On the Effectiveness of a Message-Driven Confidence-Driven Protocol for Guarded Software Upgrading

Ann T. Tai Kam S. Tso
IA Tech, Inc.
10501 Kinnard Avenue
Los Angeles, CA 90024

Leon Alkalai Savio N. Chau
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

William H. Sanders
ECE Department
University of Illinois
Urbana, IL 61801

Abstract

A methodology called "guarded software upgrading" (GSU) is proposed to accomplish dependable onboard evolution for long-life deep-space missions. The core of the methodology is a low-cost error containment and recovery protocol that escorts an upgraded software component through onboard validation and guarded operation, mitigating the effect of residual faults in the upgraded component. The message-driven confidence-driven (MDCD) nature of the protocol eliminates the need for costly process coordination or atomic action, yet guarantees that the system will reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery. To validate the ability of the MDCD protocol to enhance system reliability when a software component undergoes onboard upgrading in a realistic, non-ideal environment, we conduct a stochastic activity network model based analysis. The results confirm the effectiveness of the protocol as originally surmised. Moreover, a comparative study reveals that the dynamic confidence-driven approach is superior to static approaches and is the key to the attainment of cost-effectiveness.

Keywords: Guarded software upgrading, inherent resource redundancy, error containment and recovery, checkpointing, stochastic activity networks, reliability improvement, model-based evaluation

1 Introduction

For NASA's future deep-space exploration, new-generation onboard computing systems should be able to evolve for long-term survival in unsurveyed space environments. Concepts related to evolvability include hardware reconfigurability and software upgradability [1]. Software upgradability permits spacecraft/science functions, along a mission's long-life span, to be enhanced with respect to performance, accuracy, and dependability.

A challenge that arises from onboard software upgrading is that of guarding the system against failures caused by residual design faults introduced by an upgrade. Experiences show that it is impossible to predict, during ground testing prior to uploading, all possible onboard conditions in an unsurveyed deep-space environment; thus, an upgraded embedded software component can never be guaranteed to have ultra-high reliability. There have been cases in which unprotected software upgrades or evolution caused severe damage to space missions (see [2, 3], for example), and the necessity of devising methods for dependable software upgrading was further exemplified by MCI WorldCom's recent 10-day frame relay outage [4]. The outage began August 5, 1999, four weeks after a scheduled upgrade to a new switching software intended to allow the network to handle increased traffic. The incident affected about 15% of MCI WorldCom's network and 30% of its customers who rely on the high-speed frame relay.

Although researchers have been investigating dependable system upgrade for critical applications (see [5, 6], for example, which describe two recent projects), the proposed solutions in general all require special effort for developing dedicated system resource redundancy. Due to the severe constraints on cost, mass, and power consumption of the spacecraft, NASA's deep-space applications would not be able to benefit directly from those solutions. Moreover, new-generation onboard computing systems such as the X2000, which is being developed at NASA/JPL, employ distributed architectures [1]. Accordingly, error contamination among interacting processes (caused by residual faults in an upgraded software component) becomes a major concern. However, to the best of our knowledge, methods for error containment and recovery in a distributed environment received little attention from prior work concerning dependable system upgrade (see [5, 6], for example) or dynamic program modification (see [7, 8], for example).

With the above motivation, we have developed a methodology called *guarded software upgrading* (GSU) [9]. The methodology is based on a two-stage approach. The first stage is called the *onboard validation* stage, during which we attempt to build confidence in the new version of a software component through onboard test runs under the real avionics system and environment conditions. The second stage is the *guarded operation* stage, during which

we allow the new version to actually service the mission under the protection of a protocol that intends to mitigate the effect of residual faults in the upgraded component.

Since application-specific techniques are an effective strategy for reducing fault tolerance cost [10], we exploit the characteristics of our target system and application. To ensure low development cost, we exploit *inherent* system resource redundancies as the means of fault tolerance. Specifically, we let an old version of the upgraded software component (that is already available to us), in which we have high confidence due to its long onboard execution time, escort the new version through onboard validation and guarded operation. We also make use of the processor that is active in the encountering/fly-by phases and would otherwise be idle in a cruise phase during which onboard software upgrade takes place (i.e., *non-dedicated* redundancy), allowing concurrent execution of the new and old versions.

To reduce performance cost, we take a crucial step in devising error containment and recovery methods by introducing the "confidence-driven" notion. This notion complements the message-driven (or "communication-induced") approach employed by a number of existing checkpointing protocols for tolerating hardware faults. In particular, we discriminate between the individual software components with respect to our confidence in their reliability; moreover, at onboard execution time, we dynamically adjust our confidence in the processes corresponding to those software components, according to the knowledge about potential process state contamination caused by errors in a low-confidence component and message passing. The resulting protocol is thus both message-driven and confidence-driven (MDCD). In [11], we described in detail the error containment and recovery algorithms that constitute the protocol. The central purpose of this paper is to evaluate the effectiveness of the protocol with respect to enhancing system reliability during guarded operation.

To account for potential process state contamination and message validity, we adapt the notion of "global state consistency" from the literature concerning checkpointing and rollback recovery for hardware faults [12, 13]. Based on the adapted notion, we have developed theorems and formal proofs to show that the MDCD protocol guarantees that the system will reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery [11]. The global state consistency, which is the most fundamental criterion for correct recovery, will also assure that our target system will be failure-free if the MDCD protocol is run in an *ideal execution environment* where 1) the "old" software components are truly faultless, 2) error conditions in a process state are always manifested in the messages sent by the corresponding process, and 3) the error detection mechanism employed has a perfect coverage.

On the other hand, as with any other fault tolerance schemes, the realistic goal of the MDCD protocol is to significantly reduce system failure probability rather than to assure

that the system is failure-free, since the ideal execution environment rarely exists in reality. Accordingly, we are motivated to validate, through probabilistic modeling, the protocol's effectiveness in terms of reliability improvement when the criteria for the ideal execution environment are not satisfied. To accomplish the goal requires a model to capture numerous interdependencies among system attributes. Accordingly, we choose to use stochastic activity networks (SANs) [14] to perform the analysis due to their capability of explicitly representing dependencies among system attributes. The results from the SAN-based evaluation confirm the effectiveness of the protocol as originally surmised. The analysis also provides useful insights about the system behavior resulting from use of the protocol under various conditions in its execution environment. Moreover, we conduct a comparative study to assess two MDCD variants, namely, the MDCD-O and MDCD-P protocols, that are based on static confidence-driven approaches. The assessment results demonstrate that the dynamic confidence-driven approach employed by the MDCD protocol is superior to static approaches and is the key to the attainment of cost-effectiveness.

The remainder of the paper is organized as follows. Section 2 provides an outline of the GSU methodology. Section 3 reviews the MDCD protocol. Section 4 presents a SAN-based analysis that validates the effectiveness of the protocol, followed by Section 5 which compares dynamic and static confidence-driven approaches by assessing two MDCD variants. The concluding remarks highlight the significance of this effort and discuss our future research.

2 Overview of GSU Framework

The GSU framework is based on the Baseline X2000 First Delivery Architecture [15], which is comprised of three high-performance computing nodes (each of which has a local DRAM) and multiple subsystem microcontroller nodes that interface with a variety of devices. All nodes are connected by a fault-tolerant bus network which complies with the commercial interface standard IEEE 1394, facilitating reliable onboard distributed computing.

Since a scheduled software upgrade is normally conducted during a cruise phase when the spacecraft and science functions do not require full computation power, only two processes, corresponding to two different application software components, are supposed to run concurrently and interact with each other. To exploit inherent system resource redundancies, we let the old version, in which we have high confidence due to its long onboard execution time, escort the new version software component through onboard validation and guarded operation. Further, we make use of the processor that otherwise would be idle to enable the three processes (i.e., the two corresponding to the new and old versions, and the process corresponding to the second application software component) to execute concurrently. To

aid in the description, we introduce the following notation:

- P_1^{new} The process corresponding to the new version of an application software component.
- P_1^{old} The process corresponding to the old version of the application software component.
- P_2 The process corresponding to another application software component (which is not undergoing upgrade).

Figure 1 illustrates the two-stage approach. In order to minimize the impact and risk on mission operation, onboard software upgrading is usually carried out in an incremental manner. In particular, onboard upgrades for spaceborne systems typically involve only a single software component at a time. As a result, the interaction patterns (message types and ordering) among the processes will remain the same after an upgrade. Accordingly, as shown in Figure 1(a), during onboard validation, the outgoing messages of the shadow process P_1^{new} are suppressed but selectively logged (as shown by the dashed lines with arrows), while P_1^{new} receives the same incoming messages that the active process P_1^{old} does (as shown by the solid lines with arrows). Thus, P_1^{new} and P_1^{old} can perform the same computation based on identical input data. Note that each of the dashed circles that encapsulate P_1^{new} and P_1^{old} indicates that the two processes are created by two different versions of the same application software component.

