primary component will notify the fault tolerance manager about the manifested failure. As a recovery action, the fault tolerance manger will switch the control to the second component, which will resume the execution from the most recent updated checkpoint. The fault tolerance behavior is executed inside the composite component and does not affect the other components from the original software architecture. In modeling this tactic's structure, we use the *AspectComponent* profile that is a part of CeBAM profiles set, as well as the DAM profile for dependability annotations [5]. The main composite component stereotyped *PointCut* is used by the transformation tool to identify the critical component, as specified in the OCLQuery attribute. It is also stereotyped as *Refactor*, which will guide the transformation to replace the selected component in the base architecture model with a composite component. Note that the port(s) of the replaced component will not change and we just add delegation connections from the main port to each replica to pass the incoming or outgoing messages. Therefore, there are no changes in the other original component(s), which are unaware of the component replacement which is the effect of the aspect application. New activities and actions must be added to the original behavior of both replicated components to support fault tolerance. In order to automate refactoring the internal behavior of the replicated component we develop four generic aspects models as shown in Figure 3 (b, c, d, e).

First, the *standby refactor* aspect will be instantiated once to modify the behavior of both replicas to start initially in standby mode and remain in that mode until receiving a *setPrimary* message from the fault tolerance manager. This aspect model has two states stereotyped with *pointCut* that are used to identify the source and target states in the base model. As explained earlier, the string value of the *OCLQuery* attribute will be passed to the OCL parser, which returns the join point in the base model. A similar technique is used for the transition stereotyped by *Refactor* to either replace a model element or to modify its attributes. It starts by identifying the transition in the base model using *OCLQuery* and then modifies its destination to the new added state (i.e., *Standby*). In addition, any model element stereotyped with *Add* will be treated by the transformation as a new behavior that needs to be added to the base model. Figure 3(c) shows the second refactor aspect, dedicated to refactoring failure modes states by adding an entry operation, which is used to notify the fault tolerance manger about the failure. In consequence, the control is either switched to the spare, or failure propagation is allowed in the case of failure in the spare component.

The *Role* interface is implemented by each replica to provide a service invoked by the fault tolerance manager to set the primary component. The main difference between *setPrimary()* and *setPrimary(ChcekpointName)* operations is that the former will be called during the initialization of the component, while the later will be called after a failure to resume the service from the last updated checkpoint. In fact, the implementation of these interfaces depends on the fault tolerance manager behavior. For instance, in our case the first replica will be always the primary component, until a failure occurs, in which case the second replica will resume the service from the most recently updated checkpoint.

In this tactic, the primary component will keep the spare component updated by sending periodic state updates. Figure 3(a) shows the checkpoint interface implemented by the second replica, which is the checkpoint receiver. In order to capture this behavior, we need to refactor the internal behavior of both replicas, by adding a checkpoint state in similar join points. However, the first replica will act as checkpoint sender, while the second will act as receiver. Therefore, we develop two refactor checkpoint aspects. It is the responsibility of the dependability modeler to insure that the checkpoint states are added in similar join points using OCL queries. Figure 3(d) shows the sender refactor aspect, which has an entry action for sending a checkpoint synchronization request to its counterpart.

