Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium (IPDS 2000), Schaumburg, Illinois, March 2000.

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

In order to accomplish dependable onboard evolution, we develop a methodology which is called "guarded software upgrading" (GSU). 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, safeguarding mission functions. 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, the model-based analysis provides useful insight about the system behavior resulting from the use of the protocol under various conditions in its execution environment, facilitating effective use of the protocol.

1 Introduction

The onboard computing systems for NASA's future deep-space applications must be able to enhance their own performance and dependability during a long-life mission. This capability is referred to as *evolvability* [1]. Concepts related to evolvability include hardware reconfigurability and software upgradability. 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 performance loss caused by residual design faults introduced by an upgrade. Experience has shown that it is very difficult to predict, during ground testing prior to uploading, all possible onboard conditions in an unsurveyed deep-space environment, and thus an upgraded embedded software component could not be guaranteed to have ultra-high reliability. There have been cases in which unprotected software upgrades caused severe damage to space missions, and the necessity of devising methods for dependable software upgrading was further exemplified by MCI WorldCom's recent 10-day frame relay outage [2]. The outage began August 5, 1999, four weeks after an 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 [3, 4], for example, which describe two recent projects), the proposed solutions 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 [3, 4], for example) or dynamic program modification (see [5], for example). With the above motivation, we have developed a methodology called *guarded software upgrading* (GSU) [6]. The methodology is based on a two-stage approach: the first stage is called the *onboard validation* stage, during which we attempt to establish high confidence in the new version, 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 an error containment and recovery protocol.

Since application-specific techniques are an effective strategy for reducing fault tolerance cost [7], we exploit the characteristics of our target system and application. To ensure low development cost, we exploit inherent system resource redundancies as the fault tolerance means.

*The work reported in this paper was supported in part by Small Business Innovation Research (SBIR) Contract NAS3-99125 from Jet Propulsion Laboratory, National Aeronautics and Space Administration.

Specifically, we let an old version (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 otherwise would be idle during a non-critical mission phase (i.e., non-dedicated redundancy) during which onboard software upgrade takes place, allowing concurrent execution of the new and old versions of the application software component aimed for upgrading.

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, and keep track of changes of our confidence (due to knowledge about potential process state contamination caused by errors in a low-confidence component and message passing) in particular processes. The resulting protocol is thus both message-driven and confidence-driven (MDCD). In [8], 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 [9, 10]. 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 [8]. 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 (which are regarded as the high-confidence components by the protocol) 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) [11] to perform the analysis due to their capability of explicitly representing dependencies among system attributes. The resulting model provides useful insights about the system behavior resulting from the use of the protocol under various conditions in its execution environment, facilitating effective use of the protocol.

The remainder of the paper is organized as follows. Section 2 provides background information. Section 3 describes the MDCD protocol, followed by Section 4 which presents a SAN-based analysis that validates the effectiveness of the protocol. The concluding remarks highlight the significance of this effort and outline our future research plan.

2 Background: GSU Framework

The GSU framework is based on the Baseline X2000 First Delivery Architecture, which comprises 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 the fault-tolerant bus network which complies with the commercial interface standard IEEE 1394 [12].

Since a scheduled software upgrade is normally conducted during a non-critical mission 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. 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.

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 and enter the guarded operation stage. In order to minimize the impact

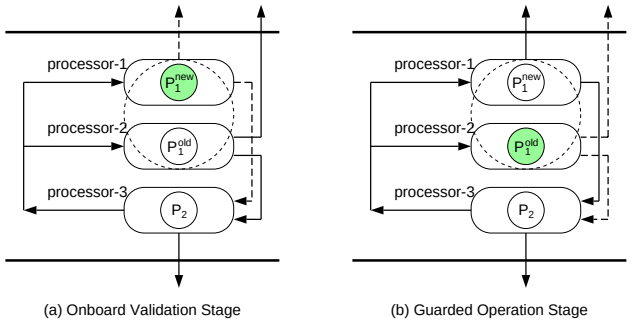

Figure 1: Two-Stage Approach to GSU

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 indicated by Figure 1(b), during the guarded operation, P_1^{new} actually influences the external world and interacts with process P_2, while the messages of P_1^{old} that convey its computation results to P_2 or devices are now suppressed and logged. 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 [8]. In this section, we review the protocol and its properties to illustrate our motivation for the analysis conducted in Section 4.