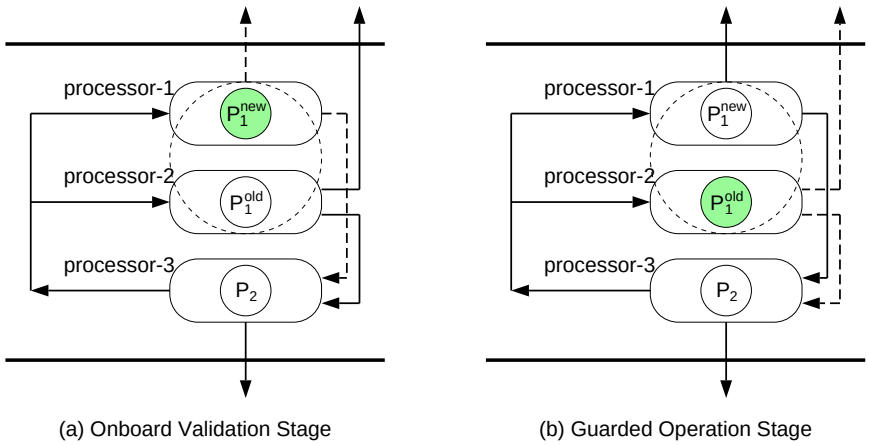

Figure 1: Two-Stage Approach to GSU

By maintaining an onboard error log that can be downloaded to the ground to facilitate statistical modeling and heuristic trend analysis, onboard validation facilitates the decisions on whether and when to permit P_1^{new} to enter mission operation. If onboard validation

completes successfully, then P_1^{new} and P_1^{old} switch their roles to enter the guarded operation stage. Since 1) P_1^{new} and P_1^{old} run concurrently and independently on two different processors prior to their role-switching, and 2) P_1^{new} maintains a message log and keeps track of the messages sent by P_1^{old}, the transition to guarded operation can be realized in a seamless fashion, with no risk of state inconsistency or message missing [9].

After role-switching brings the system into guarded operation, P_1^{new} starts to actually influence the external world and interact with process P_2, while the messages of P_1^{old} that convey its computation results to P_2 or devices are suppressed and logged, as indicated in Figure 1(b). Should an error of P_1^{new} be detected, P_1^{old} will take over P_1^{new}'s active role and the system will resume its normal mode. The guarded operation is enabled by an error containment and recovery protocol that is described in the next section.

3 MDCD Protocol

The MDCD error containment and recovery protocol is discussed in detail in [11]. In this section, we review the protocol and its properties to illustrate our motivation for the analyses conducted in Sections 4 and 5.

3.1 Design Assumptions

The following are the assumptions upon which we devise the error containment and recovery protocol:

A1) The old version of a software component that has a sufficiently long onboard execution time can be considered significantly more reliable than the upgraded version newly installed through uploading.

A2) An erroneous state of a process is likely to affect the correctness of its outgoing messages, while an erroneous message received by an application software component will result in process state contamination.

A3) The error detection mechanism, an acceptance test (AT), has a high coverage (the conditional probability that the testing mechanism will reject a computation result given that the result is erroneous).

A1 implies that the likelihood of error occurrence due to manifestation of the faults in a non-upgraded software component can be considered negligible, suggesting that P_1^{old} and P_2 need not be treated as the possible sources of process state contamination. A2 implies

that if an outgoing message is validated by AT, then the process state of the sender process and all the messages sent or received prior to performing the AT can be considered *non-contaminated* and *valid*, respectively. A3 suggests that the release of an erroneous command to an external device is unlikely to occur.

Note that A1 is applicable not only to the upgrades for performance tuning and accuracy improvement but also to the *scheduled upgrades* aimed at fault removal [16]. The rationale is that the deep-space application software components which have sufficiently long onboard execution times are expected to be highly reliable, and that the scheduled onboard fault removal usually deals with the isolated faults that result in infrequent error conditions tolerable by the spaceborne system. On the other hand, the new version with a known fault removed may contain new undiscovered faults.

3.2 The Protocol: Description and Discussion

A major difficulty in error recovery for embedded systems is that we are unable to roll back the effect of a computation error after it propagates to an external device. Since error propagation in a distributed system is, in general, caused by message passing, the invocations of the two major functions of the MDCD protocol, namely, AT and checkpointing, are all associated with the message sending or receiving actions.

We call the messages sent by processes to devices and the messages between processes *external messages* and *internal messages*, respectively. In embedded systems, external messages are significantly more critical than internal messages because i) they directly influence the mission operation and functions, and ii) their adverse effects can not be reversed through rollback. Hence, in the low-cost error containment and recovery protocol, ATs are only used to validate external messages from the processes that are potentially contaminated (see below for the definition of *potentially contaminated process state*). Further, P_1^{old} does not perform ATs, because its external messages will not be released to devices during guarded operation. On the other hand, when P_1^{new} or P_2 passes an AT successfully, it sends a notification message to P_1^{old} to let it update its knowledge about the validity of process state and messages.

We enforce the following confidence-driven checkpointing rule to facilitate error containment and recovery efficiency:

Checkpointing Rule: We save the state of a process via checkpointing if and only if the process is at one of the following points: 1) immediately before its state becomes potentially contaminated, or 2) right after its potentially contaminated state gets validated as a non-contaminated state.

By a "potentially contaminated process state," we mean i) the process state of P_1^{new} in which we have not yet established enough confidence, or ii) a process state that reflects the receipt of a not-yet-validated message that is sent by a process when its process state is potentially contaminated. Correspondingly, we use the term "a non-contaminated process state" to refer to a process state that is not potentially contaminated. Figure 2 illustrates the above concepts. The horizontal lines in the figure represent the software executions along the time horizon. Each of the shaded regions represents the execution interval during which the state of the corresponding process is potentially contaminated. In the diagram, checkpoints B_k, A_j, and B_{k+2} are established immediately before a process state becomes potentially contaminated (we call them *Type-1* checkpoints), while B_{k+1}, A_{j+1}, and B_{k+3} are established right after a potentially contaminated process state gets validated as a non-contaminated state (we call them *Type-2* checkpoints).

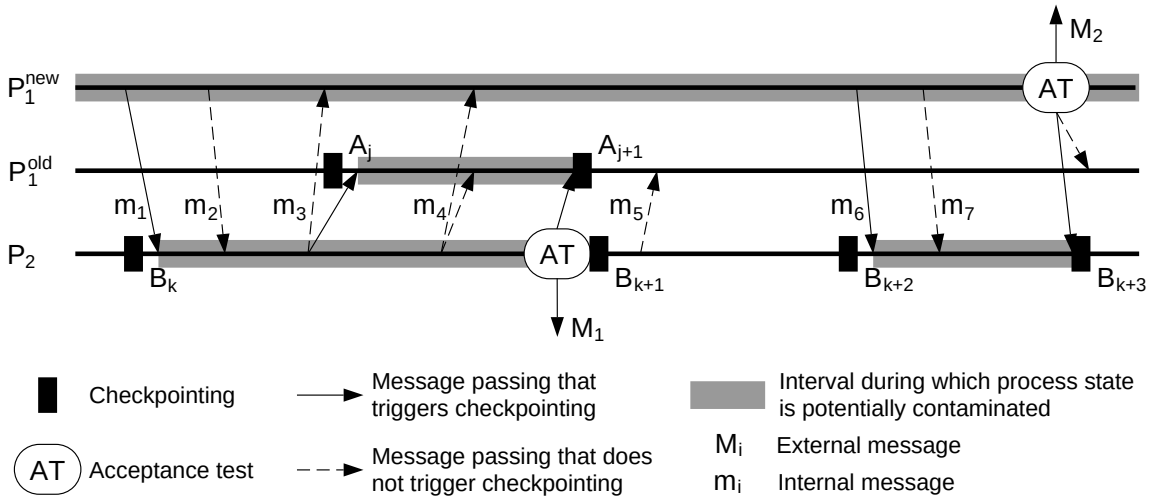

Figure 2: Message-Driven Confidence-Driven Checkpoint Establishment

While all these checkpoint establishments are triggered by the events of potential process state contamination and process state validation which change our confidence in particular processes, these triggering events themselves are induced by message passing. In other words, the checkpointing rule implies that a message passing event will not trigger a process to establish a checkpoint unless the event alters our confidence in the process state(s) — to make a potentially contaminated process state become a validated state or vice versa. For example, as shown in Figure 2, while P_2 establishes a checkpoint upon receiving message m_1 from P_1^{new} (before passing it to the application), the subsequent message m_2 from P_1^{new} does not trigger P_2 to establish another checkpoint, since the former message passing event alters our confidence in the process state of P_2, but the latter does not. Without losing generality, P_1^{new} is exempted from performing checkpointing because we invariably view it

as a low-confidence component throughout guarded operation. The error containment and recovery algorithms that constitute the MDCD protocol are shown in Appendix A. With those algorithms, P_1^{old} and P_2 will update their knowledge (through changing the value of `dirty_bit`) about potential process state contamination after and before the Type-1 and Type-2 checkpoint establishments, respectively.

Error recovery actions are also message-driven and confidence-driven. Specifically, the AT-based error detection mechanism is triggered by the event that a potentially contaminated process (P_1^{new} or P_2) attempts to send an external message. Upon the detection of an error, P_1^{old} will take over P_1^{new}'s active role and prepare to resume normal computation with P_2 (such that the MDCD protocol will go on leave until the next upgrade attempt). By locally checking its knowledge about whether its process state is contaminated or not (indicated by `dirty_bit`), a process will decide to roll back or roll forward, respectively. Accordingly, there are three possible scenarios in error recovery:

Scenario 1: Both P_1^{old} and P_2 roll back to their most recent checkpoints.

Scenario 2: Both P_1^{old} and P_2 roll forward.

Scenario 3: P_2 rolls back to its most recent checkpoint, while P_1^{old} rolls forward.