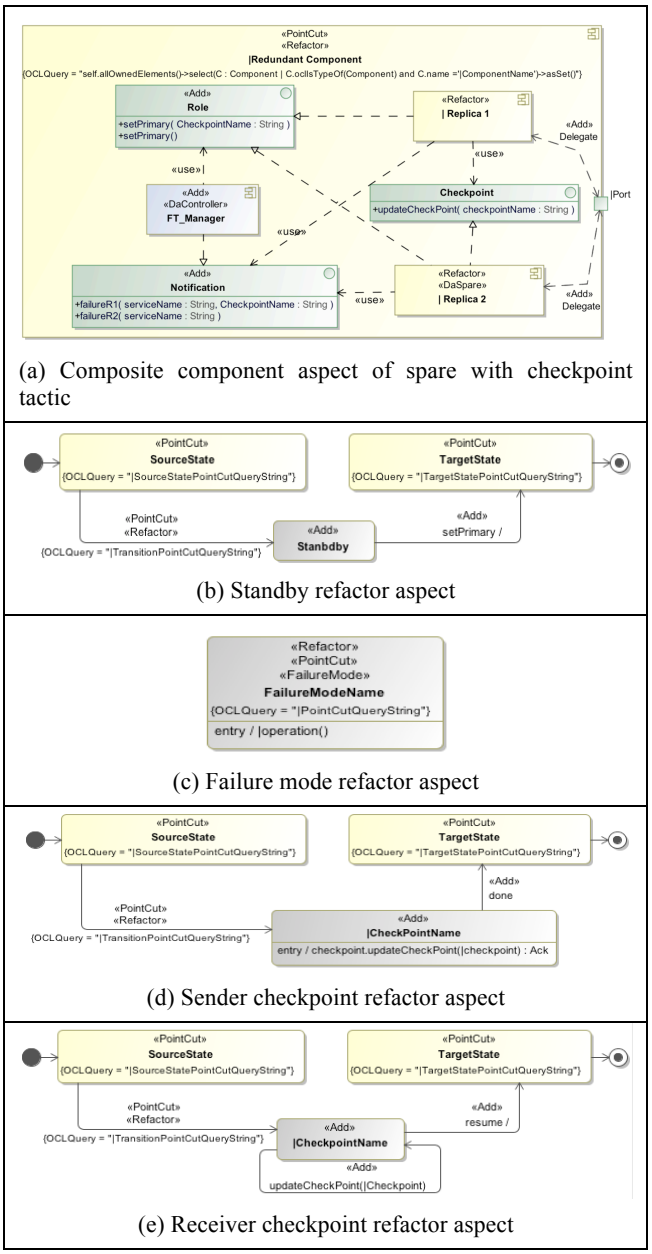


Figure 3. Spare with checkpoint tactic: structural and behavior aspects models

The call is synchronous, as the checkpoint sender waits for an acknowledgment from the receiver. A special transformation rule that maps the checkpoint behavior to SRN is presented in section 5.1.The fault tolerance manager has two main tasks: it acts as a failure detector receiving failure notifications from the replicas and it decides which component is the primary and when to switch the control to the standby component in case of

unrecovered failure. Both replicas start in the standby mode, until the fault tolerance manager sends the *setPrimary* message to the first replica. The primary replica keeps the standby replica updated by sending checkpoint updates, as already mentioned. Any incoming message will be passed to both replicas, but only the primary component will handle the request and the standby component will simply ignore it. A failure will propagate to other outside components only if the second replica fails, too. Software repair behavior is not considered in this tactic. However, the failed component due software failure will be repaired only if the hardware restarted. The fault tolerance manager behavior and the number of checkpoints can be customized according to the software context. For instance, it can detect the failure of the primary component due to hardware failure and then it switches to the second active redundant component.

3.2 Standby Spare Tactic

The structure of the standby spare tactics has two identical redundant components: one of them active and the second in standby mode. If the primary replica fails, then the fault tolerance manager will switch the control to the standby component to start handling the request form the beginning. Indeed, such behavior is suitable for systems with high reliability requirements [4]. The structure is modeled as a reusable aspect template by following similar concepts as in the previous tactic (see section 3.1), but without adding checkpoint interface between the two replicas. For the behavior we apply the same standby refactor aspect and failure mode aspect from Figure 3(b) and Figure 3(c), respectively. During fail over to standby component, the fault tolerance manager will call the *setPriamry(serviceName)* method implemented by the second replica to change its mode to primary and to start over the processing for the failed service from the beginning based on the service name passed as a parameter. A special transformation rules is implement to capture this behavior in the analysis model.

3.3 Retry and Restart Tactics

In principle, the previous tactics depends on replicating identical software components on different deployment nodes, without modifying the software behavior. It helps to avoid the effects of Mandelbugs by masking the failure and trying to process the request again on a different node hosting another instance of the same software component. However, the retry tactic depends on an error detection mechanism that identifies the erroneous state and retries the failed action before the failure mode manifested. This behavior can be modeled using CeBAM by including the retry action in the erroneous aspect model.

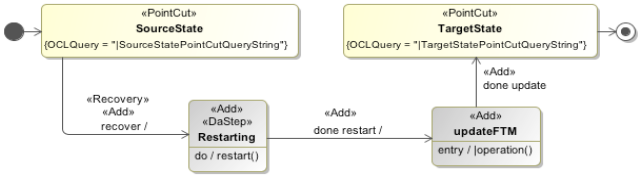


Figure 4. Restart aspect

The restarting tactic is a similar single version FT tactic that forces the software component to restart if any local failure mode is manifested. The restart behavior is modeled as a separate aspect that can be applied to the component internal behavior as shown in Figure 4. In this aspect the source state point cut will be the failure mode state while the target point cut state will be the initial state of the component internal behavior.

4. ASPECT COMPOSITION

The proposed process of using aspects for refactoring the software architectural and behavioral models in order to add fault tolerance capabilities is illustrated in Figure 5. As discussed in the previous sections, single version fault tolerance tactics can be modeled as generic reusable aspect models, which are then instantiated and their parameters bound to application specific values, producing context specific aspect models.

