Derivation of Stochastic Reward Net for Compatability and Conformance Verification of Component Erronous Behaviour Model

Naif A. Mokhayesh Alzahrani , Dorina C. Petriu
Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, Ottawa, ON, Canada, K1S 5B6
{ nzahrani | petriu }@sce.carleton.ca

***Abstract*— The compatibility verification between interacting components and the conformance verification of their internal behavior with the corresponding ports protocol behavior are crucial steps for the early identification of unexpected messages between components. The behavior models used for verification include erroneous behavior along with normal behavior, in order to ensure greater accuracy in reliability and availability analysis. We use our Component Erroneous Behavior Aspect Modeling (CeBAM) approach introduced in previous work, which applies aspect-oriented modeling for adding erroneous behavior to UML state machines representing normal behavior. In this paper we present transformation rules for deriving Stochastic Reward Net (SRN) from CeBAM representations. The first step is to generate SRN for individual component behavior in order to check the conformance between component internal behavior and their ports protocol behavior. Subsequently, we compose the generated SRNs models of the connected components to verify their compatibility. We show how to identify conformance and compatibility issues during the construction and composition of components SRN model by analyzing SRN properties (e.g., deadlocks). We illustrate the proposed verification approach through a case study modeled according to CeBAM.**

Keywords- conformance; compatibility; failure propagation

I. INTRODUCTION

Software architecture provides a set of high-level abstractions for representing structure, behavior and Non-Functional Properties (NFP) of the software. According to [1], component structure, dynamic behavior and allocation are the categories of architectural constructs. Component Based Development (CBD) applies the "divide and conquer" principle to manage system complexity. It produces an abstract model that shows the static structure of the system in which each component is assigned a specific functionality. In addition, each component is a unit of composition that interacts with other components through predefined interfaces.

Using the software architecture as a basis for the early reasoning and evaluation of the system's NFP helps to reduce the cost and produce software conforming to specifications [2]. Software dependability is among one of the examples of NFP that need to be evaluated during the design phase. This in turn encompasses a set of attributes: reliability, availability, maintainability, integrity and safety [3]. The quantitative results of these analyses will support the developer in making the right decisions for building dependable systems. Although different approaches were proposed in the literature, in order to address the reliability and availability modeling [4], [5], many existing approaches do not adequately consider error propagation in predicting system reliability [6].

Model Driven Development (MDD) is a promising approach for software development that changes the focus from code to models. This focus on models facilitates the analysis of different NFP by using formal analysis models obtained by model transformations from the software models [7] [8]. In fact, combining MDD and CBD is an appealing approach for software development, as it reduces the complexity, time, cost and it helps to integrate NFP analysis during the design phase.

In our previous work [7], we presented a framework for automating dependability analysis, which considers error modeling and failure propagation of a component-based system. A component's internal behavior and its ports' behavior are modeled using the Component Erroneous Behavioral Aspect Modeling (CeBAM) approach. In CeBAM we model the component's erroneous behavior as an aspect model that is automatically composed with the component's normal behavior. Modeling the erroneous behavior of components and verifying the failure propagation is a crucial part in our approach. We believe that a proper representation of error and failure propagation in CBD has an impact on the accuracy of reliability (availability) predictions for the system. Moreover, it helps developers to take the right decisions based on quantitative data such as selecting proper fault tolerance mechanism, placing error detection, and using suitable recovery approaches.

In this paper, the focus is centered towards verifying the components' conformance and compatibility. A component's ports behavior described by an extended protocol state machine (PSM) must conform to its internal behavior modeled using a behavioral state machine (BSM). Thus, the goal of conformance verification is to avoid any unexpected messages between the component internal behavior and its ports, while component compatibility verification is to avoid mismatch between connected components in terms of provided services or failure propagation. A mismatch created when internal component failure is raised and not captured in corresponding component port or when component failure can not be propagated and handled by communicated components.

The proposed verification process is done by transforming the components' BSM and their PSMs to Stochastic Reward Net (SRN). Initially we begin by transforming each BSM and PSM model separately to SRN, and then for each component we compose the obtained SRN of component internal behavior with its ports' SRN for conformance verification. Next, we consider component compatibility by composing the SRNs of the connected ports. The conformance and compatibility violations and mismatches are then determined during the composition phase, and also by analyzing the generated SRN to identify deadlocks. Although model checker techniques can be used in such verification process, but dependability analysis model (SRN) is utilized in our approach to avoid having different models with different properties.

This paper is organized as follows. Section two presents the background of the CeBAM approach. Section three explains the case study and how we apply CeBAM. The next section describes the proposed approach for component conformance and compatibility verification. Then we illustrate the derivation and composition of SRN, as well as the semantics behind each transformation rule. Section six presents examples of our proposed approach applied to the case study. Related works is presented in section seven, and then we conclude and summarize our work in progress.

II. BACKGROUND

In previous work [7], we presented our long-term objective that is to develop a framework based on standard modeling languages (such as UML and QVT), which would help developers to evaluate the dependability properties during a CBD + MDD process, by taking into consideration component erroneous behavior and error propagation. We believe that including component erroneous behavior in dependability analysis and prediction will help developers to make the right design decisions. We proposed the CeBAM approach, which applies aspect-oriented modeling techniques in order to model erroneous behaviors separately from the normal behavior. This approach reduces the model complexity and improves its readability and modifiability.

Modeling component behavior using CeBAM can be done in two phases [7]. In the first phase we only model the normal behavior of both component views (internal and external). BSM is used for the component's internal normal behavior and extended PSM is used for the external view. The second phase is focused on modeling component erroneous behavior separately using two profiles: ErroneousBehavior and AspectBSM profile. The outcomes of this phase are represented by two aspect models: one for the erroneous behavior of the internal view and another for the external view. We may need a few iterations to build these two models. First we capture the local failures and then in the next iteration(s) we may have to add propagated failures that originate in other components. The iterations will end when all errors/failures have been "propagated". In some cases we may need to use refactoring aspects to preserve the run-to-completion semantics of BSM transitions. Fig. 2 and Fig. 5a show the final BSM and PSM respectively for the case study described in the next section. Also, in [9] we present all CeBAM models for the case study.