Note that our message-driven confidence-driven strategy is adapted from checkpointing techniques for hardware error recovery [12]. Nonetheless, checkpointing techniques for hardware error recovery concern solely the consistency between process states for assuring correct recovery from hardware faults. In contrast, since our objective is to mitigate the effect of residual faults in an upgraded software component, our particular concern is the consistency among the views of different processes on process state integrity, especially on *validity* of the messages (see Section 3.1) reflected in the process states. Accordingly, the confidence-driven notion leads us to adapt the terminologies and definitions in [12, 17, 13] as follows. A *global state* comprises the states of individual processes, including messages between the processes and *information concerning their verified correctness*. A valid checkpointing mechanism must assure that it is always possible for the error recovery mechanism to bring the system into a global state that satisfies the following two properties:

Consistency If m is reflected in the global state as a valid message received by a process, then m must also be reflected in the global state as a valid message sent by the sender process.

Recoverability If m is reflected in the global state as a valid message sent by a process, then m must also be reflected in the global state as a valid message received by the receiving process(es) or the error recovery algorithm must be able to restore the message m.

When two or more process states (or checkpoints reflecting the process states) comprise a global state that satisfies the consistency property, we say that these process states are *globally consistent*, or that they comprise a *consistent global state*. Based on the above concepts and the definition of potentially contaminated process state, we can informally explain why the MDCD protocol is capable of guaranteeing global state consistency for correct error recovery as follows. First, by definition, a non-contaminated process state will not reflect any not-yet-validated messages from a potentially contaminated process. This implies that, based on their most recent non-contaminated process states, P_1^{old} and P_2 will have a consistent view on message validity. In other words, the most recent non-contaminated process states of P_1^{old} and P_2 will always be globally consistent. Consequently, as the message-driven confidence-driven checkpointing rule and recovery policy together allow: 1) a potentially contaminated process to roll back to its most recent checkpoint that reflects the most recent non-contaminated state of the process, and 2) a non-contaminated process to roll forward, i.e., to continue execution from its current state (which can be viewed as the most recent non-contaminated state of the process), the MDCD protocol will lead the system to a consistent global state upon the completion of error recovery. The associated theorems and formal proofs can be found in [11].

4 Analysis of Effectiveness

4.1 Objective

As discussed in the previous section, the MDCD protocol guarantees that the system will reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery. It is worthy noting that the global state consistency can further guarantee that the system will be failure-free if the MDCD protocol is run in an ideal execution environment. By an "ideal execution environment," we mean an execution environment for the protocol that satisfies the following criteria:

C1) P_1^{old} and P_2 are perfectly reliable.

C2) Error conditions in a process state will be definitely manifested in the messages sent by the corresponding process.

C3) Each AT has a perfect coverage.

As the realistic goal of the MDCD protocol is to significantly reduce the probability of system failure rather than to guarantee the system to be failure-free, the protocol is

anticipated to be effective in a non-ideal execution environment. Accordingly, the objective of the model-based reliability analysis presented below is to validate the effectiveness of the protocol when it is run in an environment where C1, C2, and C3 are not satisfied. Before we proceed to describe the SAN model, we discuss the effect of relaxing these criteria on system failure behavior.

Clearly, an imperfect coverage of AT (which violates C3) may cause an erroneous external message to go undetected and thus lead to a system failure. And, a fault in P_1^{old} or P_2 (which is not allowed by C1) may result in an undetected external erroneous message after error recovery that brings the system back to its normal computation mode in which P_1^{new} becomes inactive and thus AT is no longer applied. Also, if error manifestation in messages is indeterministic (contrary to C2), we may have two types of undesirable scenarios in which dormant error conditions cause false confidence in process states. In the scenarios of the first type, an erroneous external message from a contaminated process would be exempted from undergoing AT. Consider the scenario illustrated in Figure 2. If a residual fault in P_1^{new} causes an error condition before P_1^{new} sends message m_1 to P_2 and the error condition is subsequently manifested in m_1, then P_2 gets contaminated. However, if the error condition in the contaminated P_2 is not manifested in M_1, the external message P_2 subsequently intends to send, the contaminated process state of P_2 would go undetected through the corresponding AT and consequently be saved in the checkpoint B_{k+1}. Suppose that after performing AT for M_1 and establishing B_{k+1} but before receiving m_6, P_2 sends another external message M' in which P_2's error condition is manifested. Then, the erroneous message M' would not be checked by AT because the process state of P_2 is considered valid. As a result, M' is released to a device, causing system failure. In the scenarios of the second type, an erroneous process state would remain after error recovery, which in turn, could eventually lead to system failure. Consider again the scenario illustrated in Figure 2. And again, we assume that P_1^{new}'s error condition is manifested in m_1 and the error condition in the contaminated P_2 is not manifested in M_1. But differing from the previous example scenario, we now do not assume the occurrence of the erroneous message M'. It follows that, after the contaminated P_2 passes AT for M_1 and establishes B_{k+1}, a subsequent contaminated state of P_2 will be saved in the next checkpoint B_{k+2} (triggered by m_6). Consequently, if P_1^{new} fails its AT when attempting to send M_2, P_2 will roll back to B_{k+2}, which contains dormant error conditions, and P_1^{old} will simply roll forward, because its process state is considered non-contaminated by the protocol (regardless of the fact that P_1^{old} may be contaminated through messages m_3, m_4, or m_5 from P_2). Although the process state of P_2 reflected in B_{k+2} and the process state of P_1^{old} upon recovery are globally consistent, the dormant error conditions may eventually cause the system to fail.

Note that criteria C1, C2, and C3 for the ideal execution environment of the MDCD protocol are similar to but stronger than assumptions A1, A2, and A3, upon which we have based the design of the protocol (see Section 3.1). Indeed, the differences between the criteria and design assumptions collectively imply, as analyzed in the previous paragraph and summarized below, the types of error conditions the MDCD protocol may encounter and be unable to mitigate when it is executed in a non-ideal environment:

- The error conditions caused by the faults in P_1^{old} and P_2, denoted as $\mathcal{E}_1^{old}$ and $\mathcal{E}_2$, respectively.
- The following subsets of error conditions caused by the faults in P_1^{new} —

  $\check{\mathcal{E}}_1^{new}$ The error conditions which are first manifested in an internal message sent by P_1^{new} (to P_2) at time t but not manifested in the first external message from the system after t (sent by P_2 or P_1^{new}), causing false confidence in process states.

  $\hat{\mathcal{E}}_1^{new}$ The error conditions which are manifested in an external message but fall outside the coverage of AT.

The above discussion suggests that it will be meaningful to quantitatively validate the effectiveness of the protocol with respect to the reliability improvement it provides under realistic, non-ideal conditions. Accordingly, we carry out probabilistic modeling by relaxing the criteria for the ideal execution environment, as described below.

4.2 SAN Models

Stochastic activity networks [18], a variant of stochastic Petri nets (SPNs), are employed in the evaluation tool *UltraSAN* [14]. Through the use of additional primitives such as *cases*, *input gates* and *output gates*, SANs have a relatively rich syntax for the purpose of specifying complex system interactions. Specifically, cases permit an expression of uncertainty about the marking that results from the "completion of an activity" (analogous to the "firing of an SPN transition"), specified by a discrete probability distribution over the cases of that activity. Moreover, the values of this distribution can depend on the marking of the network. In other words, SANs permit an explicit specification of spatial as well as temporal uncertainty. Input and output gates associated with an activity describe, respectively, how that activity is enabled and how its completion affects the subsequent marking of the network. Each of the SAN primitives has a graphical representation as follows: a place is represented as a circle, a timed activity is represented as an oval, an instantaneous activity is represented as

a solid bar, cases are represented as small circles at the right of an activity, and input/output gates are represented as triangles.

4.2.1 Largeness Avoidance

Recall that the MDCD protocol is intended to achieve error containment and recovery efficiency by discriminating between the individual software components with respect to our confidence in their reliability. Accordingly, the behaviors of the three processes, namely, P_1^{new}, P_1^{old}, and P_2, resulting from the use of the protocol exhibit little symmetry, which could lead to a complex model. However, by exploiting SANs' marking-dependent specification capability, we are able to keep the complexity of the model manageable. To further prevent the state space from being unnecessarily large, we avoid explicit representation of the algorithmic details (of the error containment and recovery protocol) which have no impact on the reliability measure. Instead, we focus on the system behavior resulting from those algorithmic details, and we assure that the model captures every aspect (of the resulting system behavior) which is necessary for the reliability measure we seek to evaluate. For example, we avoid explicit representation for the details of the message-driven confidence-driven checkpointing activities. Rather, we make the model precisely represent their impact on error recovery, by exhaustively enumerating the scenarios characterized by whether the system can recover from an error under particular system conditions that have interdependencies with checkpointing activities. The resulting SAN model is shown in Figure 3 and described in the following subsections.

4.2.2 Modeling Error Occurrence

The major components of the left part of the SAN model shown in Figure 3 are the timed activities `P1Nec`, `P1Oec`, and `P2ec`, which represent the own-fault-caused error occurrence in P_1^{new}, P_1^{old}, and P_2, respectively. By assigning a non-zero (exponential) rate to each of those timed activities, we relax criterion C1. Recall that the non-upgraded software components to which P_1^{old} and P_2 correspond are regarded as high-confidence components by the MDCD protocol, meaning that the error conditions in P_1^{old} and P_2 caused by their own faults will be neglected by the error containment and recovery mechanisms of the protocol. This necessitates different representations of error conditions caused by the faults in differing processes. Therefore, the output gates `P1Oerr` and `P2err` will result in two tokens in `P1Octn` and `P2ctn`, instead of one token (which represents the state contamination caused by the error conditions propagated from P_1^{new}), upon the completion of `P1Oec` and `P2ec`, respectively.