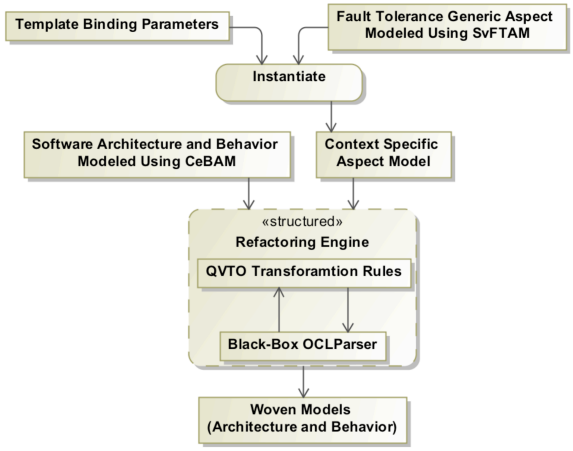


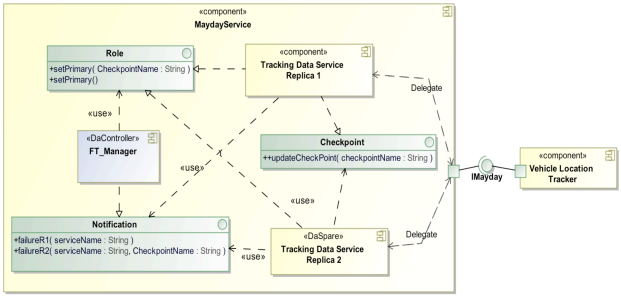
Figure 5. Refactoring Aspect to add fault tolerance

We choose to specify the point cuts as OCL queries, represented by string attribute of the stereotype *PointCut*. For instance, Figure 3(a) shows a parameterized OCL query expression as a string. Before the instantiation, the modeler will provide values for the template parameters to build up the complete query (e.g., the component name). During the aspect composition, the refactoring engine will invoke the OCLParser to parse and execute the OCL expression for finding the join points.

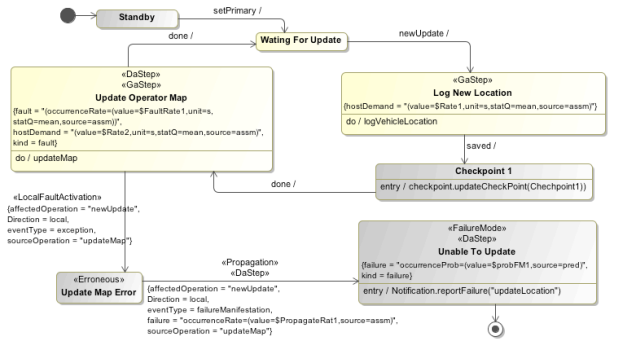
In the case of the VTS system we refactor the base model (architecture and behavior) by adding the fault tolerance tactic "spare with checkpoint". At the architecture level we just need to replace the *Tracking Data Service* component with a composite component that supports fault tolerance, keeping all the other dependent components unchanged. The modeler is expected to provide values for the template parameters such as component name and OCL query parameters. The refactoring engine will use the *AspectComponent* profile stereotypes as composition directives to guide the refactoring process. It starts by looking for *PointCut* stereotype and passes the *OCLQuery* string to the *OCLParser* black-box code to parse and execute the OCL expression for identifying the join points in the base model. In this case, the query should return at least one component (i.e., *Tracking Data Service*).

The next step in the transformation algorithm is to replace the join point element with a composite component (instantiated as a context specific aspect) which is stereotyped with *Refactor*. Note that this replacement will preserve the existing ports and connections with the other original components. An internal redundant component stereotyped with *Refactor* is in fact a copy of the replaced component in the original model. Every model element stereotyped with *Add* will be added as a new model element in the final woven model. Figure 6(a) shows the refactored architectural model obtained as a result.

The internal behavior of both replicated components is similar. It is refactored by three aspects. The first aspect will add a *standby* state after the initial *PseudoState* and a new transition triggered by the event *setPrimary* sent by the fault tolerance manager. The second aspect will refactor all failure mode states by adding an entry operation to notify the fault tolerance manager about the failure manifestation. The last aspect is the checkpoint aspect, which will be added in the same joint point to each replica, but with different behavior. *Tracking Data Service Replica1* acts as a checkpoint sender, while *Tracking Data Service Replica2* acts as a receiver. To ensure that the checkpoint is added in the same join point in each replicas we verify the result of *PointCut* OCL query during composition. Figure 6(b) shows the internal behavior of the contained primary redundant component. The behavior of the second redundant component will be identical to the primary component, except for the checkpoint state, which receives updates rather than sending them.



(a) Components involved in the update location scenario with fault tolerance tactic



(b) Internal behavior of Tracking Data Service Replica1

Figure 6. Applying spare with checkpoint tactic to VTS case study

5. DEPENDABILITY ANALYSIS

The dependability model of a system describes the failure and repair process of each component and also captures the failure propagation between software components, as well as failure propagation from a hardware node to the hosted software. In practice, software engineers model the normal behavior of the system and ignore the erroneous behavior due to its complexity. In our previous work [1, 2] we introduced a modeling approach that utilizes Aspect Modeling to separately model component erroneous behavior annotated with dependability information using the MARTE and DAM profiles [5, 19]. We extend this approach here to model fault tolerance patterns as generic aspects that can be applied automatically to any design. The purpose is to help developers to quantitatively compare different design alternatives, in order to estimate the improvement brought by different fault tolerance tactics in terms of reliability and availability.

