Proceedings of the 18th Digital Avionics Systems Conference, St. Louis, MO, October 24-29, 1999.

On-Board Guarded Software Upgrading for Space Missions[*]

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
CRHC
University of Illinois
Urbana, IL 61801

1 Introduction

The evolvable avionics systems such as the X2000 at NASA/JPL are able to have software upgrades during a long-life mission for dependability, performance and functionality improvement (we call it "on-board software upgrade") [1]. While evolvability itself can be viewed as on-board perfective maintenance, it necessitates preventive maintenance and corrective maintenance for eliminating or mitigating potential error conditions caused by residual faults in an upgraded system configuration or software version, and tolerating possible inconsistencies between the old and new configurations/versions. We collectively view the three types of mechanisms as *on-board maintenance* and have been investigating into the development and implementation issues [1, 2]. To date, on-board software upgrade still requires to reboot the entire flight software for terminating the old version and starting the new one. In the Mars Pathfinder mission, it took two hours to complete the patch process for two small changes (in the flight software) made as a result of Operational Readiness Test, during which the normal functions of the flight computer was stopped [3]. The cost of the unavailability is apparently unacceptable for the future NASA missions.

Other types of deficiency in software upgrading may even cause more severe damages to a mission. For example, NASA experienced a gap in fault protection on April 10, 1981, when a timely synchronization check was omitted after the addition of an alternate reentry program [4]. As a result, the first flight of the US space shuttle program was aborted 19 minutes before launch. The necessity for guarded software upgrading is further testified by the recent event: At 39 seconds after launch, the Ariane 5 self-destruct mechanism activated, obliterating the rocket; the Ariane 5 was an upgrade of the Ariane 4. The upgraded software, based on in part of the Ariane 4 software, could not handle the higher velocities of the Ariane 5 [5].

In summary, with respect to long-life applications, the key factors that necessitate on-board maintenance for evolvable software are: 1) potential loss of system availability, normal mission functions, and message/data during uploading and version switching; 2) possible gaps between the old and new software versions with respect to their computation algorithms, or fault protection mechanisms; and 3) residual design faults introduced by the addition or modification of a spacecraft/science function. In this paper, we describe a methodology, called guarded software upgrading (GSU), that enables seamless and dependable on-board software upgrading and feasible for middleware implementation.

The error containment and protection methods for GSU are based on checkpointing, message logging and rollback/roll-forward recovery techniques [6, 7, 8, 9] that are adapted and extended to accommodate the requirements from the X2000 architecture and applications. The same methodology can be applied to the two stages of guarded software upgrading, namely, on-board validation and guarded operation, as well as version switching for the transition from the first stage to the second stage, to minimize performance degradation and prevent message/data loss.

The remainder of the paper is organized as follows. Section 2 provides a scenario-based description of our GSU methodology, followed by Section 3 presents a middleware architecture for the GSU methodology realization. The concluding section discusses the signif-

[*]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.

icance of this effort and our plan for the subsequent research.

2 Guarded Software Upgrading

2.1 General Approach

The following methodology design is based on the X2000 architecture configuration with three processors, two of which are required to perform spacecraft/science functions during a non-critical mission phase. With the GSU methodology, the software aimed for upgrade will go through two stages:

- In the first stage, *on-board validation*, we confirm, through on-board test runs, a software version's ability to perform its functions complying with the mission requirements under the current conditions of the spaceborne system and environment.
- In the second stage, *guarded operation*, we allow the new software version to actually perform its science or spacecraft functions for the mission (version switching after confidence is established through on-board validation), guarded by the old version and a set of error containment and protection mechanisms.

As shown in Figure 1 during the validation stage, the upgraded version is executed concurrently with the old version and its computation results are suppressed but are checked by an acceptance test or compared with the results from the old version; at this stage, checkpointing is used to enable the upgraded version to recover from a detected error by restarting from a state copied from the old version, based on checkpointing.

Since the checkpointing-based error containment and protection technique facilitates the maintenance of an on-board error log and classification of error events, those error data can be downlinked to the ground to facilitate statistical modeling and heuristic trend analysis. This in turn, will be used to support decision making on whether and when to switch the versions. During version switching, message logging enables the avoidance of missing messages and redundant messages.