III. CASE STUDY

Factory Automation System (FAS) is the case study that will be used throughout this paper to illustrate the transformation of state machine (behavioral and protocol) to SRN. This case study was presented in [10], and our objective is to verify the compatibility between the FAS components and the conformance of components ports with their internal behavior. It is important to mention that this step is required before starting dependability analysis (as explained in [7]).

The FAS is an example of a distributed real-time system. It consists of three components: Automated Guided Vehicle (AGV), Supervisory System and Display System. Each of these components represents a subsystem in FAS and they interact with each other through predefined interfaces. Fig. 1 shows the component's architectural model. In this figure, AGV is the main component in FAS and it consists of subcomponents such as motor, arm, sensor and timer. Here the Supervisory System sends the command (load/unload) to the AGV and the status is reported to the display component.

COMET methodology [10] was used to develop this case study. We extend the development of this case study by applying the CeBAM approach [7] to model component erroneous behavior and error propagation. Due to the limited space we select only "move to the station" use case to illustrate our transformation approach; more details can be found in [9].

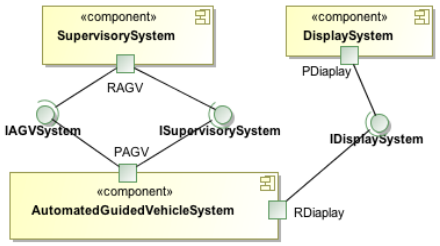


Figure 1. Component structure of case study [10]

Since the AGV component is the core component of the FAS, we choose to model its internal behavior. Initially we model the internal behavior and then we apply the refactoring aspect according to the CeBAM approach. For each transition that has an action we create an intermediate state with a *do* activity corresponding to the action that was on the transition. Fig. 2 shows the final internal behavioral state machine after applying the refactoring aspect and erroneous behavior aspect ([9] has all aspect models). Due to limited space, as well as for the sake of simplicity, we model just one fault that is raised from the *chcekDestination* method. If this fault is activated, it will change the behavior of the AGV from normal behavior to the erroneous behavior and since this error is not recoverable it will propagate to cause a failure mode of the AGV. Moreover, this failure mode will be propagated to the Supervisory System

component throughout the provided interface and it will cause another error and failure mode of the Supervisory System component according to [3].

In order to study the error propagation from component BSM to its PSM and then to the connected component we have to model the port behavior of AGV and the Supervisory System components according to CeBAM. Fig. 5a shows *PAGV* port behavior model. Hence, we show the final model after applying erroneous behavior aspect according to the CeBAM approach [7]. The report [9] has the complete model list used during iterations of the CeBAM modeling approach, as well as other component ports models.

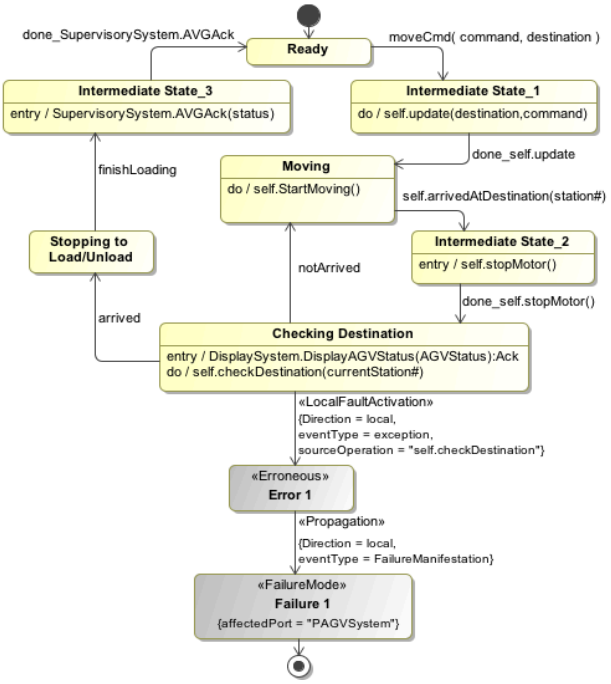


Figure 2. Automated guided vehicle component internal behavior

In the following section we will focus on illustrating the transformation of these models to SRN in order to verify the conformance and compatibility of the FAS system. Consequently, we will not show the modeling and transformation of the DAM profile, which will be part of our dependability analysis framework as explained in [7].

IV. Conformance and Compatibility Verification Approach

In the CBD paradigm, the component is a logical unit of abstraction with well-defined interfaces that provide or require services from their environment [1]. In our approach we assume that the port is the interaction point and that it has the needed interfaces to be integrated with other components in the system. Interfaces consist of a set of public services that may be called by other components in the system through their required interfaces. This call may be synchronous so as to block the caller until it passes back the response, while the asynchronous call starts a new thread. In all connection styles the received call will be based on the internal component's behavior since the interface has no implementation.

However, in order to study and analyze the reliability (availability) we need to verify the conformance and compatibility of the involved components. In conformance verification we verify whether the component's internal behavior state machine is conforming to its ports' protocol state machine or not. In other word, we need to check whether or not each modeled incoming message on the component port has a corresponding transition on the internal component's behavior that can handle the message. It should also be verified whether or not each outgoing event modeled in the internal component behavior has been captured on the port behavior model or not. These events may include normal behavior events or erroneous and propagation behavior.

Compatibility deals with the communication between components in the system. For two connected components, the provided services must be compatible with the requester. In fact, using extended PSM as proposed in CeBAM helps us to model how components in the system interact with one another, by capturing the passing messages through its ports in a specific order.

UML [11] considers the conformance of protocol state machine through the *ProtocolConformance* model element. It explicitly assures that a specific state machine is conforming to a generic one. Thus, component realization must be conforming to its interfaces. Unfortunately, the conformance definition in the UML standard is limited and there is no clear framework for reasoning and verification. As a result, there are many approaches in the literature [12], [13] that attempt to address the problem of how to verify the components conformance and compatibility. In our dependability predication framework we are following state-based analysis methods based on SRN. In other words, we are using the same formal model for conformance and compatibility verification, since this, to us seems like a logical step before starting the dependability analysis using the same SRN model.