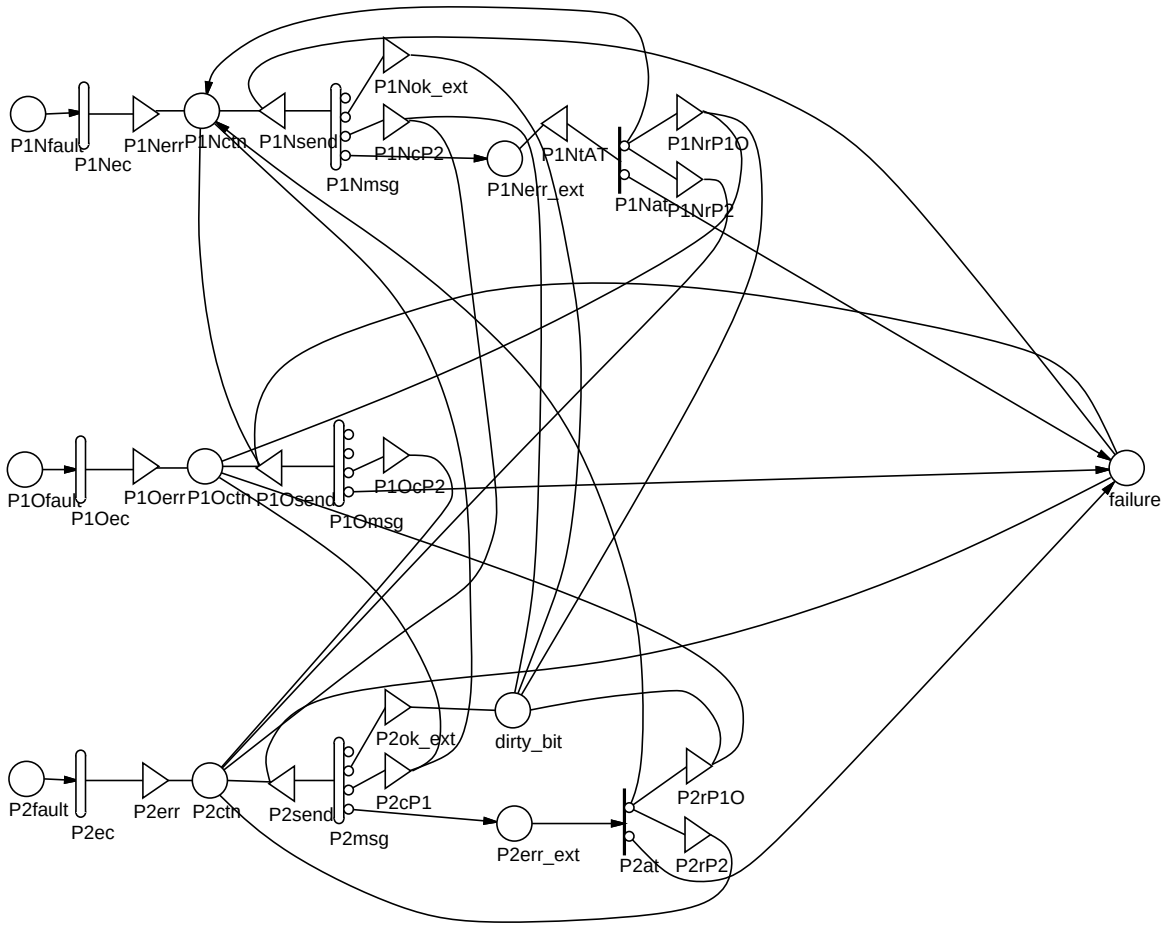

Figure 3: SAN Model for Reliability Measure of the MDCD Protocol

4.2.3 Modeling Process State Contamination

The middle part of the SAN representation (see Figure 3) comprises the timed activities `P1Nmsg`, `P1Omsg`, and `P2msg`. These three activities play important roles in representing the interdependencies among the processes in terms of error contamination. By specifying marking-dependent probability distributions over the cases of these timed activities, uncertainty about the manifestation of error conditions in a contaminated process state in the messages generated by the corresponding process is explicitly represented, which enables us to relax criterion C2.

As shown in Table 1, the possible combinations of the characteristics of an outgoing message from P_1^{new} are enumerated by the cases of the activity `P1Nmsg`. Specifically, each message is first characterized probabilistically by the external and internal message types. Furthermore, if the message is generated when the process is in an erroneous state (which will be indicated by the marking of the input place `P1Nctn`), then the message will be further

characterized probabilistically with respect to whether it is affected by the error conditions in the process state. However, for the circumstance where the process state of P_1^{new} is not erroneous (which will be indicated by the empty marking of `P1Nctn`), the above uncertainty is irrelevant. Accordingly, by assigning a zero probability to them, cases 3 and 4 which represent erroneous internal and external messages, respectively, become degenerate. The timed activities `P2msg` and `P1Omsg` are specified in a similar manner. However, since the messages of P_1^{old} are suppressed and logged before it takes over from P_1^{new}, `P1Omsg` will not be activated until the marking of `P1Nctn` indicates the mode of error recovery.

Table 1: Case Probabilities for Timed Activity `P1Nmsg`

Activity	**Case**	**Probability**

P1Nmsg	1	`if (MARK(P1Nctn)==0)`
		`/* non-contaminated internal msg from a non-contaminated state */`
		`  return(1-GLOBAL_D(prob_ext));`
		`/* non-contaminated internal msg from a contaminated state */`
		`else return((1-GLOBAL_D(prob_ext))*(1-GLOBAL_D(prob_s2m)));`

	2	`if (MARK(P1Nctn)==0)`
		`/* non-contaminated external msg from a non-contaminated state */`
		`  return(GLOBAL_D(prob_ext));`
		`/* non-contaminated external msg from a contaminated state */`
		`else return(GLOBAL_D(prob_ext)*(1-GLOBAL_D(prob_s2m)));`

	3	`if (MARK(P1Nctn)==0)`
		`/* contaminated internal msg from a non-contaminated state */`
		`  return(ZERO);`
		`/* contaminated internal msg from a contaminated state */`
		`else return((1-GLOBAL_D(prob_ext))*GLOBAL_D(prob_s2m));`

	4	`if (MARK(P1Nctn)==0)`
		`/* contaminated external msg from a non-contaminated state */`
		`  return(ZERO);`
		`/* contaminated external msg from a contaminated state */`
		`else return(GLOBAL_D(prob_ext)*GLOBAL_D(prob_s2m));`

Message-passing caused process state contaminations are represented by the output gates `P1NcP2`, `P1OcP2`, and `P2cP1`, which are connected to the cases (of the timed activities `P1Nmsg`, `P1Omsg`, and `P2msg`, respectively) representing erroneous internal messages. Because C1 is relaxed in this model whereas P_1^{old} and P_2 are not considered as the possible sources of error contamination by the MDCD protocol, we again need to make the representations of the resulting erroneous states discriminable with respect to the source of error contamination. Accordingly, each of the output gates `P1NcP2`, `P1OcP2`, and `P2cP1` first examines whether the "target" process state (P_1^{old} or P_2) is already contaminated due to "local" faults, and if so, the marking that indicates the own-fault-caused error contamination will be preserved.

The output gates `P1Nok_ext` and `P2ok_ext` are connected, respectively, to the cases of the timed activities `P1Nmsg` and `P2msg` that represent successful external message sending.

The output functions of these two gates are to reset the marking of the place `dirty_bit` to zero, which implies that the process states of P_2 and P_1^{old} are considered non-contaminated after a message passing event. These carefully specified cases and output gates play an important role in accurately and concisely representing the system behavior in a non-ideal execution environment. In particular, by resetting `dirty_bit` (through the output function of `P1Nok_ext` or `P2ok_ext`) while leaving the markings of `P1Octn` and `P2ctn` unchanged, we are able to enumerate the following scenarios without introducing separate representations for them: 1) a process that is considered potentially contaminated actually is not contaminated (by own-fault-caused error or error propagation) and thus its external message results in a successful AT, 2) a process that is considered potentially contaminated is actually contaminated but its error condition is not manifested in the external message, thus the process states of P_2 and P_1^{old} are wrongly considered non-contaminated after that message passes AT, and 3) a process that is considered non-contaminated (and may or may not be actually contaminated) sends a correct external message without undergoing AT.

4.2.4 Modeling Error Recovery

The right part of the SAN model (see Figure 3) consists of instantaneous activities `P1Nat` and `P2at`, which describe i) how a detected erroneous external message triggers recovery actions, and ii) how an erroneous external message that is undetected by AT, or exempted from undergoing AT causes system failure. Specifically, these scenarios are enumerated by the first and second cases (in the top-down order) of each of the activities as follows. For the first case, the corresponding output gates will 1) set the marking of the place `dirty_bit` to zero, and 2) if the marking of the place `P1Octn` or `P2ctn` prior to the completion of `P2at` is 1, set the marking to zero, implying that the rollback recovery brings a process to the non-contaminated state saved in its most recent checkpoint. Meanwhile, the marking of `P1Nctn` will be set to 2, indicating that P_1^{new} stops execution upon error recovery. On the other hand, if the marking of `P1Octn` or `P2ctn` is equal to 2, which implies that the state contamination is caused by an error of P_1^{old} or P_2 itself, respectively, the marking will not be altered by the output gates representing recovery actions. This is because the MDCD protocol does not consider P_1^{old} and P_2 as the possible sources of error contamination, and thus will not be able to assure that the global state after recovery will be free of the error conditions caused by P_1^{old} and P_2 themselves. The second cases of the activities `P1Nat` and `P2at` are self-explanatory — the outcome (i.e., an erroneous external message is undetected by AT or exempted from undergoing AT) will simply set the marking of the place `failure` to one. The case probability specification of `P2at`, as shown in Table 2, is also marking-dependent. This is necessary because P_2 does not perform AT for its external messages under the following

scenarios: 1) after error recovery (because the low-confidence process P_1^{new} becomes no longer active then), or 2) when P_2's process state is considered non-contaminated. It is worthy noting that the marking-dependent case probability specification indeed treats the above two scenarios as limiting cases in which the coverage of AT is zero.