We developed a tool based on QVTO and Acceleo [18, 20] that perform a set of model-to-model and model-to-text transformations to derive an SRN dependability analysis model. The proposed analysis approach is carried out through the following steps: 1) apply erroneous aspect to the normal behavior of each component; 2) iteratively derive SRN models for each component and compose them according to the system architecture; 3) verify conformance and compatibility between software components by analyzing the structure of the derived SRN; 4) extend the SRN model by adding deployment SRN subnet; 5) generate CSPL code from SRN model (where CSPL is the input language to the SPNP solver [8]). The SvFTAM patterns can be instantiated and applied to the original design after applying erroneous behavior aspects.

DAM profile allows software engineers to specify the output dependability measures of interest [5]. These measures are computed by solving the derived analysis model to get results that can be interpreted to improve the system design. We are interested here in unreliability and instantaneous availability of the system. According to our approach, the derived SRN model describes the healthy states along with erroneous behavior and failure propagation. Considering unreliability, i.e., the probability that the system has failed by time *t*, we just focus on failure mode states of the system. On the other hand, for the instantaneous availability we need to compute the probability that the system does not arrive to any failure mode by time *t*. Both of these measures are computed using transient analysis. For instance, if *Pi(t)* is the probability of the system being in state *i* at time *t*, then the unreliability is computed by summing the probabilities that the system is in any state *i* whose corresponding marking contains at least one token in a failure mode place [15].

One of the advantages of SRN is the ability to define a reward rate function for the system states of interest. To compute the unreliability of the system, we define a reward rate function as: *URi = if (#(P_failureMode_i) >= 1 || (#(P_failureMode_j) >= 1) 1 else 0*. This function includes all possible failure modes in the system. The unreliability is determined by performing first transient analysis to compute *Pi(t)* of each state and then compute the reward rate function given above. The reliability is given by *R(t)=1-UR(t)*. The instantaneous availability and mean time to failure measures are computed in a similar way, by defining a reward rate for each measure of interest. In SRN, rewards can be used in conjunction for both transient or steady state analysis.

5.1 SRN Derivation Rules

SRN is a variety of stochastic Petri nets that has some interesting features such as reward rates and marking dependency. In SRN any tangible marking can be associated with a reward rate. Moreover, the marking dependency is an essential characteristic of SRN that allows for defining model parameters as a function of the number of tokens in particular places [17]. Marking dependency was introduced for the convenience of the modeler, as it may simplify the model specification. For instance, arc multiplicity, transition guard and firing rate can be defined with marking-dependent feature to simplify the graphical model.

As mentioned in the introduction, the long-term objective of our research is to automatically derive dependability analysis models from annotated UML software models, in order to predict

dependability properties (such as reliability and availability) of the software architecture in the early development stages. In our previous work [2] we proposed a set of transformation rules for deriving SRN analysis model from an annotated UML software model without fault tolerance capabilities and without considering the fault assumption of the deployment nodes.

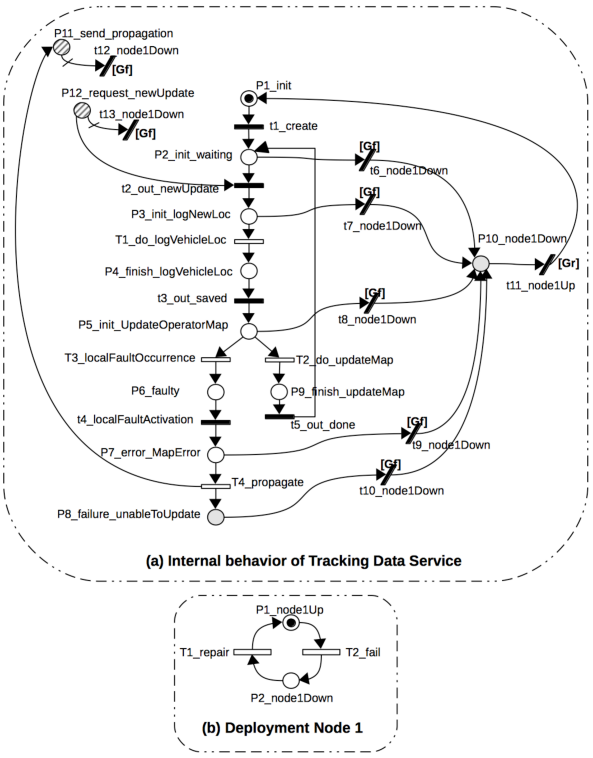


Figure 7. Derived SRN model for Tracking Data Service with deployment

Table 1. Hardware failure propagation SRN guards

Guard Name	Function
Gf	if (#(P2_node1Down) == 1) 1 else 0
Gr	if (#(P1_node1Up) == 1) 1 else 0

In this paper we propose an aspect-based approach for extending a software model with fault tolerance tactics. Since we intend to derive also the SRN model of such a system, we need new transformation rules for translating from UML to SRN the elements of the fault tolerance tactic added to the original software model. For instance, if we consider the "spare with checkpoint" tactic, we need transformation rules for the following elements of the tactic: setting the primary component, checkpoint synchronization, failure notification, and switching control to other replica. Among these transformation rules, the one for failure propagation from a deployment node to the hosted software behavior and checkpoint synchronization are the most complex rules, so we will briefly discuss both in this section.