3.1 Basic 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 that an error condition will occur in the old version of an application software component can be considered negligible, suggesting that P_1^{old} and P_2 need not be treated by the protocol as 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 [13]. 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 Protocol Description

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 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.

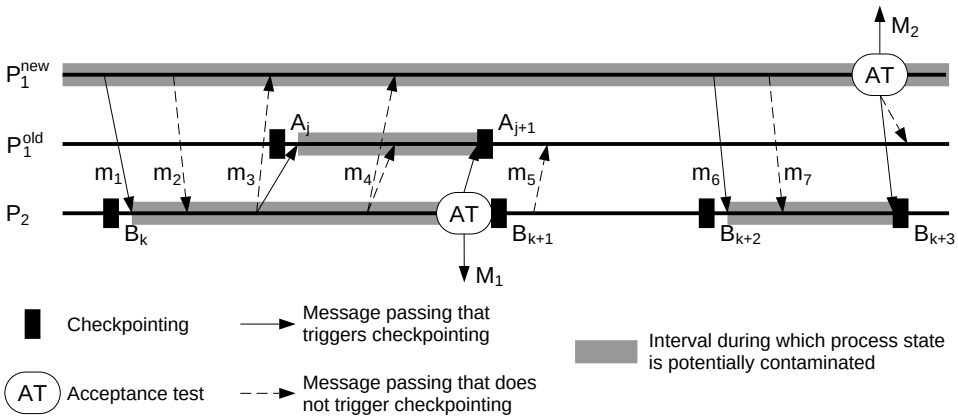

Figure 2: Message-Driven Confidence-Driven Checkpoint Establishment

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 process state gets validated (we call them *Type-2* checkpoints).

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, 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. The detailed error containment and recovery algorithms that constitute the MDCD protocol are presented in [8]. With those algorithms, P_1^{old} and P_2 will update their knowledge (through changing the value of `dirty_bit`, etc.) 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, 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 the checkpointing techniques for hardware error recovery [9]. 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 the *valid messages* (see Section 3.1) reflected in the process states. Accordingly, the confidence-driven notion leads us to adapt the terminologies and definitions in [9, 10] 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, we derive Theorem 1 and Corollaries 1 and 2 as presented below (the formal proofs can be found in [8]), which claim that the recovery decisions made locally by the individual processes assure global state consistency.

Theorem 1 *The most recent checkpoints of P_1^{old} and P_2 are always globally consistent.*

Corollary 1 *The process states of P_1^{old} and P_2 at time t that are not potentially contaminated are globally consistent.*

Corollary 2 *If at time t the process state of P_2 is potentially contaminated but that of P_1^{old} is not, then the process state of P_1^{old} at time t and the process state of P_2 reflected in its most recent checkpoint (relative to t) are globally consistent.*

Recoverability is assured by 1) the confidence-driven "rollback or roll-forward" decisions by P_1^{old} and P_2, and 2) the message log of P_1^{old} and two key entities, namely, `msg_count` and $\texttt{VR}_1^{\texttt{new}}$ (the details are omitted here but can be found in the algorithms presented in [8]).

4 Analysis of Effectiveness

4.1 Objective

The theorems presented in the previous section imply that the MDCD protocol will guarantee 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. As claimed in Section 1, 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 "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 models, we discuss the effect of relaxing these criteria on system failure behavior.