The verification phase should resolve all issues in the software model before starting the reliability/availability analysis phase. For instance, we need to be certain that any manifested internal failure inside a component will be propagated to all connected components and that the model is free of deadlocks (this is to avoid the impact of such issues on the reliability analysis results).

We identify the conformance or compatibility issues during the construction and composition of the SRN models obtained by transformation from the software model or that have identified deadlocks of the composed model using the SPNP tool. The activity diagram of Fig. 3 describes the verification process.

Iteratively for each component in the model, we apply transformation rules on its BSM and PSM models, which describe its internal and ports behaviors, respectively. Our next step is to compose the transformed models of the component's internal behavior with its ports. This composition includes normal behavior as well as the

erroneous behavior that connects incoming and outgoing messages from the components ports to its internal behavioral model. During this step, we may identify some conformance issues that must be fixed in the main software model before continuing to the next step. For example, in the PSM model we have incoming messages representing the external failure propagated from another connected component, even though the effects of this message is not modeled in the component internal behavioral model. Consequently, the component internal behavior model is not conforming to its port and this mismatch must be repaired in order to reflect the external propagated failure into the internal behavior model.

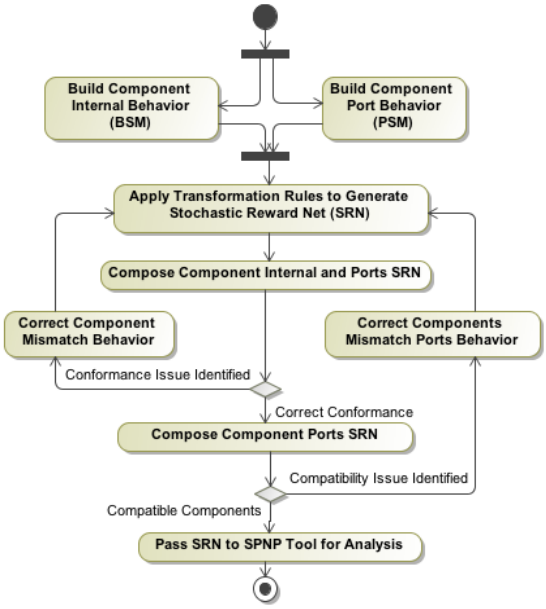


Figure 3. Conformance and compatibility verification activities

Once each of the components' internal behavior conforms to its ports behaviors, we may start the next step, which deals with the inter-component compatibility. For the connected components, we compose their port behavior model (SRN model) to identify incompatibility between components for the provided and required services and failure propagation. For instance, if there is a service failure modeled at one component's port, it may not necessarily be modeled at the connected component. This type of incompatibility must be resolved in the main model by verifying the conformance again in that component, before completing the compatibility verification (Fig. 3).

V. Deriving SRN from Components Behavioral Model

State machine is a behavioral diagram that may be used to specify the behavior of a part of a designed system, and it is characterized by states and transitions. State represents a situation of the component or object when some invariant condition holds. For instance, a specific state may represent a dynamic condition when a software component is performing a series of internal computations activities resulting from the external call event, or it may represent a static situation when the component is waiting for another event to occur [11].

UML2 defines three kinds of states: simple state, composite state, and submachine state. In addition, UML2 defines ten different kinds of pseudostates. In our case we focus on the initial and final pseudostates, as well as the simple state machine that does not have any sub-states or regions. Therefore, the limitation of our approach is that the hierarchical states and history are not supported. States can have optionally *entry/exit* actions and *do* activity. *Entry* action is executed when entering the state while the *exit* actions is executed when leaving the state. A *do* behavior is executed after an entry behavior and it continues as long as the state is active. Both actions (*entry/exit*) have "run-to-completion" semantics which means that those actions are uninterruptable and once they have started no other new events may be executed. On the other hand, *do* activity does not have "run-to-completion" semantics. In our approach we respect and preserve this semantics and we do not consider fault activation and failure propagation as new events, since it will be a part of the execution path of state actions. For instance, an entry activity that contains a method execution will not be interrupted by another call until it has finished its execution. However, during the execution, a fault may be activated to change the component's state from a normal to an erroneous state. Fault activation in this case is not a new incoming event dispatched from an event pool of that state machine, but it is changing the execution path of the activity that we precisely model in our modeling approach (CeBAM).

According to [11] state machines have three kinds of transitions: local transitions, internal transitions, and external transitions. Local transitions are used in composite state. It can not only leave any state but it may enter a new state (i.e. entering a new sub-state). Internal transitions will not cause a state change since the source and target states are the same. External transitions represent the transition between states (i.e. simple states). They have an *event[condition]/action* label that captures how the software component in a specific state receives the dispatched event and then evaluates the condition to start the action, if the condition is right. The transition action has the "run-to-completion" semantics, which means that the software component will not enter the target state until the action is successfully completed and no other event may be accepted during execution of the action associated with that transition. A special kind of external transition is the completion transition (done) that does not have an event or action and it is triggered when the activity of the state has finished execution.