The initial derived analysis model represents only the software Platform Independent Model (PIM) since it does not include the failure specification from the hardware nodes. The derived SRN model needs to be refactored to derive Platform Specific Model (PSM). We use UML component and deployment diagram to add a SRN subnet for each node as shown in Figure 7(b). Each software component is allocated to a hardware node, therefore a hardware failure propagates to the software, causing the loss of any request that is being processed or cashed. In such a case, the two SRN subnets must be synchronized in a way that the software SRN subnet model goes to the failure mode if the hardware node is down and it goes back to the initial state if the hardware node is repaired. In order to represent this synchronization without any additional complexity, we make use of a SRN feature that can simplify our model, namely guards to model failure propagation from the SRN hardware subnet to the allocated software subnet.

In Figure 7 we have two SRN subnets: subnet (a) is the derived SRN model of the *Tracking Data Service* component internal behavior and subnet (b) the failure and repair behavior model of the hardware node hosting this component. To model the failure propagation between hardware node and hosted software we add a set of guarded transitions as shown in Table 1. For instance, the *t6_node1Down* transition that has *Gf* guard is fired only if the hosted node SRN subnet is in failure mode (i.e. *P2_node1Down* has a token). During the normal operation the software the SRN subnet will be in one state at a time; if the hardware node goes down the guarded transition attached to that state will be enabled causing the loss of the request and switching to failure mode. Moreover, once the hardware node recovers, the guarded transition *t11_node1Up* is fired, allowing the software SRN subnet to be started again from the initial place to model software starting up after hardware recovery.

The striped places in subnet (a) will be used to compose the component internal behavior with its port protocol state machine. These places represent the event pool of the component internal state machine and it may contain many tokens that represent incoming and outgoing requests. Failure of the host node will trigger the flushing out transition of all pending requests. This is modeled by a guarded immediate transition with an input arc with marking dependent multiplicity, to be enabled once the host node is down.

As explained in section 3, a checkpoint may be added in different places of the main behavior of the replicas in order to synchronize their data state. In software, a checkpoint synchronization message is periodically sent from the primary replica to the spare replica along with the checkpoint name as a parameter. We need to map this semantic to the SRN model, to have a generic checkpoint mechanism without introducing extra complexity in the analysis model. Tokens in SRN models do not carry any information, so we cannot use them to carry parameters. However, we use SRN guards again for synchronizing transitions. Figure 8 shows a generic SRN model for checkpoint synchronization using a set of transition guards.

The transition *T_entry_updateCheckpoint1R1* of the sender component (Replica1) will send a checkpoint synchronization request by depositing a token in the shared place called *P_request_checkpoint*. We assume that the communication between replicas is immediate and never lost. In the software model the type of checkpoint message is synchronous, therefore the sender component will wait for an acknowledgment from the replicated component. To map this semantic, we add the guard *Gs1* to *T_Ack1R1* transition of the sender component that will prevent it from firing until a token is deposited in the *P_init_Checkpoint1R2* place of the receiver component. Another approach to model the acknowledgement could use a return path from Replica1 to Replica2, but this would add more places and transitions to the model.

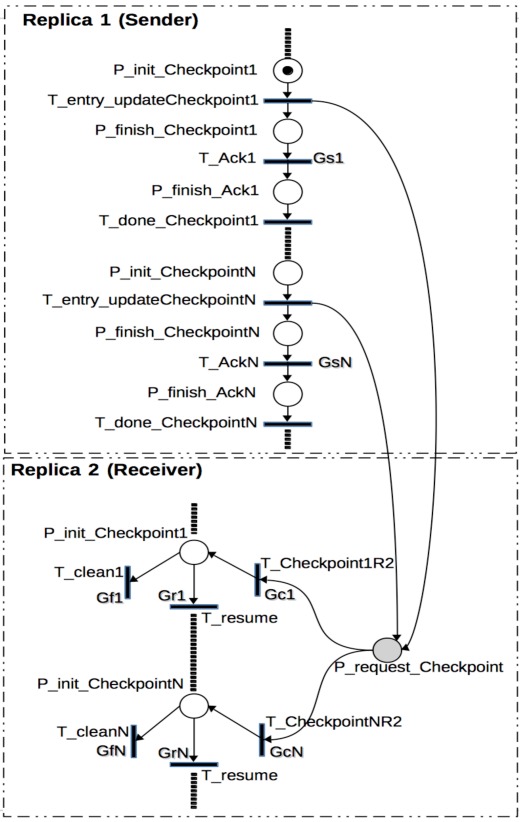


Figure 8. Checkpoint synchronization SRN model

Table 2. Checkpoint synchronization SRN guards

Guard Name	Function
Gs1	if (#(P_init_Checkpoint1) == 1) 1 else 0
Gc1	if (#(P_finish_Checkpoint1)== 1) 1 else 0
Gr1	if (#(P_inti_Primary) == 1) 1 else 0
Gf1	if (#(P_init_Checkpoint2) == 1 or .. #(P_init_CheckpointN) == 1) 1 else 0