Table 2: Case Probabilities for Instantaneous Activity `P2at`

Activity	Case	Probability

P2at	1	`if (MARK(dirty_bit) == 1 && MARK(P1Nctn) == 1 && MARK(P2ctn) == 1)`
		`/* AT is performed if dirty_bit is one; and the system will`
		`recover from a detected error caused by P1new */`
		`return(GLOBAL_D(at_coverage));`
		`/* AT is not performed if dirty_bit is zero; and thus an erroneous`
		`external message will be exempted from AT */`
		`else return(ZERO);`

	2	`if (MARK(dirty_bit) == 1 && MARK(P1Nctn) == 1 && MARK(P2ctn) == 1)`
		`/* AT is performed if dirty_bit is one; but an undetected error`
		`will cause system failure */`
		`return(1-GLOBAL_D(at_coverage));`
		`/* AT is not performed if dirty_bit is zero; and an erroneous`
		`external message exempted from AT will cause system failure */`
		`else return(1);`

4.2.5 Baseline System Model

In order to evaluate the effectiveness of the MDCD protocol in terms of reliability improvement, we also construct a SAN model which represents the "baseline system" where the protocol is not applied. The model is shown in Figure 4, which is quite simple and thus is not explained here.

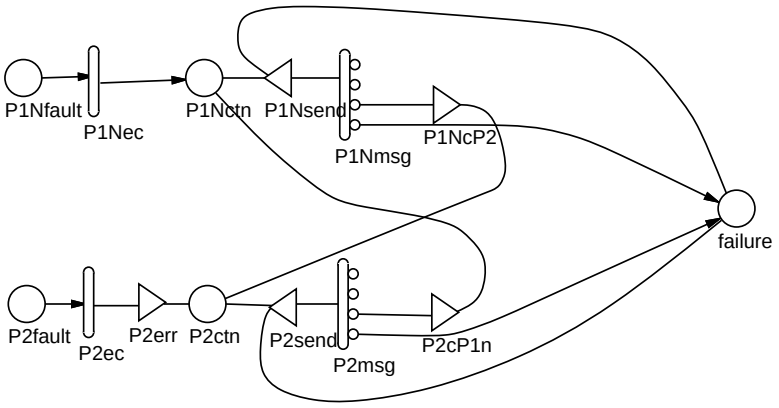

Figure 4: SAN Model for Reliability Measure of the Baseline System

4.3 Numerical Results

Based on the SAN models developed in the previous section, we analyze the effectiveness of the MDCD protocol using *UltraSAN* [14]. In particular, we define *reliability* as the probability that the system does not deliver any erroneous commands to devices (i.e., erroneous external messages) prior to time t. Letting the reliability measures for the system that applies the MDCD protocol and for the baseline system be denoted as R_t^{MDCD} and R_t^{base}, respectively, the numerical solutions of the measures can be obtained by defining a reward rate of 1 for each state of the SAN models in which the marking of the place `failure` equals zero and computing the expected reward at time t for each model.

As mentioned earlier, the central purpose of the analysis is to validate the effectiveness of the MDCD protocol in terms of reliability improvement under circumstances where the criteria for an ideal execution environment for the protocol are not satisfied. Accordingly, we focus on examining the reliability improvement in an environment where 1) the old software components (corresponding to P_1^{old} and P_2) are not perfectly reliable, 2) the probability that the error conditions in a contaminated process state are manifested in the messages generated by the corresponding process is less than one, and 3) the coverage of AT is imperfect. Before we proceed to discuss the numerical results, we define the following notation:

- μ_{new} Fault-manifestation rate of the process corresponding to the newly upgraded software version (equivalent to the rate of the timed activity `P1Nec`).
- μ_{old} Fault-manifestation rate of a process corresponding to an old software version (equivalent to the rates of the timed activities `P1Oec` and `P2ec`).
- p_{s2m} Probability that error conditions in a process state are manifested in a message generated by the corresponding process (equivalent to `prob_s2m`).
- c Coverage of an acceptance test (equivalent to `at_coverage`).
- λ Message sending rate of a process (equivalent to the rates of the timed activities of `P1Nmsg`, `P1Omsg`, and `P2msg`).
- p_{ext} Probability that the message a process intends to send is an external message (equivalent to `prob_ext`).

We first examine the effectiveness of the MDCD protocol by evaluating R_t^{MDCD} and R_t^{base}, for a mission period of 10,000 hours, as a function of μ_{new}. The value assignment for other parameters is shown in Table 3, where all parameters involving time presume that time is quantified in hours. The evaluation results are displayed in Figure 5.

Figure 5 shows that, as μ_{new} increases, the reliability improvement from applying the MDCD protocol becomes progressively more significant. Especially, after μ_{new} reaches $5 \times$

Table 3: Parameter Value Assignment

μ_{old}	p_{s2m}	c	λ	p_{ext}
10^{-8}	0.9	0.95	10	0.2

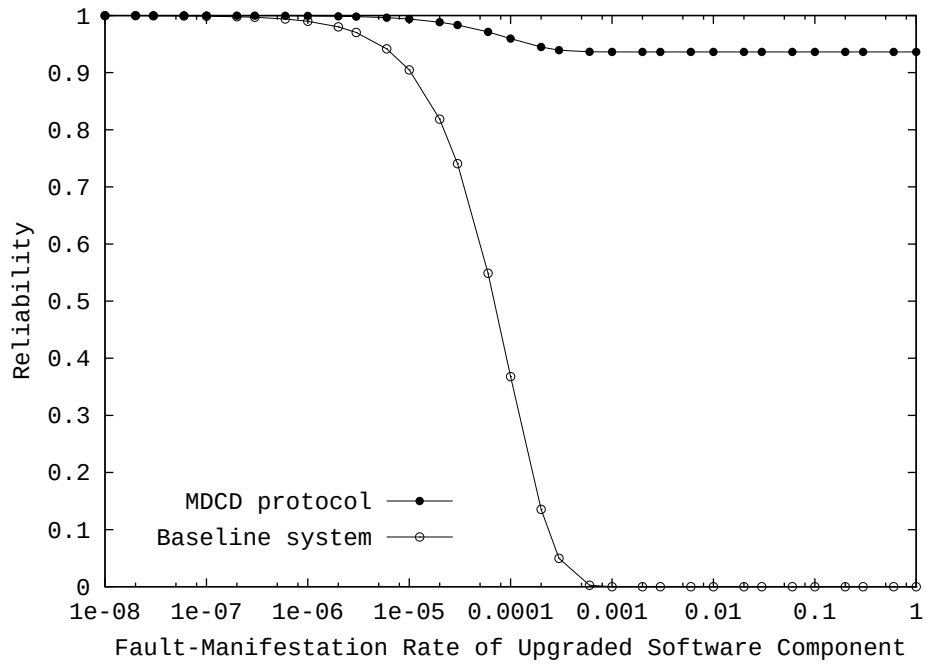

Figure 5: Reliability as a Function of μ_{new}

10^{-5}, R_t^{base} apparently becomes unacceptable while R_t^{MDCD} remains reasonable. Thus, based on this particular setting, which is rather conservative with respect to the values of p_{s2m} and c, we can observe that the MDCD protocol will offer significant benefit as originally surmised so long as the old version is appreciably more reliable than the new version. In other words, the protocol can achieve its goal without requiring that the old version of the upgraded software component be perfectly reliable. Although, due to the scale of the plot, we are unable to intuitively recognize from the curves the benefit offered by the protocol when μ_{new} is below 10^{-6}, the numerical data indeed indicate that even when the difference between μ_{old} and μ_{new} is small, the results remain favorable to the use of the MDCD protocol. Another interesting insight the curves provide is that after μ_{new} reaches 0.001, R_t^{MDCD} not only remains reasonable but also stays steady, regardless of further increase of μ_{new}. The underlying reason for this desirable result is that a higher μ_{new} will lead to a greater likelihood that error recovery will take place at an earlier time (i.e., P_1^{old} will take over from P_1^{new} sooner); as a result, μ_{old} will dominate the reliability of the system.

To confirm the above observations from a different perspective, we conduct another anal-

ysis that evaluates R_t^{MDCD} and R_t^{base} as a function of μ_{old} (t = 10,000). We again use the parameter values shown in Table 3 but fix μ_{new} at 10^{-5} and vary μ_{old}. The numerical results are shown in Figure 6. The observations obtained from these results are consistent with those from the previous study. More specifically, the reliability improvement resulting from the use of the MDCD protocol will be more significant if μ_{old} is appreciably lower than μ_{new}. On the other hand, the curves reveal that the effectiveness of the protocol increases at a much slower pace after μ_{old} decreases to 10^{-6} and becomes practically stable after μ_{old} decreases to 10^{-7}. This indicates that although the effectiveness of the protocol is in general an increasing function of the reliability of the old version, it is upper-bounded by the collective effect of other system attributes, namely, the coverage of AT, the reliability of the new version, and the likelihood of dormant error conditions that remain after message validation or error recovery.