Clearly, an imperfect coverage of AT 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 may result in an undetected external erroneous message after error recovery that brings the system back to its normal computation mode in which AT is no longer applied. Also, if error manifestation in messages is indeterministic (contrary to C2), an erroneous process state may remain after error recovery, which in turn, could eventually lead to system failure. 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, then the corresponding AT will be unable to detect the state contamination. It follows that the contaminated process state will be saved in checkpoints B_{k+1} and B_{k+2}. 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). In order to validate the effectiveness of the protocol with respect to the reliability improvement it provides under realistic, non-ideal conditions, we carry out probabilistic modeling by relaxing the criteria for the ideal execution environment, as described below.

4.2 SAN Models

Stochastic activity networks, a variant of stochastic Petri nets (SPNs), are employed in the evaluation tool *UltraSAN* [11]. 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. More precisely, input gates permit a functional specification of the enabling predicate and how the markings of input places change; output gates specify how the markings of the output places are altered when the activity completes.

Recall that the MDCD protocol is intended to achieve error containment and recovery efficiency by discriminating

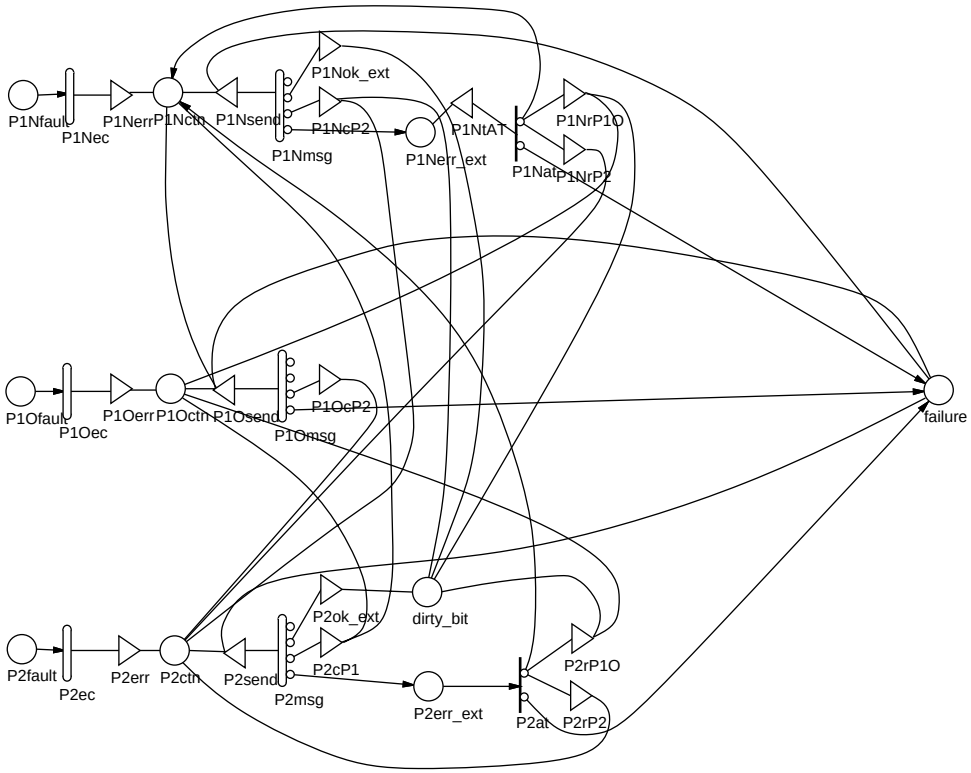

Figure 3: SAN Model for the System that Applies the MDCD Protocol

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 obtain a relatively concise SAN model that captures all the relevant details of the system behavior resulting from the MDCD protocol, as shown in Figure 3.

The SAN representation can be viewed as consisting of three parts. The major components of the left part are the timed activities `P1Nec`, `P1Oec`, and `P2ec`, which represent the error condition occurrence in P_1^{new}, P_1^{old}, and P_2, respectively. By assigning a non-zero (exponential) failure rate to each of the timed activities, we relax criterion C1. Recall that P_1^{old} and P_2 are regarded as high-confidence components in the system 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, while the output gate `P1Nerr` sets the marking of the output place `P1Nctn` to 1 upon the completion of `P1Nec`, the output gates `P1Oerr` and `P2err` will result in two tokens in `P1Octn` and `P2ctn` upon the completion of `P1Oec` and `P2ec`, respectively.