As mentioned before, we use the behavioral state machine to describe the internal behavior of each component in the system. In order to capture the erroneous behavior of the activity on state transition without violating the "run-to-completion" semantics, we introduce the refactoring aspects[7]. This will add new intermediate states and will move the transition action activity to be a *do*

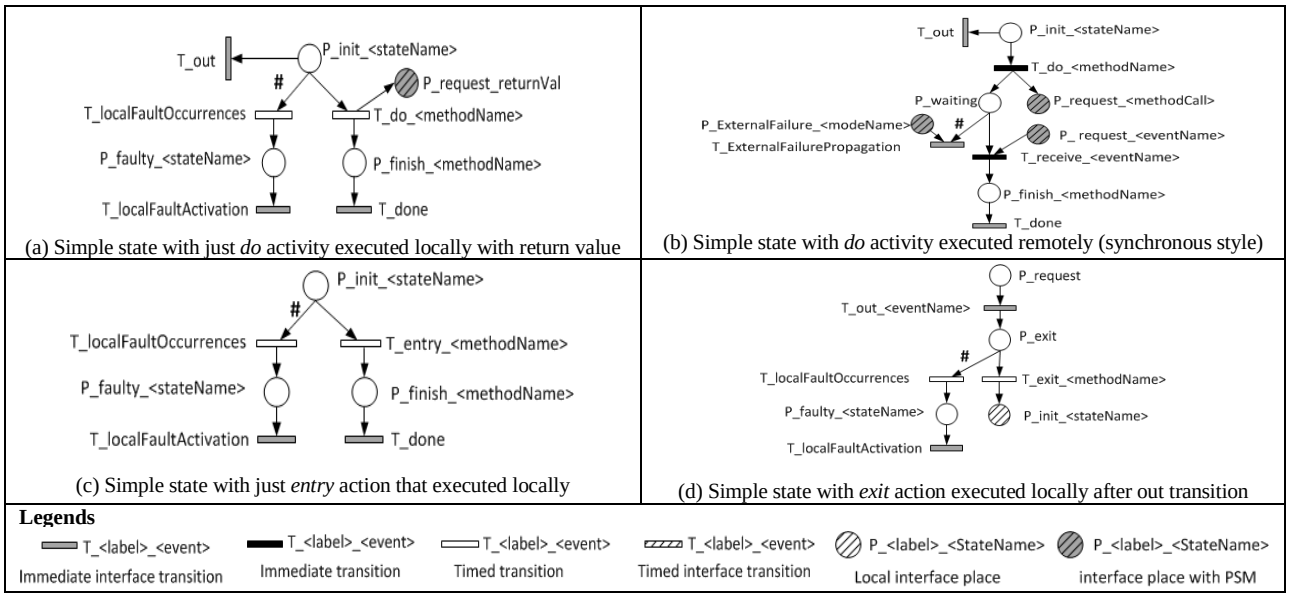


Figure 4. Graphical transformation rules for behavioral state machine.

activity of the new state. As a result, the *do* activity is interruptible and does not have a "run-to-completion" semantic allowing us to model the fault activation and transition from that new introduced state. Moreover, after applying the refactoring aspect on the original behavioral state machine, the refactored model will only have transitions with events or completion (done) transitions.

In the following section we will describe the informal semantics of how we derive the SRN from the refactored behavioral states machine that captures the internal software components behavior (normal and erroneous). In the literature, different approaches are suggested to translate the state machine to Petri net for the analysis such as [12]-[14].

Our approach of deriving SRN from BSM is inspired by [14], but in this transformation we do not show how to derive the dependability attributes of Dependability Analysis and Modeling profile (DAM) [8] from the BSM or PSM model to the SRN; we simply concentrate on how to iteratively build the SRN model from BSM and PSM for the compatibility and conformance verifications.

A. Transformation Rules for Behavioral State Machine

Simple state in our approach may represent a normal or erroneous behavior of the component. Normal state may have optional *entry/exit* actions and/or *do* activity that model method execution or signals. Erroneous states (error and failure modes), conversely, does not have any actions or activity since they capture the components dependability threats and error propagation.

Accordingly, for the normal behavior we define the different generic transformation rules shown in Fig. 4 that cover selected possible cases of a simple state. In [9] we presents all possible transformation rules with more details. In order to compose the translated model elements we adopt the idea of interface transitions and places that are presented in [14]. We have interface transition and places that connect simple state with the transitions based on matched names. We have three types of interface transitions: completion execution interface transition (*done*) that will be triggered once the state activity is completed and it no longer has any triggering events, the *out* interface transition that represents the transition between two states with a triggering event, and the *recovery* interface transition is associated with the initial place of the target state and it represents recovery from the erroneous state.

Legends in Fig. 4 describe the SRN model elements and how we graphically differentiate between different transitions and place types. To facilitate things, we concatenate the labels along with the states and transitions name in order to help identify the errors in composition and to trace them back to the conformance and composition issues in the main software model.

In all transformation rules we consider the local fault occurrences and the fault activation of every state. For instance Fig. 4a shows the translation of a simple state with the *do* activity executed locally and sends a return value. This activity may be executed successfully or exceptions may be raised during the execution that will change the component's behavior from normal to erroneous, as it propagates to the caller and then to the connected components. To capture this semantics we add two timed transitions that connect the initial place. In fact, this place (*P_init_<stateName>*) represents the entrance of the simple state. The first timed transition (*T_do_<methodName>*) captures the correct execution of the activity, while the other timed transition (*T_localFaultOccurrences*) captures the fault occurrences of *do* activity. When the first one is enabled, it will gain the tokens of the initial place. For instance, if the fault occurrences are enabled first, then all tokens will be moved to the faulty place, which means that

all pending requests of that activity will fail. The SRN multiplicity on arc that depends on marking and shown as (#) on the arc is utilized here to model the failure of all pending requests due to exceptions.

To compose this state with a connected transition, we use matched *out* or *done* interfaces transition to be composed with the state transition to the next simple state, according to the software model. Similarly local fault activation interface transition will be composed with matched erroneous transition to the erroneous state in the error propagation chain. In some cases the error may be recovered and the system can be restored to a specific state. To model the recovery, we add timed interface transition connected to the initial state as explained in [9].

Software components interact with each other in two styles: synchronous and asynchronous. Fig. 4b show synchronous styles and we note that the *do* activity in this case is modeled as an immediate transition, which represents the calling of a remote method implemented by another connected component. As mentioned earlier, the internal behavior state machine will send or receive the messages to other connected component only through its ports. Moreover, we have different interface places (shaded and striped places): 1) one for sending the request; 2) one for receiving the replay from the called method; and 3) in case the called method fails during the execution, the failure will be propagated to the caller. All these interface places will be used to connect the component port behavior to the internal behavior during the BSM and PSM composition.