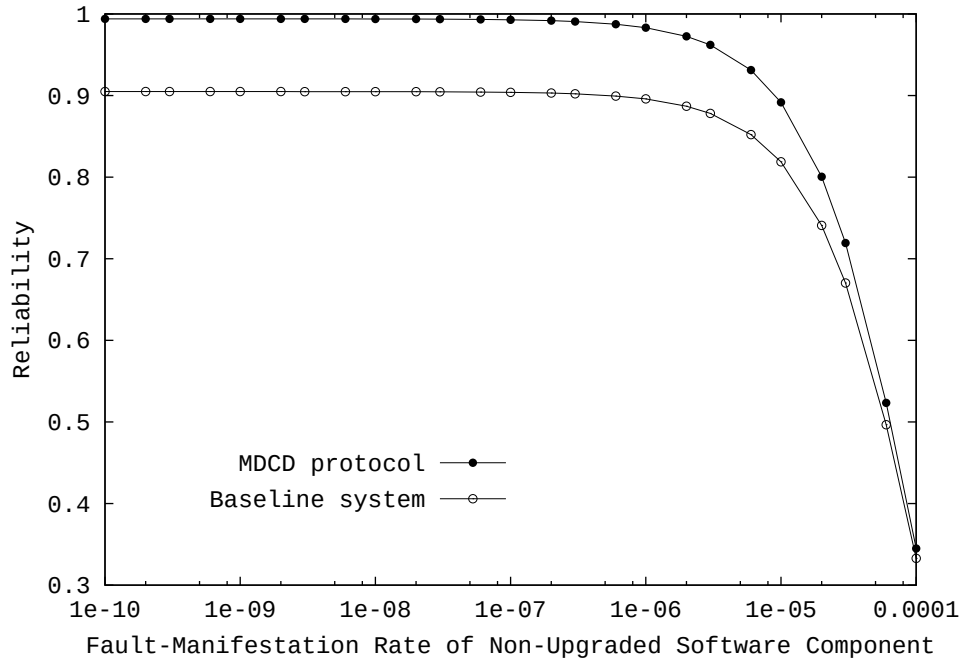

Figure 6: Reliability as a Function of μ_{old}

Next we study the effect of AT's coverage on the effectiveness of the protocol. That is, we evaluate R_t^{MDCD} and R_t^{base}, for a mission period of 10,000 hours, as a function of c. We again use the set of parameter values in Table 3 but fix μ_{new} and μ_{old} to 10^{-5} and 10^{-8}, respectively, and vary c. For the sake of illustration, we present the coverage of AT and the evaluation results (R_t^{MDCD} and R_t^{base}) in their complementary forms in Figure 7. Since AT is not employed by the baseline system, its unreliability is completely insensitive to the variations of c, as expected. The numerical results show that as long as AT's "uncoverage" is less than

0.1 (i.e., c is greater than 0.9), the unreliability reduction (i.e., reliability improvement) from applying the MDCD protocol will be significant.

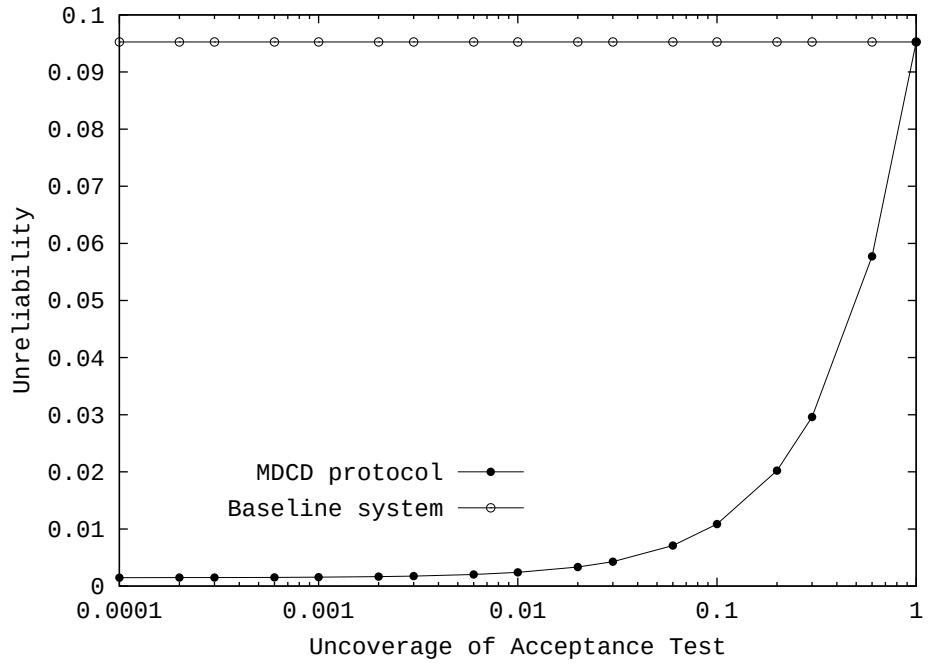

Figure 7: Unreliability as a Function of AT's Uncoverage

We also conduct an evaluation to study the effect of p_{s2m} on the effectiveness of the MDCD protocol. Rather surprisingly, reliability improvement from applying the protocol is rather insensitive to variations of this parameter. This is indeed a reasonable result because some tradeoffs exist. Specifically, while a greater value of p_{s2m} tends to reduce the likelihood of dormant error conditions that remain in process states after message validation or error recovery, it amplifies the vulnerability to error contamination among interacting processes (through error condition manifestation in internal messages). In other words, the effects cited above compensate for each other, collectively resulting in a negligible impact on the system's overall failure behavior and the effectiveness of the protocol.

5 Dynamic vs. Static Confidence-Driven Approaches

5.1 MDCD Variants

Recall that our confidence-driven approach to error containment and recovery is two-tiered: first, we discriminate between software components with respect to our confidence in their reliability, and second, we adjust our confidence in the processes corresponding to those

software components during onboard execution. The latter indeed implies that we adjudge the "trustworthiness" of a process in a dynamic manner.

To further assess the effectiveness of the MDCD protocol, potential variants of the confidence-driven approach have been studied. In particular, we contrast the MDCD protocol with two variants that employ static confidence-driven approaches. We first consider a variant which treats the processes corresponding to the upgraded and non-upgraded software components, respectively, as low-confidence and high-confidence processes in a static sense. In other words, error propagation among interacting processes during execution is ignored and the confidence level for a particular process is fixed. Accordingly, the error detection mechanism AT is applied only to the external messages sent by the process corresponding to an upgraded software component (i.e., P_1^{new}, for the application considered in this paper); whereas checkpointing is performed (by P_1^{old} and P_2) only when the system enters the guarded operation stage (i.e., immediately before P_1^{new} becomes active). Since the resulting protocol is more optimistic, relative to the MDCD protocol, we call this variant the *MDCD-O* protocol.

We then consider another variant that treats all the interacting processes, in a system where an application software component is undergoing an upgrade, as low-confidence processes in a static sense. Specifically, for our application, all the processes, including P_1^{old} and P_2, are always regarded as potentially contaminated processes. Accordingly, in the same manner as P_1^{new} does, P_2 applies AT to every external message it intends to send (followed by checkpointing if passing the AT), regardless of whether P_2's process state is influenced by a not-yet-validated message from P_1^{new}. In addition, P_1^{old} and P_2 will perform checkpointing upon every message receiving event. Because of the pessimistic nature of this variant, we call it the *MDCD-P* protocol. Table 4 compares the three protocols.

Table 4: Protocols based on Dynamic and Static Confidence-Driven Approaches

Protocol	Approach	Upgraded component (P_1^{new})	Non-upgraded component (P_1^{old}, P_2)
MDCD	Dynamic	Always treated as a low-confidence process.	Treated as a low- or high-confidence process depending upon whether its process state reflects the receipt of a not-yet-validated message from a low-confidence process.
MDCD-O	Static	Always treated as a low-confidence process.	Always treated as a high-confidence process.
MDCD-P	Static	Always treated as a low-confidence process.	Always treated as a low-confidence process.

5.2 Comparative Study

For a quantitative comparative study of the dynamic and static confidence-driven approaches, we conduct an effectiveness assessment using *UltraSAN*. With some minor modifications, we adapt the SAN model depicted in Figure 3 to compute the reliability measures for MDCD-O and MDCD-P. Specifically, in the SAN model for MDCD-O, all the functions of the output gates concerning the marking of `dirty_bit` are changed to null operation (i.e., the `identity` function of *UltraSAN*), meaning that `dirty_bit` (which initially has an empty marking) will never be set to 1. This enables the modified SAN model to characterize MDCD-O's behavior — P_1^{old} and P_2 are always treated as high-confidence processes such that P_2 never performs AT. Likewise, in the SAN model for MDCD-P, all the functions of the output gates concerning the marking of `dirty_bit` are changed to null operation while the initial marking of `dirty_bit` is set to 1, representing that P_1^{old} and P_2 are always treated as low-confidence processes such that P_2 applies AT for every external message it intends to send during guarded operation.

Applying the modified SAN models and the parameter values used for the studies presented in Section 4.3, which evaluate R_t^{MDCD} as functions of μ_{new} and μ_{old}, we evaluate the reliability measures for the variants, namely, $R_t^{\text{MDCD-O}}$ and $R_t^{\text{MDCD-P}}$, for $t = 10,000$. Figures 8 and 9 illustrate the evaluation results. Note that the numerical results of R_t^{MDCD} and R_t^{base} from Figures 5 and 6 are re-displayed, in Figures 8 and 9, respectively, for comparison.