In the receiver (Replica2), a token will be placed in *P_request_checkpoint* place that represents the receiving of a checkpoint update message, but this token does not carry information indicating which checkpoint place should take it. In order to deposit the received token in the correct checkpoint place that matches the place in the sender component, we add a $T\_checkpoint_iR2$ transition (where $i=1,N$) for each checkpoint place in the receiver component. Each transition has a guard that checks the marking of the corresponding checkpoint in the sender component, as shown in Figure 8 and Table 2. Now just one $T\_checkpoint_iR2$ transition will fire to receive the message. Transition T_clean_i (where $i=1,N$) and its associated guard is added for each checkpoint transition to flush out the token once a new checkpoint update was received from the sender component and for every new request processed by the primary replica.

In case of failure manifested in the primary replica, the fault tolerance manager will change the state of the second replica to primary and then a T_resume_i transition will be enabled to complete the execution of the request from the last updated checkpoint.

5.2 Setting the SRN Parameters

We use different UML diagrams to model the system. Component diagram(s) along with deployment diagram capture how the software components are composed and what are the provided and required services. Moreover, it shows the deployment of the software component instances on hardware nodes (see Figure 7). Behavioral state machine describe component internal behavior, while extended protocol state machine describe the provided and required services along with failure propagation. The component diagram will be used to guide the composition of the derived SRN model from each component state machine. MARTE and DAM profiles are used in our approach to augment the UML design with annotation dependability specifications that will be mapped to SRN parameters [5, 19]

Each deployment node is transformed to an SRN subnet that models the failure and repair of each node in the system, as shown in Figure 7(b). The annotations applied on this model are *DaComponent*, *DaConnector* and *GaCommHost*. *DaComponent* is applied to a hardware node to describe the aspect failure and repair. The transition rate of *T2_fail* is mapped to the *failure.occurenceRate*, and the rate of *T1_repair* to repair tagged-value. *DaConnector* and *GaCommHost* both describe connector specification: that the first captures the failure rate of the connector, while the second describes the connector capacity that is used to compute the transfer time between software components.

In our approach, the behavior model encompasses component normal and erroneous behavior. According to CeBAM, all the transitions are atomic transitions without any action. Figure 7 show the internal behavior state machine of *Tracking Data Service* component. This model is transformed to a SRN model as shown in Figure 7(a). The striped places will be used to compose the component internal behavior with the port's protocol state machine. In fact, all of these places together model the event pool of the state machine. *DaStep* and *GaStep* are applied on state activities such as *entry, do, exit* to annotate them with the processing demand, as well as with fault activation occurrences rate if an error propagation chain is attached to the state. For example, "*Update Operator Map*" state has *do/ updateMap* activity and an output transition to the error propagation state starting with "*Update Map Error*" state until the failure mode is manifested. In this state we use *hostDemand* attribute of *GaStep* stereotype to specify the processing time of the *updateMap* activity. This activity will be transformed into a timed transition called *T2_do_updateMap* as shown in Figure 7(a), whose rate is mapped to the value of *hostDemand*. Additionally, *DaStep* stereotype is applied to the same activity (*updateMap*) to annotate the fault activation rate. According to the transformation rules presented in [2] the fault activation of an activity is transformed to a time transition. In VTS case study, the timed transition *T3_localFaultOccurrence* represents the fault activation occurrence rate of *updateMap* activity and its rate is mapped to *occurrenceRate.* In some cases, a state is annotated with *GaStep* only, since the error propagation is not modeled (e.g., "*Log New Location*" state).

DaStep is applied to two other model elements of the component internal behavior state machine: a) to the propagation transition specifying the time between the switching to erroneous state and the failure manifestation (translated to the SRN timed transition *T4_propagate*); and b) to the failure mode state specifying the occurrence probability of failure model state).

5.3 Analysis of Results

We used the SPNP tool for solving the derived SRN model [8]. For the case study of this paper we use the analytical solver to compute the unreliability using transient analysis. The values assigned to the input parameters of the *Tracking Data Service* component and its deployment node is shown in Table 3. Similar parameters are assigned to each similar component and connectors. All timed transitions have exponential distributions.

The generated SRN model is based on the following assumptions:

- Components fault occurrences are independent;
- Failure modes of each component are mutually exclusive, therefore, if the component fails with a given failure mode then it is considered failed and no other failure mode may occur until it is repaired;
- Once a local fault occurs with a given rate, it switches to the erroneous state immediately;
- The fault tolerance manager is deployed on a node that has negligible failure rate;
- Communication between the internal components of the composite component is immediate with no delay;
- Network failure is recoverable.

In this example we focus on unreliability (failure probability) analysis, trying to answer two questions: First, what impact have the different SvFTAM tactics on the system unreliability . Second, what is the best SvFTAM tactics to be applied for the particular system in terms of reliability improvement. As explained in the previous sections, the analysis model is automatically derived before and after applying fault tolerance tactics and then solved to get quantitative data for comparison. Figure 9 shows the unreliability (failure probability) results of VTS case study before and after applying SvFTAM tactics. It is clear that the system failure probability is lower after applying any SvFTAM tactics. This answers the first question and the quantitative results will encourage the designers to consider such tactics for improving the reliability of the system.