Fig. 4c shows a simple state with *entry* activity. The *entry* activity, in this case, is a local method execution that may raise an exception during the execution. In order to model the normal execution and local fault activation semantics, we add two timed transitions connected to the initial place that represents the entrance of the state. First, the transition model's execution time for the method is represented, and second, the fault occurrences are represented. In the case of an *entry* action that calls a method's implementation remotely, the transformation will be same as the *do* activity translation (Fig 4b). In these transformation rules of the *entry* action, we do not violate the "run-to-completion semantic" since the transition that represents the execution of the method is not interruptible and we do not model the acceptance of a new event dispatched from the event pool. Indeed, fault activation during exaction of the action is not a new event, but it belongs to the execution path of that method. On the other hand, for the *do* activity in Fig. 4a, we have *out* transition connected to the initial place to represent the fact that the *do* activity may be interrupted during the execution by other events. In case of entry activity with just sending signals we transform it as an immediate transition, since no fault may be activated during the sending of the message [9].

According to UML standard [11], the *exit* action will be executed before exiting the current state and after receiving the external transition event. For instance, Fig. 4d shows the execution of the *exit* action after the *out* transition. Since the *exit* action in this case is a method execution, we have to model the normal execution and fault activation in the same manner of *do* activity and *entry* action. If the *exit* action is just sending signals, then the transformation will be an immediate transition. Once the *exit* action is finished, the component will enter a new state.

Error state and failure mode state represent the dependability threats of each operation in the component's behavior and it shows the fault activated and propagated or how it may be recovered. These states do not have an *entry/exit* action or *do* activity. Therefore, we transform each error state and failure mode state to a place in SRN model, which is connected with fault activation or propagation transition. Usually an error state is connected with two interface transitions. One transition represents the propagation to other error or failure mode manifestation, while the second one models the recovery transition. These transitions are timed to represent the delay of error propagation. More details can be found in [9].

In the CeBAM approach, all transitions of BSM do not have the action due to the refactoring aspect that moves all transition actions to a new intermediate state. The objective of this refactoring aspect, as mentioned earlier, is to model the fault activation and failure propagation of the transition action, without violating the "run-to-completion" semantics. These transitions will be composed with the source state by the interface transition that matches the name and label. The initial places of the target state are used as an interface place in these transformation rules. This illustrates that in all cases the state machine transition should only be connected to the initial place of the target state (the striped place in Fig. 4).

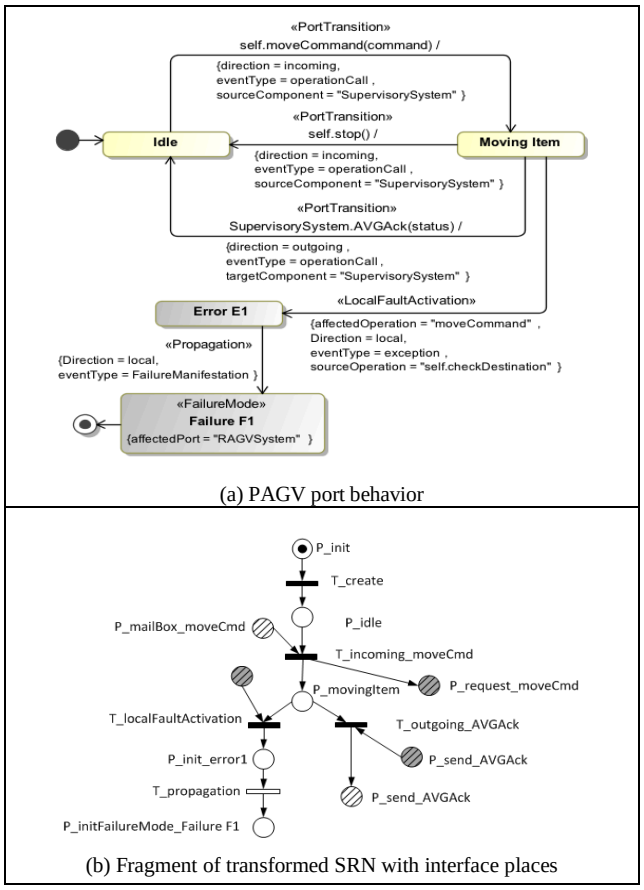
For the internal behavior of the AGV component (see Fig. 2) we apply the transformation rules explained earlier in order to get the corresponding SRN model as shown in Fig. 9. In fact, the translation to the SRN is performed in an iterative fashion. We begin by transforming each state and transition separately then using the matching interface places and transition to connect the states and transition, according to the main software model.

B. Transformation Rules for Extended Protocol State Machine

A protocol state machine (PSM) is a specialized behavioral state machine [11]. What makes PSM different is that the state in the protocol does not have an *entry/exit* action or a *do* activity. In addition, the transition does not have an action, but it has a pre and a post condition. On the other hand, the composite state and concurrent regions are permitted, but the history pseudostate are not. In our approach we are not using composite states or concurrent regions. Usually, a protocol state machine describes which operations of the classifier (i.e. interface or class), may be called in which states and under which conditions. In other word, it describes the legal usage of the classifier.

In our previous work [7], we identified the limitations of the PSM and we have extended it to model the component's port behavior. Thus, the CeBAM approach extended to the PSM to describe the external view of the component by specifying incoming and outgoing messages through each port. In fact, this extension helps us to model the failure propagation between the component interfaces in the CBD

along with their normal behavior. For instance, required and provided interfaces may be attached to the component port (see Fig. 1) and then, by using the extended PSM the order of operation calls for these interfaces, as well as failure propagation can be precisely captured (see Fig. 5a). Note that, the PSM in our approach does not show any behavioral implementation.



(a) PAGV port behavior

(b) Fragment of transformed SRN with interface places

Figure 5. Applying PSM transformation rules on PAGV component port.