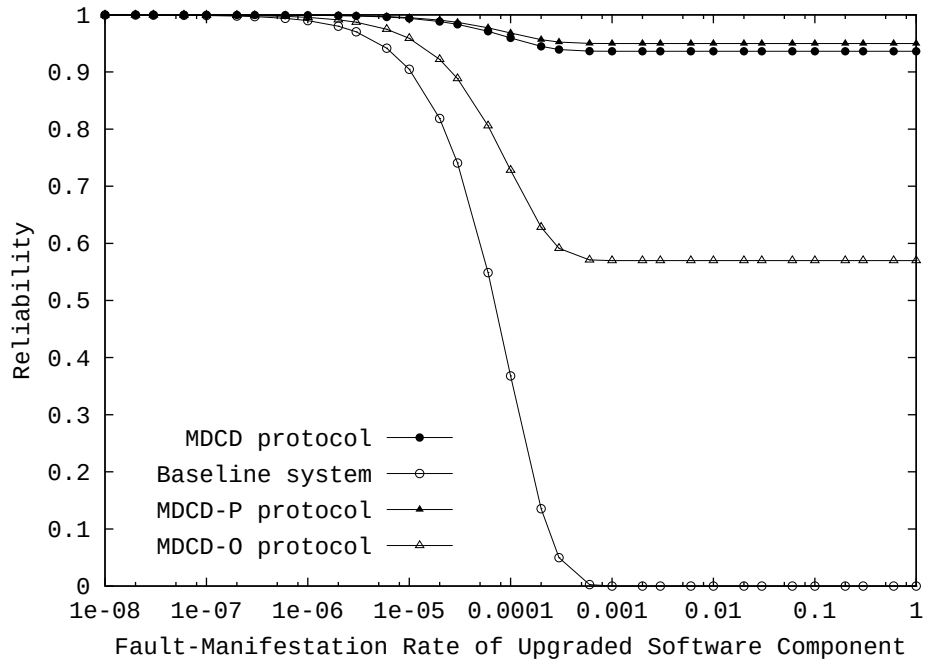

Figure 8: Comparison of Dynamic and Static Confidence-Driven Approaches (I)

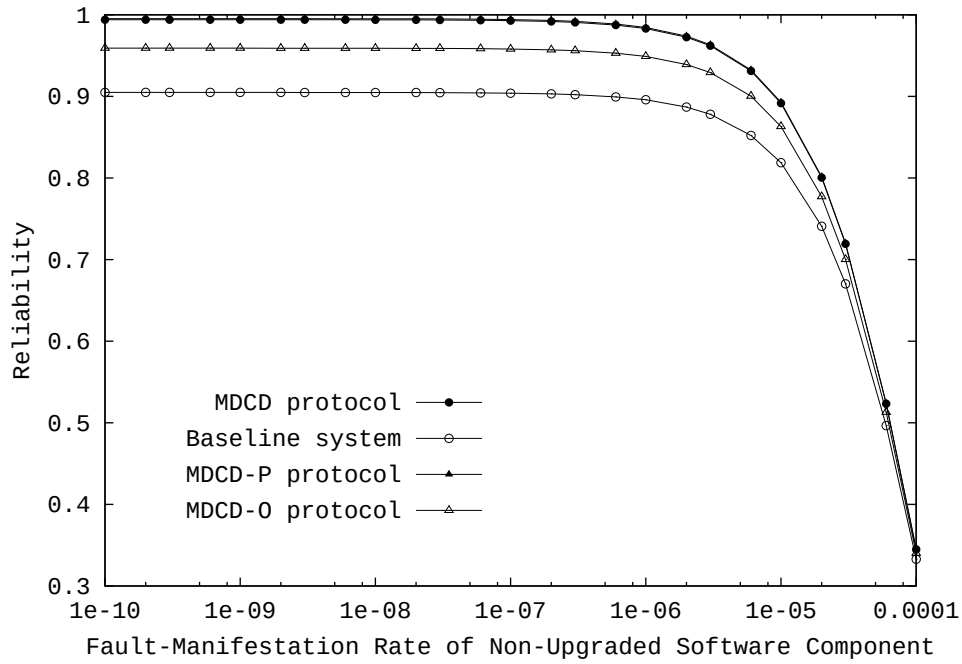

Figure 9: Comparison of Dynamic and Static Confidence-Driven Approaches (II)

Figures 8 and 9 both reveal that the optimistic static confidence-driven approach is consistently inferior to the dynamic approach. Contrasting Figure 8 with Figure 9, the former reveals more severe effectiveness reduction from using the optimistic static approach than the latter does. The difference can be understood by comparing the parameter values used in these two studies. In Figure 8, μ_{new} is a variable parameter ranging from 10^{-8} to 1, whereas in Figure 9, μ_{new} has a constant value of 10^{-5}. And in Figure 8, the reduction in effectiveness from the use of the optimistic static approach becomes progressively more severe after μ_{new} reaches 10^{-5}. As an increased value of μ_{new} will lead to a greater likelihood of contamination of P_2, the relations between the severity of effectiveness reduction and the value of μ_{new} as illustrated by the figures confirm that the penalty for use of the optimistic static approach is due to its negligence of error propagation among interacting processes.

On the other hand, the pessimistic static confidence-driven approach is capable of offering very limited effectiveness improvement over the dynamic confidence-driven approach, as shown in Figures 8 and 9 (note that the two curves consisting of the values of R_t^{MDCD} and $R_t^{\text{MDCD-P}}$ appear overlapping in Figure 9). Apparently, the practically negligible further improvement is derived at the cost of using additional ATs. To be more specific, even when its process state does not reflect the receipt of a not-yet-validated message from P_1^{new}, P_2 will still perform AT for the external message it intends to send. There are two types of error conditions that may exist and cause erroneous external messages in that situation,

namely, $\mathcal{E}_2$ and $\check{\mathcal{E}}_1^{\text{new}}$ (i.e., the error conditions caused by faults of P_2 itself and the dormant error conditions caused by indeterministic error manifestation in messages, respectively, as explained in Section 4.1). To this end, the underlying key distinction between the design of the MDCD and MDCD-P protocols becomes clear. That is, while the former is based on assumptions A1 and A2 (see Section 3.1), the latter is not. Nonetheless, as demonstrated by the numerical results described above, further reliability gain from an attempt to relax those assumptions in protocol design will be insignificant. Some insight as to why this is so can be gained by first examining MDCD-P's ability in error recovery. Specifically, when MDCD-P detects an erroneous message caused by $\mathcal{E}_2$, it will not be possible for the protocol to identify the type and/or source of the error. Due to the lack of diagnosis capability, coupled with the central goal of the protocol, a detected error would always be presumed caused by P_1^{new} and treated accordingly. Consequently, the type $\mathcal{E}_2$ error condition that is manifested in the external message of P_2 and triggers error recovery would remain in P_2's state after recovery and may eventually cause system failure. Therefore, additional error detections enabled by MDCD-P virtually offer no benefit to mitigating the effect of type $\mathcal{E}_2$ error conditions.

On the other hand, MDCD-P will enable the system to recover from error conditions of type $\check{\mathcal{E}}_1^{\text{new}}$, through process rollback and switching. Nonetheless, as discussed at the end of Section 4.3, due to the tradeoffs between the susceptibility of dormant error conditions and vulnerability to error contamination across interacting processes, the system's overall failure behavior is indeed rather insensitive to the uncertainty of error condition manifestation in messages (which is the cause of dormant error conditions). This, in turn, imposes a significant limitation on the benefit from adding to a protocol the capability for detection and recovery of type $\check{\mathcal{E}}_1^{\text{new}}$ error conditions.

We omit details of the performance-cost analysis [19] here because they are out of the scope of this paper. However, it is worthy noting that MDCD-O will suffer a great rollback distance for error recovery, whereas MDCD-P will incur 1) a higher performance overhead as the pessimistic protocol performs AT and checkpointing more frequently, and 2) the performance penalty for executing a costly decision algorithm during error recovery that identifies to which checkpoints P_2 and P_1^{old} should roll back in order to reach a consistent global state that assures correct recovery (instead of simply checking the local `dirty_bit`, as the MDCD protocol does).

Therefore, an interesting and important insight offered by this comparative study is the following: the assumptions made for the execution environment of the MDCD protocol are not only reasonable, but also the crucial basis for the conception of the dynamic MDCD approach, which, is the key to the attainment of cost-effectiveness and is superior to the static confidence-driven approaches.

6 Summary and Future Work

We have presented a study of the effectiveness of the MDCD protocol, an error containment and recovery protocol for onboard software upgrading. In order to mitigate the effect of residual faults in an upgraded software component, we introduce the idea of "confidence-driven," which complements the message-driven approach employed by a number of existing checkpointing protocols for tolerating hardware faults. In particular, we discriminate between the individual software components with respect to our confidence in their reliability. Moreover, at onboard execution time, we dynamically adjust our confidence, in the processes corresponding to those software components, according to the knowledge about potential process state contamination caused by errors in a low-confidence component and message passing. The combined message-driven confidence-driven approach eliminates the need for costly process coordination or atomic action, while guaranteeing that the system will reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery.

To validate the effectiveness of the MDCD protocol with respect to its ability in a non-ideal execution environment to enhance system reliability when a software component undergoes onboard upgrading, we have conducted a SAN model based analysis. The capability of SANs to represent the interdependencies among system attributes in an explicit fashion enables us to precisely characterize the system behavior resulting from the use of the protocol. The analysis results confirm the protocol's ability to enhance reliability for onboard software upgrading in a non-ideal execution environment. Moreover, the model-based comparative study reveals that the dynamic confidence-driven approach is the key to the attainment of cost-effectiveness for the MDCD protocol and is superior to static approaches.