In addition, the collected results will guide the developers in the comparison of different SvFTAM tactics and help them select the best one in terms of reliability improvement. For instance, for the values chosen for the failure and repair rates, the retry tactic is the best option for the VTS case study. For different failure and repair rates, the comparison results may be different.

Table 3. Parameters of *TrackingDataService* component

SRN Transition	DAM Parameter	Assumed Rate
T1_do_logVehicleLoc	$Rate1	1/(15 s)
T2_do_updateMap	$Rate2	1/(40 s)
T3_locaFaultActivation	$FaultRate1	1/(2700 s)
T4_propagate	$PropagateRate1	1/(10 s)
T1_repair	$RepairFreq1	1/(300 s)
T2_fail	$FailFreq1	1/(604800 s)

Checkpoint tactic shows a slight improvement compared to the standby spare tactic. The difference is not very significant in this particular case study due to the fact that the size and simplicity of VTS case study that has only single checkpoint synchronization between the two redundant components. According to these values the noticeable improvement will be much clear in a bigger system that has multiple checkpoints. Standby spare tactic is ranked low compared to other tactics. In this tactic after switching to the second redundant component, the failed request starts from the beginning and fault may again reappear in any operation. On the other hand, retry and checkpoint tactics continue from the last failed operation.

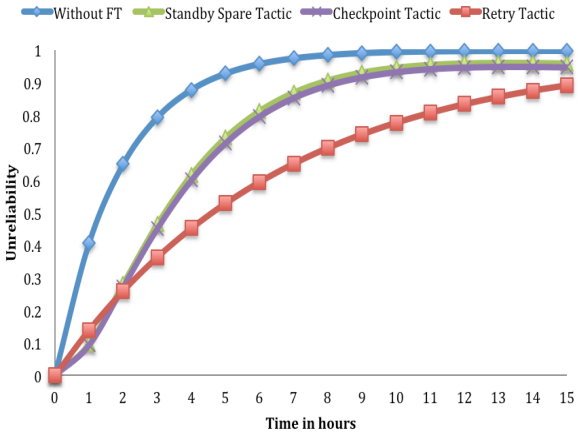


Figure 9. Unreliability of VTS case study with SvFTAM tactics comparison as function of time

6. RELATED WORKS

Software fault tolerance was addressed heavily in the literature from different perspectives. For example, in [22] are explained fault tolerance techniques and implementations, while in [11] are presented over sixty software fault tolerance patterns that cover all fault tolerance phases and were modeled as UML profiles in [21]. Availability tactics using redundancy were introduced in [4]. In [12] it is studied the effect of applying fault tolerance tactics to software architecture patterns, but the approach was qualitative, based on interviews with expert developers to evaluate how much change is needed in order to incorporate fault tolerance to the set of existing software design. The work in [24] shows how the recovery process is implemented in real IT systems for different kinds of Mandelbugs. The effects of applying software fault tolerance mechanisms to Palladio Component model was presented in [6]. A validation approach to achieve fault tolerance requirements in component based system is proposed in [7] .

Using AOM for refactoring the software architecture by adding fault tolerance techniques has been suggested by several authors. For instance, in [16] is presented an approach for modeling and integrating AOM into CBD. The paper also illustrates how AOM can be used to model component's dependability aspects separately, by modeling a template for fault tolerance that provides error detection and recovery services. This template can be customized and composed with base models through a weaving process. However, this approach does not consider internal behavior to derive an analysis model, as we have proposed. A similar approach presented in [9] uses AOM to construct and build fault tolerance systems. New notations are introduced to capture dependability aspects. The authors created a library of fault tolerance mechanisms along with its dependability analysis model template that is not derived from the software model. A model weaver is designed to integrate the fault tolerance aspect with the base software model, as well as to perform the model analysis, which is done separately. Role-Based Metamodeling language was employed in [13] to generalize modeling architectural tactics, i.e., performance and availability that can be reused to refactor existing designs.

7. CONCLUSION AND FUTURE WORK

The work presented here is part of a larger research project aiming to integrate dependability analysis in the early phases of software development process, by generating and analyzing Stochastic Reward Net (SRN) models from UML software models. The paper is based on our previous work: a) an framework for modeling normal and erroneous behavior and failure propagation of component based systems (CeBAM) [1]; and b) a set of transformation rules [2] to automatically derive SRN analysis models from UML architecture and behavior models.

In this paper, we present the Single-Version Fault Tolerance Aspect Modeling (SvFTAM) approach, which is an aspect-based technique for modeling structural and behavioral fault tolerance tactics as reusable models. The following examples of fault tolerance tactics have been modeled using SvFTAM: spare with checkpoint, standby spare, retry and restart. We show how to model in UML the "spare with checkpoint" tactic and illustrate with the help of a case study how to use it for adding fault tolerance capabilities to the original design.