All transitions of the extended PSM are atomic transitions and they possess "run-to-completion-semantics". As a result, they may be transformed to the immediate transition in the SRN model, and their states will be transformed as places. Fig. 5b shows the implementation of these rules on the PAGV port of AGV component of this case study. Note that we show the interface places that will be used to compose the port's behavior model with its internal component's behavior model and this port with other component's ports according to the component structural model (shaded and striped places to be connected with the BSM and striped places to be connect with other components ports).

C. Components Behavior Composition Patterns

The port is the property of a classifier (i.e. component). It represents a distinct interaction point between classifier and its environment and its internal behavior [11]. In UML two types of ports are defined: service ports and behavioral ports. A behavioral port will possess the implementation of its classifier and will not be externally visible. In our approach, we limit ourselves to the service port, since it lacks implementation and we use extended PSM to capture the component's external visible behavior through its stateful ports. In fact, the port PSM model will precisely capture both incoming and outgoing messages for normal and erroneous messages. These messages represent how the components interact and communicate with each other, as well as the legal usage of the interfaces attached to the port.

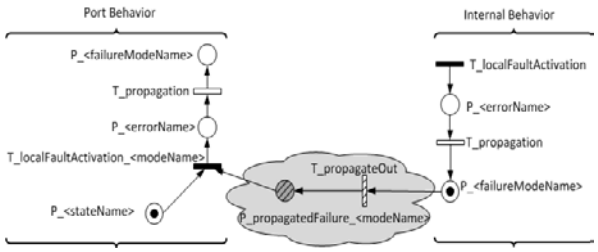


Figure 6. Composition pattern of BSM to PSM for outgoing failure propagation

According to [11], once the instance of the classifier is created, a new instance of its ports will also be created, along with its specific interaction points. A link from the port instance to the owning classifier instance will be created in order to forward any incoming requests from the environment to the owning classifier instance, or to send the outgoing requests from classifier's internal instance to the environment (to other connected components).

In composing the PSM with the BSM for normal behavior, we apply this semantics by adding an interface place between the PSM and the BSM. In fact, each provided service modeled in the component port's PSM must have a corresponding transition in the internal component's behavior (BSM) with the same event name, because otherwise the incoming request will not be handled properly. For the required services to become implemented in the other component, each activity in the BSM that calls a provided service must have a corresponding transition in the PSM with the same event name, because likewise this request will also not be passed to the connected component. These conditions must hold for all communication types (i.e. operation calls, signals and failure propagation). In Fig. 9 of the case study, we shows how we compose the BSM to its port's PSM for the normal behavior by only using interface places (gray striped places).

As explained earlier, the PSM captures the legal usage of the interfaces (provide/required) attached to the component's ports and it does not show any implementation. In fact, the implementation of the provided service is implemented as an internal component behavior. In case of the fault activated in the internal component behavior, it will be propagated to the interface's implementation causing a new error type and failure mode. Accordingly, this propagation will be shown in the port's PSM, since it describes the interface usage and it captures

its states, being either normal or erroneous. In other words, any internal failure that does not get recovered will be propagated to the component's ports, and then to the connected components. In Fig. 9 and Fig. 10 of the case study we show a failure manifested at the internal behavior as it propagates to the port, causing a new error and failure mode.

In order to show the error propagation composition from the BSM to the PSM in the transformation rules, we introduce an intermediate SRN connection subnet that connects the failure mode's place on the BSM to the fault activation transition in the PSM. For the external propagated failure from the PSM to the BSM, we will use the same connection subnet. Fig. 6 shows the propagation composition of that manifested failure in the BSM and propagated out to the corresponding PSM. For the external failure that is propagated from the other connected component, we use the same connection subnet as it explained in [9]. Note that the T*_propagateOut* is a timed transition that represents the propagation delay. We assume that this event is never lost, since it will be passed within same component that is deployed in a single machine.

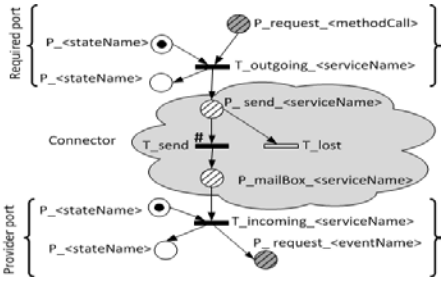


Figure 7. Service request composition pattern between two components ports (Asynchronous style)

The communication between the different instances of the UML state machine is handled by the underlying event pool, and it is usually not modeled [11]. A caller object i.e. component instance, can call an operation of another object by sending an event. An implicit event pool of the called object will receive this event. At a later point in time, the event will be dispatched from the event pool to the state machine where it will either trigger a transition, or it may be discarded. Actually, the UML 2 does not specify the semantics of the selection criteria or the priority of the event from the event pool.

In our transformation rules, the event pool is considered for each provided service or incoming signals. We have thus created a *request mailBox* place that is attached to each incoming operation call transition, and a *signals mailBox* place for each incoming signal. A *send* place is attached to each of the outgoing transitions that represent the buffering of the outgoing requests at the connector before passing it to the connected component port. Moreover, we add a timed transition (*T_lost*) from the *send* place to represent the loss of the request due to the communication channel. In fact, this transition may be replaced with a connection availability subnet that models the availability of the network connection. Note that for our current work, we do not consider the deployment.

The *send* and the *mailBox* places are connected by a *T_send* transition that has a guard to check that it is the provided service in a state that may readily consume and process the request. As shown in Fig. 7, in order to connect provided and required service between two components, we need to construct an intermediate SRN subnet consisting of a *send* place for the outgoing requests and a *mailBox* place for the incoming requests. These two places are then connected by a T*_send* transition. This composition pattern is used only if the two components are communicating in asynchronous style. Another synchronous communication style that waits for the return value or the completion of the execution signals explained in more depth in [9].

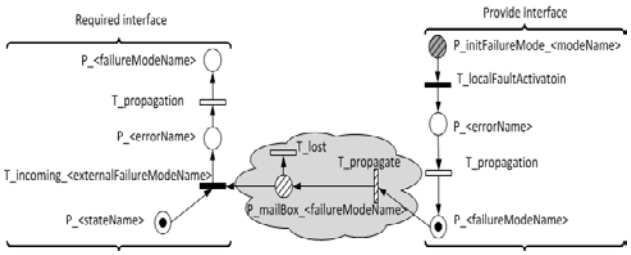


Figure 8. Failure propagation between component ports.