Unlike the traditional software fault tolerance schemes in which recovery-point establishment and/or rollback patterns need to be pre-structured in application software, the dynamic nature of the MDCD protocol allows the error containment and recovery mechanisms to be transparent to the programmer and facilitates a middleware implementation. Currently, we are prototyping the MDCD protocol in a middleware architecture that implements the GSU methodology (called the GSU Middleware). The current version of the GSU Middleware includes 1) a group of MDCD modules responsible for assisting individual processes to maintain knowledge about process state contamination, and to make decisions on whether to take a checkpoint upon message passing and whether to roll back or roll forward during recovery, and 2) a set of invocable services (implemented as distributed objects) which execute a message sending or receiving request from an application process, in a manner adaptive to the confidence in the sender and/or receiving processes. For example, when a

low-confidence process issues a request to send a message to a device, the invokable service that executes message sending requests may make the requesting process undergo the following actions: i) performing AT and updating its knowledge about state contamination, ii) sending the application-purpose message to the device and the "passed AT" notification message to other processes, and iii) establishing a checkpoint. Whereas a similar message sending request from a high-confidence process may just simply result in a message transmission. The "overloading" characteristics of the invokable services allow the MDCD protocol to be transparent to the programmer and the middleware itself to be generic, modifiable, and upgradable. After the completion of the prototyping effort, performance and reliability benchmarking will be conducted through fault injection. In particular, as the goal of the MDCD protocol is to mitigate the effect of error conditions caused by the residual faults in an upgraded software component, we are designing the fault injection and reliability benchmarking techniques that will facilitate us to qualitatively and quantitatively analyze the coverage of the error containment and recovery mechanisms, and to validate the results and conclusions of our model-based evaluation.

As mentioned earlier in this paper, the distributed nature of the embedded systems we concern imposes further challenges to dependable onboard upgrading, which fosters the "one component at a time" onboard software upgrading policy. By exploiting this application feature, the MDCD protocol accomplishes cost-effectiveness in error containment and recovery. We have also started our initial investigation into the issue of how to protect onboard upgrades which involve multiple software components and changes of interprocess communication pattern, although this category of onboard upgrade is anticipated to be rare during a space mission. A strategy we look into is to let the GSU Middleware supply a verifiable "mediator" that enables the interprocess communication (between an upgraded software component and a non-upgraded one) which otherwise would be impossible due to mismatches in message types or ordering. In particular, the mediator will be responsible for translating the messages, through manipulating message contents and delivery, into a format the receiving process expects. In this manner, the "one component at a time" onboard software upgrading policy and the MDCD protocol will remain applicable. To be more specific, if an upgrade involves two software components and changes of interprocess communication pattern, the upgrade of the second component can be postponed until the first one, with the assistance of the mediator, goes through the guarded operation (enabled by the MDCD protocol) and becomes a high-confidence component. When the second component gets upgraded successfully in a similar fashion, the mediator completes its duty and thus retires. We plan to investigate the feasibility of the mediator-assisted approach and other alternatives for expanding the capability of the GSU methodology in future research.

Acknowledgment

The authors are thankful to the anonymous reviewers for their helpful comments. The work reported in this paper was supported in part by NASA Small Business Innovation Research (SBIR) Contract NAS3-99125.

References

[1] L. Alkalai and A. T. Tai, "Long-life deep-space applications," *IEEE Computer*, vol. 31, pp. 37–38, Apr. 1998.

[2] J. L. Lions (The Chairman of the Board), *ARIANE 5 Flight 501 Failure*, July 1996. `http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5r.htm`.

[3] A. Avižienis, "Towards systematic design of fault-tolerant systems," *IEEE Computer*, vol. 30, pp. 51–58, Apr. 1997.

[4] J. Rendleman, "MCI WorldCom blames Lucent software for outage," in *PC Week*, Ziff-Davis, August 16, 1999. `http://www.zdnet.com/pcweek/stories/news/0,4153,2318289,00.html`.

[5] L. Sha, J. B. Goodenough, and B. Pollak, "Simplex architecture: Meeting the challenges of using COTS in high-reliability systems," *CrossTalk: The Journal of Defense Software Engineering*, vol. 11, pp. 7–10, Apr. 1998.

[6] D. Powell *et al.*, "GUARDS: A generic upgradable architecture for real-time dependable systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, pp. 580–599, June 1999.

[7] M. E. Segal and O. Frieder, "Dynamic program updating: A software maintenance technique for minimizing software downtime," *Journal of Software Maintenance: Research and Practice*, vol. 1, pp. 54–79, Sept. 1989.

[8] M. E. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53–65, Mar. 1993.

[9] A. T. Tai and K. S. Tso, "On-board maintenance for affordable, evolvable and dependable spaceborne systems," Phase-I Final Technical Report for Contract NAS8-98179, IA Tech, Inc., Los Angeles, CA, Oct. 1998.

[10] J. A. Abraham, "The myth of fault tolerance in complex systems," in *Proceedings of Pacific Rim International Symposium on Dependable Computing*, (Hong Kong, China), Dec. 1999. `http://www.cs.ust.hk/PRDC1999/keynotes.html`.

[11] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "On low-cost error containment and recovery methods for guarded software upgrading," in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, (Taipei, Taiwan), pp. 548–555, Apr. 2000.

[12] E. N. Elnozahy, D. B. Johnson, and Y.-M. Wang, "A survey of rollback-recovery protocols in message-passing systems," Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Oct. 1996.

[13] N. Neves and W. K. Fuchs, "Coordinated checkpointing without direct coordination," in *Proceedings of the 3rd IEEE International Computer Performance and Dependability Symposium*, (Durham, NC), pp. 23–31, Sept. 1998.

[14] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The *UltraSAN* modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89–115, 1995.

[15] S. N. Chau, L. Alkalai, A. T. Tai, and J. B. Burt, "Design of a fault-tolerant COTS-based bus architecture," *IEEE Trans. Reliability*, vol. 48, pp. 351–359, Dec. 1999.

[16] C. T. Baker, "Effects of field service on software reliability," *IEEE Trans. Software Engineering*, vol. 14, pp. 254–258, Feb. 1988.

[17] N. Neves and W. K. Fuchs, "Adaptive recovery for mobile environments," *Communications of the ACM*, vol. 40, pp. 68–74, Jan. 1997.

[18] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proc. Int'l Workshop on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.

[19] A. T. Tai and K. S. Tso, "Verification and validation of the algorithms for guarded software upgrading," Phase-II First Interim Technical Progress Report for Contract NAS3-99125, IA Tech, Inc., Los Angeles, CA, Sept. 1999.

A Error Containment and Recovery Algorithms

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (AT(m) == success) {
      // P_1^new maintains its msg count and conveys it to P_2 and P_1^old for recovery purpose
      msg_count++;
      msg_send(m, null, device);
      // inform P_1^old and P_2 that prior messages are valid
      msg_send("passed_AT", msg_count, P_1^old);
      msg_send("passed_AT", msg_count, P_2);
    } else {
      error_recovery(P_1^old, P_2);
      exit(error);
    }
  } else { // m is an internal message
      msg_count++;
      msg_send(m, msg_count, P_2);
  }
}
if (incoming_message_m_arrives) {
  application_msg_reception(m);
}
```

Figure 10: Error Containment Algorithm for P_1^{new}

```
if (outgoing_message_m_ready) {
  msg_count++;  // msg_count keeps track of P_1^old's own messages
  msg_log(m, msg_count); // suppress and log the outgoing message
}
if (incoming_message_m_arrives) {
  if (m.body == "passed_AT") { // P_1^new or P_2 reports a successful AT
    VR_1^new = m.msg_count; // last valid msg of P_1^new
    if (dirty_bit == 1) {
      dirty_bit = 0;
      checkpointing(P_1^old);
    }
  } else { // application-purpose message from P_2
    // check the piggybacked dirty bit and own process state
    if (m.dirty_bit == 1 && dirty_bit == 0) {
      checkpointing(P_1^old);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
}
```

Figure 11: Error Containment Algorithm for P_1^{old}

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        dirty_bit = 0;
        // msg_count of P_2 keeps track of msg sequence number of P_1^new
        msg_send(m, null, device);
        msg_send("passed_AT", msg_count, P_1^old);
        checkpointing(P_2);
      } else {
        error_recovery(P_1^old, P_2);
      }
    } else {
      // outgoing msg from a clean process state, no check needed
      msg_send(m, null, device);
    }
  } else { // internal message
    msg_send(m, null, P_1^new);
    // piggybacking dirty_bit to msg to P_1^old to signal possible contamination
    m = append(m, dirty_bit);
    msg_send(m, null, P_1^old);
  }
}
if (incoming_message_m_arrives) { // must be from P_1^new
  msg_count = m.msg_count;
  if (m.body == "passed_AT") {
    if (dirty_bit == 1) {
      dirty_bit = 0;
      checkpointing(P_2);
    }
  } else {
    if (dirty_bit == 0) { // checkpointing before getting "dirty"
      checkpointing(P_2);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
}
```

Figure 12: Error Containment Algorithm for P_2

```
if (dirty_bit == 1) {
  rollback(most_recent_ckpt);
}
// switch role with P_1^new and go forward
switch_to_active(VR_1^new, msg_count);
continue;
```

(a) For P_1^{old}

```
if (dirty_bit == 1) {
  rollback(most_recent_ckpt);
}
// go forward
continue;
```

(b) For P_2

Figure 13: Error Recovery Algorithms