The middle part of the SAN representation 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`

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));`

Table 2: Case Probabilities for Instantaneous Activity `P2at`

Activity	Case	Probability

P2at	1	`if (MARK(P1Nctn) == 1 && MARK(dirty_bit) == 1)`
		`/* AT is performed before recovery if dirty_bit is one */`
		`  return(GLOBAL_D(at_coverage));`
		`/* AT is not performed after recovery or if dirty_bit is zero */`
		`else return(ZERO);`

	2	`if (MARK(P1Nctn) == 1 && MARK(dirty_bit) == 1)`
		`/* AT is performed before recovery if dirty_bit is one */`
		`  return(1-GLOBAL_D(at_coverage));`
		`/* AT is not performed after recovery or if dirty_bit is zero */`
		`else return(1);`

will not be activated until the marking of `P1Nctn` indicates the mode of error recovery.

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 the error contamination. Accordingly, each of the output gates of `P1NcP2`, `P1OcP2`, and `P2cP1` first examines whether the "target" process state (P_1^{old} or P_2) is already contaminated by its own error, and if so, the marking that indicates the own-error-caused process state 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 validated by a successful AT. Although P_2 will not perform AT for its external message if the message is generated when P_2's process state is considered non-contaminated according to the MDCD protocol, a separate representation for this scenario is not required. This is because the marking of `dirty_bit` would be zero before the completion of the activity `P2msg` for this scenario and thus resetting will have no effect. This, in turn, implicitly represents the scenario in which P_2 sends a correct external message (when its process state is considered non-contaminated) without performing AT.

The right part of the SAN model consists of instantaneous activities `P1Nat` and `P2at`. The first and second cases (in the top-down order) of each activity represent, respectively, the scenarios where an erroneous external message that is detected by an AT triggers error recovery and an undetected erroneous external message causes system failure. 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 comple-

tion 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 to be 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 `P1Oat` and `P2at` are self-explanatory, i.e., the outcome (an undetected erroneous external message) will simply set the marking of the place `failure` to 1. 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, or 2) when its process state is considered non-contaminated. It is worth 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.

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.

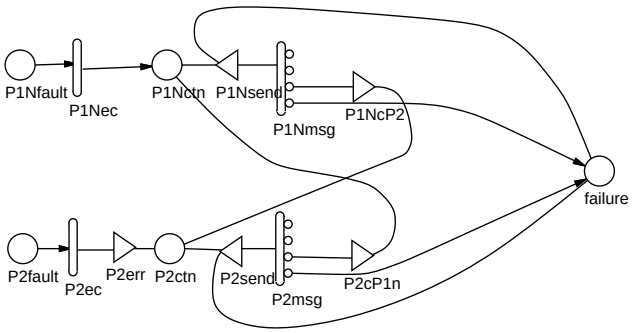

Figure 4: SAN Model for the Baseline System

4.3 Numerical Results

Based on the SAN models developed in the previous section, we analyze the effectiveness of the MDCD protocol using *UltraSAN* [11]. 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} Failure rate of the process corresponding to the newly upgraded software version (equivalent to the rate of the timed activity `P1Nec`).
- μ_{old} Failure 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 the parameters involving time presume that time is quantified in hours. The evaluation results are displayed in Figure 5.

Table 3: Parameter Value Assignment

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

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 5×10^{-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