According to [3], if an internal fault is activated but not properly handled inside the component, then this fault will end up in a failure mode, which will be propagated to its corresponding port and to the other components that depend on it. To model the failure propagation between components, we need to add a SRN intermediate subnet that links the failure mode of the provided interface to the connected required interface. As a result, for every failure mode of the provided service, we have to have an incoming transition on the required interface that model and capture the external failure propagation. This transition will cause a new error state and failure mode of the caller component. Fig. 8 shows how we compose the failure mode of the provided service with the external propagated failure transition of the required interface by adding an intermediate SRN subnet that consists of a *propagate* timed transition to model the propagation delay between components and the *failure mailBox* place. Failure propagation events could be lost due to the communication channel. To model this, we have added a timed transition connected to the *mailBox* place.

VI. Conformance and Compatibility Verification Examples

The purpose of this work is to identify the mismatch between the connected components, their internal behavior, and their ports. In fact, this is a mandatory step in our dependability analysis approach that helps to fix all components mismatch behaviors that impact the reliability prediction results of the system. The verification process is performed in two phases. First, during the construction of

the analysis model (SRN) we identify the dangling model elements, which are not connected to other model elements. These non-connected model elements may represent the failure propagation not modeled in the connected component, or a service not utilized correctly. Second, once the analysis model is constructed, we study the properties of the SRN model, such as deadlocks and dead transitions. By interpreting the results from these two phases, we may trace back the location of the mismatch and fix it in the software model, and then run the verification process again until we get the correct model that may be used for the reliability analysis.

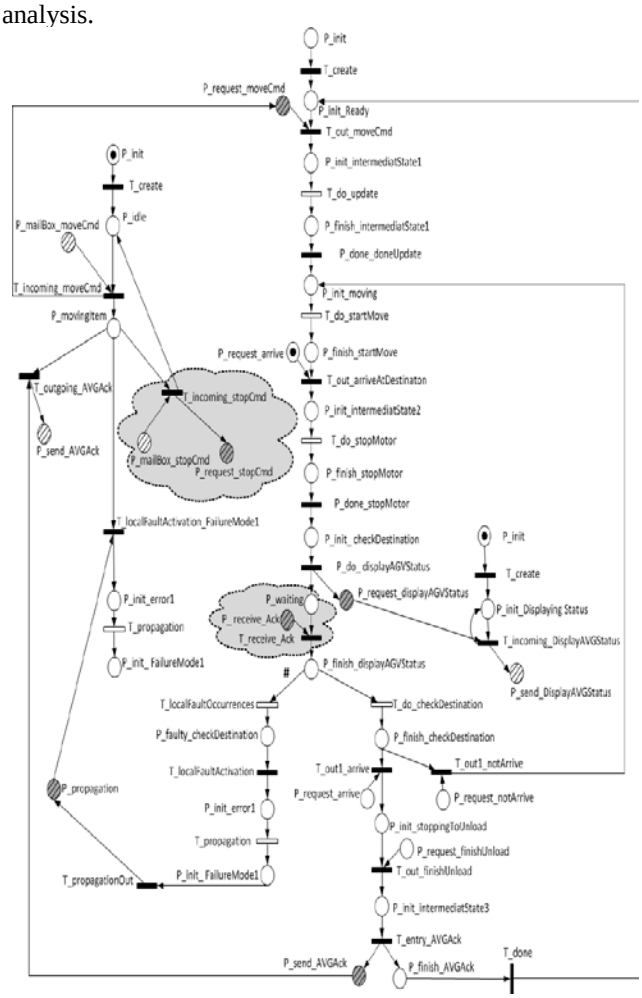


Figure 9. Example of conformances issues in internal behavior of AGV component (deadlock and dangling provided service).

PN analysis tools, such as SPNP [15], identify deadlocks as soon as they are found and they terminate the analysis execution. We utilize this feature to identify the conformance issues that are not identified during construction. For example, Fig. 9 shows a fragment of an analysis model of an AGV component and its ports that are connected to the Supervisory System and Display System components. The AGV component sends a request to the display system component through a *RDisplay* port. This call is synchronous, but the reply message is not defined in the *PDisplay* and *RAGV* ports. As a result, the internal behavior of the AGV component will wait forever for the reply from Display System component. This issue can be identified as a deadlock using the SPNP tool.

Another example of conformance issue is that, if the incoming message is modeled in the component port (PSM) and there is no corresponding event in the component's internal BSM behavior model, then that message will not be handled (possibly lost) and it will keep the requester waiting forever, especially if the call style is synchronous. Fig. 9 is an example of such issue identified in the case study. The stop service (*T_incoming_stopCmd*) is modeled in the AGV component port (*PAGV*), and it is has no corresponding event modeled in its internal component's BSM behavior model.

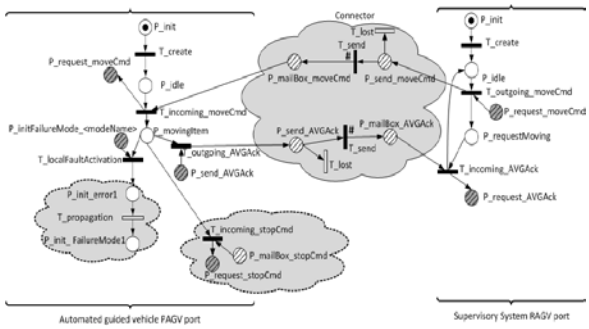


Figure 10. Example of components compatibility issues (unused provided service and nonpropagated failure)

For the compatibility verification, Fig. 10 shows the generated SRN model from the *PAGV* port of an AGV component wired with the *RAGV* port of the Supervisory System component. During construction, we have identified two incompatibility issues. The first is a dangling and absorbing place that does not connect to any transition. This indicates that the failure in the *PAGV* port is not being propagated to the *RAGV* port. We, therefore consider this as incompatibility between these ports. In fact, the ultimate goal of our research is to study the effects of the failure propagation between components on the system's reliability and how we can add a fault tolerance mechanism to handle this failure. In order to get an accurate reliability prediction, we must solve this issue in the software model and repeat the verification process as it is explained early in section four of this paper.