Since the automatic derivation of the SRN model of a system with fault tolerance from the software model is part of the proposed approach, new transformation rules are needed for translating from UML to SRN the fault tolerance tactics added to the original software model. We discuss how to map the semantics of the checkpoint mechanism from UML to SRN. Moreover, we solve the derived model to study the effect of applying fault tolerance to the VTS case study and compare different SvFTAM tactics in terms of reliability. Currently, we are working on completing the QVT transformation that implements the UML to SRN translation proposed in the paper.

Future work will address the problem of solving the SRN model for reasonable large systems. State space grows exponentially with the model size. To avoid that we are planning on using software architecture decomposition to guide the solution of the corresponding SRN model by Petri Nets decomposition.

8. ACKNOWLEDGMENTS

Authors acknowledge the support provided by the Albaha University and Ministry of Education, KSA. This research was partially supported by NSERC, Canada.

9. REFERENCES

[1] Alzahrani, N. and Petriu, D.C. 2013. Modeling Component Erroneous Behavior and Error Propagation for Dependability Analysis. *16th International SDL Forum on Model-Driven Dependability Engineering*. LNCS, vol.7916, 124-143. Springer (2013).

[2] Alzahrani, N. and Petriu, D.C. 2013. Derivation of Stochastic Reward Net for Compatibility and Conformance Verification of Component Erroneous Behavior Model. *Proceedings* of *IEEE 19th Pacific Rim International Symposium on Dependable Computing - PRDC2013*, 142-151.

[3] Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 11–33 (2004).

[4] Bass, L., Clements, P. and Kazman, R. 2012. Software Architecture in Practice. Addison-Wesley.

[5] Bernardi, S., Merseguer, J. and Petriu, D.C. 2011. A dependability profile within MARTE. *Software and Systems Modeling*. (2011).

[6] Brosch, F., Buhnova, B. and Koziolek, H. 2011. Reliability prediction for fault-tolerant software architectures. *Proceedings of the Federated Events on Component-Based Software Engineering and Software Architecture* - QoSA+ISARCS'11, 75–84 (2011).

[7] Bucchiarone, A., Muccini, H. and Pelliccione, P. 2007. Architecting Fault-tolerant Component-based Systems: from requirements to testing. *Electronic Notes in Theoretical Computer Science*. 168, 77–90 (2007)..

[8] Ciardo, G., Muppala, J. and Trivedi, T. 1989. SPNP: stochastic Petri net package. *Petri Nets and Performance*.

[9] Domokos, P. and Majzik, I. 2005. Design and analysis of fault tolerant architectures by model weaving. *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*. (2005).

[10] Grottke, M. and Trivedi, K.S. 2007. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *Computer*. 40, 2, 107–109 (Feb. 2007).

[11] Hanmer, R. 2007. Patterns for Fault Tolerant Software. John Wiley & Sons.

[12] Harrison, N.B. and Avgeriou, P. 2008. Incorporating fault tolerance tactics in software architecture patterns. *RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems-SERENE'08*, 9-18 (2008).

[13] Kim, S., Kim, D.-K., Lu, L. and Park, S. 2009. Quality-driven architecture development using architectural tactics. *Journal of Systems and Software*. 82, 8 (Aug. 2009).

[14] Knight, J. 2012. Fundamentals of Dependable Computing for Software Engineers. Chapman and Hall CRC Press.

[15] Lyu, M.R. 1994. Software Fault Tolerance. John Wiley & Sons Inc.

[16] Michotte, L., France, R.B., Fleurey,F. 2007. Modeling and Integrating Aspects into Component Architectures. *11th IEEE International Enterprise Distributed Object Computing Conference*, 181-190 (2007)

[17] Muppala, J., Ciardo, G. and Trivedi, K.S. 1994. Stochastic reward nets for reliability prediction. *Communications in reliability, maintainability and serviceability*., 9–20 (1994).

[18] Object Management Group: MOF Model to Text Transformation Language, v1.0. (Feb. 2008).

[19] Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. (Jun. 2011).

[20] Object Management Group: Query View Transformation (QVT) v1.1 formal/2011- 01-01. (Jan. 2011).

[21] Ongsiriporn, O. and Senivongse, T. 2013. UML profile for fault tolerance patterns for service-based systems. *10th International Joint Conference on Computer Science and Software Engineering*, 240-245 (2013).

[22] Pullum, L.L. 2001. Software Fault Tolerance Techniques and Implementation. Artech House Publishers.

[23] Torres-Pomales, W. 2000. Software Fault Tolerance: A Tutorial, NASA.

[24] Trivedi, K.S.,Mansharamani, R., Kim, D.S., Grottke, M. and Nambiar, M. 2011. Recovery from Failures Due to Mandelbugs in IT Systems. *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing - PRDC2011*, 224–233 (2011).

[25] Yedduladoddi, R. 2009. *Aspect Oriented Software Development:* An Approach to Composing UML Design Models. VDM Publishing.