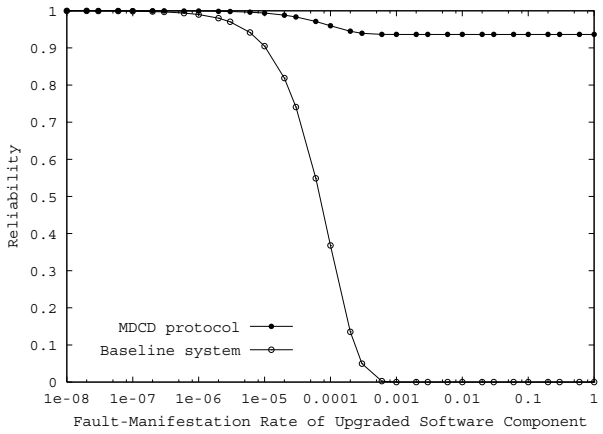

Figure 5: Reliability as a Function of μ_{new}

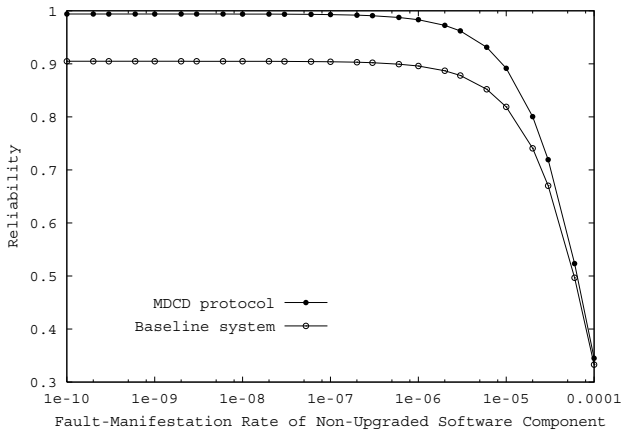

Figure 6: Reliability as a Function of μ_{old}

software component be perfectly reliable. Although we are unable to intuitively recognize from the curves the benefit offered by the protocol when μ_{new} is below 10^{-6} due to the scale of the plot, 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 in the failure rate of the new version. 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 analysis 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 the dormant error conditions that remain after recovery.

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.

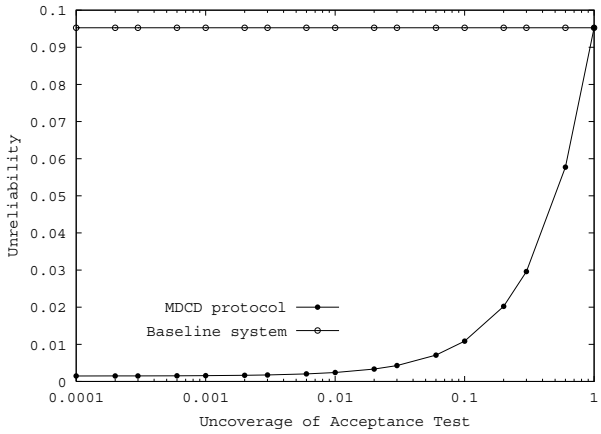

Figure 7: Unreliability as a Function of AT's Uncoverage

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 relatively 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 recovery, it amplifies the vulnerability to error contamination among interacting processes (through error condition manifestation in internal messages). In other words, the two effects compensate for each other, collectively resulting in a negligible impact on the effectiveness of the protocol.

5 Summary and Future Work

We have presented an analysis 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, and keep track of changes of our confidence in particular processes. 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 explicitly represent the interdependencies among system attributes 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 analysis provides useful insights about the system behavior resulting from the use of the protocol under various conditions in its execution environment. Coupled with the observations from our studies on the performance cost of the MDCD protocol, the results of this model-based analysis are leading us to analyze the tradeoffs in the error containment and recovery mechanisms for further improvement of the protocol.

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 GSU Middleware). The initial version of the GSU Middleware comprises 1) a group of agents responsible for maintaining knowledge about process state contamination, and making decisions for individual processes 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 with "overloading" characteristics, which allows the protocol to be transparent to the programmer and the middleware itself to be generic, scalable, and upgradable. After the completion of the prototyping effort, performance and reliability benchmarking will be conducted through fault injection; the results will be compared with those from the model-based studies, and be used to improve the algorithms and methodology.

References

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

[2] J. Rendleman, "MCI WorldCom blames Lucent software for outage," in *PC Week*, Ziff-Davis, August 16, 1999.

[3] 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*, pp. 7–10, Apr. 1998.

[4] 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.

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

[6] 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.

[7] 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`.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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