Second, the stop service of the AGV component is not linked to the connected component. This non-connected provided service is either extra, or not needed within its context. It may also mean that the required interface does not model it correctly to use it. As a result, this mismatch must be fixed in the software model before starting the reliability analysis.

VII. RELATED WORKS

Using formal models in verifying components' compatibility has been employed in different approaches. In [12] the focus is on the component's interface that modeled

using labeled PN. Components are compatible if the composition of the components model is free of deadlocks. However, assuming the conformance between the internal component's behavior and its interfaces, it is not helpful in making a reliability prediction without considering failure propagation. This assumption is avoided in our approach.

A similar approach is presented in [13]. The evaluation is done as part of the development environment tool called SEA. System behavior is obtained automatically by combining all BSM of all connected components to be then transformed to PN. In this approach, we notice that the application of BSM resulting from the union of the individual component BSM will not scale up for large application. In our approach of dependability analysis, we focus on the critical scenarios and we use SRN to have a more compact analysis model. Also, we consider component internal and ports behaviors in verification process for both normal and erroneous behaviors

Our transformation approach of state machines was inspired by [14]. The authors present a set of transformation rules from the UML state machine to Deterministic and Stochastic PN. Their objective is to analyze system dependability that is build based on MARTE and DAM profiles [8]. We differentiate our approach in many cases [9]. For instance, they assume implicitly that the receiver state machine will lose an incoming signal if it is not in a state that is ready to handle that signal. In our case, we have taken a different approach. As mentioned earlier, a state machine has an event pool that collects the event and dispatches it when the state machine is in a correct state. As a result, in our approach we model the buffering in the connector subnet. Moreover, the losing event will happen only if the communication link is down.

The work in [16] derive SRN from a guarded BSM for their dependability analysis. They only consider simple BSM without actions or activities. Moreover, their approach for modeling faults is limited in comparison with our approach that captures propagation. In fact, in the literature different approaches proposed to derive PN from BSM for different purposes, but none of them consider failure propagation. More detailed discussion can be found in [9].

VIII. Conclusion and Future Work

This paper presents the second phase of the proposed automated dependability prediction framework [7]. It focuses on the verification of conformance and compatibility of components modeled by the CeBAM approach. We show how we derive and compose SRN models from component's internal behavior and its port behavior, and we explain the semantics of each transformation rule. Using a case study we demonstrate how we identify conformance and compatibility issues during the composing of the SRN models and by analyzing the properties of PN.

Currently we are developing a tool that automates the proposed approach by using QVT Operational [17]. We are also working on extending the transformation rules to include the DAM profile [8] in order to analyze the system reliability and availability. Next we plan to add to the system a fault tolerance mechanism, and to predict its effect on the system in terms of reliability and availability.

Acknowledgment

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

References

[1] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed. Addison Wesley, 2012.

[2] G. N. Rodrigues, D. S. Rosenblum, and S. Uchitel, "Reliability prediction in model-driven development," Model Driven Engineering Languages and Systems, pp. 339–354, 2005.

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," Dependable and Secure Computing, IEEE Transactions on, vol. 1, no. 1, pp. 11–33, 2004.

[4] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and analysis of software systems specified with UML," Computing Surveys (CSUR, vol. 45, no. 1, Nov. 2012.

[5] A. Immonen and E. Niemelä, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," Softw Syst Model, vol. 7, no. 1, pp. 49–65, Jan. 2007.

[6] H. Aysan, S. Punnekkat, and R. Dobrin, "Error modeling in dependable component-based systems," Annual IEEE International Compu, pp. 1309–1314, 2008.

[7] N. A. Mokhayesh Alzahrani and D. C. Petriu, "Modeling Component Erroneous Behavior and Error Propagation for Dependability Analysis," Proceedings of the 16th International System Design Languages Forum Model-driven dependability engineering (SDL 2013), pp. 124–143, Apr. 2013.

[8] S. Bernardi, J. Merseguer, and D. C. Petriu, "A dependability profile within MARTE," Softw Syst Model, vol. 10, no. 3, pp. 313–336, Aug. 2009.

[9] N. A. Mokhayesh Alzahrani and D. C. Petriu, "Derivation of Stochastic Reward Net (SRN) from Component Erroneous Behavior Model for Compatibility and Conformance Verification," Technical Report SCE-13-02,Dept. of Systems and Computer Engineering, Carleton University, Canada, May 2013.

[10] H. Gomaa, "Software Modeling and Design," Cambridge University Press, Feb. 2011.

[11] OMG, "OMG Unified Modeling Language - Superstructure 2.3," May 2010.

[12] D. C. Craig and W. M. Zuberek, "Compatibility of Software Components - Modeling and Verification," presented at the Proceedings of the International Conference on Dependability of Computer Systems, 2006.

[13] N. S. Teixeira and R. P. E. Silva, "Compatibility Evaluation of Components Specified in UML," presented at the Computer Science Society, 30th International Conference of the Chilean, pp. 90–99, 2011.

[14] J. Merseguer and S. Bernardi, "Dependability analysis of DES based on MARTE and UML state machines models," Discrete Event Dynamic Systems, vol. 22, no. 2, pp. 163–178, 2012.

[15] G. Ciardo, J. Muppala, and K. Trivedi, "SPNP: stochastic Petri net package," presented at the Proceedings of the Third International Workshop on Petri Nets and Performance Models, pp. 142–151, 1989.

[16] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, and M. Dal Cin, "Quantitative analysis of UML statechart models of dependable systems," The Computer Journal, vol. 45, no. 3, pp. 260–277, 2002.

[17] OMG, "Query View Transformation (QVT) v1.1," Apr. 2008.