After the upgraded software version enters its actual operational phase, the computation results of the old version will be suppressed although its execution continues. When an error is detected in the upgraded version, the old version will take over the operation based on checkpointing and message logging.

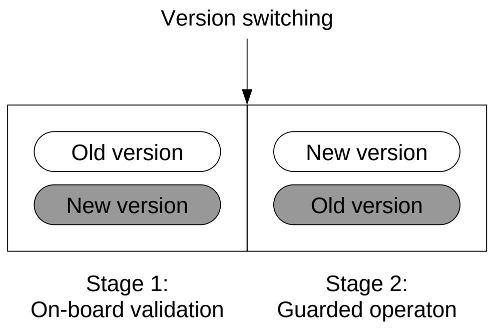

Figure 1: Two-Stage Approach

To facilitate the description of our checkpointing and messaging logging schemes for error containment and recovery, we introduce the following notation:

V_{old}	The process corresponding to the software version before an upgrade.
V_{new}	The process corresponding to the upgraded software version.
P_c	The cooperative process (another program in the distributed system) which runs on the third processor and interacts with V_{old} and/or V_{new}.

Further, we call the version in use (i.e., its outgoing messages are actually sent to a device or a cooperative process) an *active version*, and refer the version which executes with its outgoing messages suppressed as a *shadow version*.

The algorithms are message driven in the sense that, checkpointing and message logging are performed before or after a process sends or receives a message. An acceptance test is performed for messages to be sent to a device (or anywhere external to the computing system). More precisely, acceptance test is carried out after a process generates a message and before it is transmitted. The following are our basic assumptions:

1. Messages between processes and between a process and a device are functions of input data (received by a process from incoming messages or

from instrument readings). For the same input data, the output data (message) from V_{new} and V_{old} shall be equivalent.

2. An earlier version of the software and the software comprising the cooperative process are more reliable than the upgraded version.

3. An erroneous state of a process is likely to affect the correctness of its outgoing messages, while an erroneous message received by an application will result in an erroneous state of the corresponding process. Therefore, the correctness of checkpoints and messages can be traced and analyzed for rollback or roll-forward recovery.

To implement the methodology, we design a middleware architecture, the GSU Middleware, which enables the checkpointing and message logging mechanisms to be transparent to the programmer. For example, the GSU middleware provides a message sending service that can be invoked by an application software — `send()`. This application invokable service may involve one or some combination of the following: 1) actual message transmission, 2) message suppression, 3) message logging, 4) taking a checkpoint, and 5) performing acceptance test. When `send()` is invoked by an application, the GSU Middleware selects action(s) from the above set based on (see Section 3 for more design details):

- *Execution context* (in which the message is sent):
  - Version switch
  - On-board validation
  - Guarded operation
- *Execution process* (that sends the message):
  - Active version process
  - Shadow version process
  - Cooperative process

2.2 On-Board Validation and Seamless Version Switching

During on-board validation and seamless version switching, V_{new} is the shadow process and V_{old} is the active process; different processes in the system perform checkpointing and message logging as follows.

- **V_{new}**

  Checks correctness of every message it intends to send by applying acceptance test and logs the message with a sequence number, during its execution as a shadow.

  Deletes a message from the log after it receives the notification from V_{old} that indicates the corresponding message (identified by message sequence number) has and been sent (by V_{old}).

  Takes a checkpoint after it receives a message from P_c and before passing it to the application.

  Upon version switching, it becomes the active version and the logged messages are sent to P_c or devices, if its message log is not empty.

- **V_{old}**

  Actually sends out the messages to P_c and devices. When it sends a message, it 1) sends a copy of the message together with the message sequence number to P_c or a device; and 2) sends the message sequence number and a copy of its state, or just the former, to V_{new}. Sending its complete state provides another means for V_{new} to check its correctness and enables the recovery upon error detection (at the cost of additional data transmission).

- **P_c**

  Takes a checkpoint which saves the message sequence number after it receives a message from V_{old} and before passing it to the application.

  When it sends a message to V_{old}, it also sends a copy of the message to V_{new}, such that V_{new} can execute in the same control flow as V_{old} does.

  When it receives messages from V_{new} after version switching, it checks whether a message is redundant by comparing the sequence number with that of the latest message received from V_{old}.

Figure 2 illustrates the scenario in which V_{new} directly goes to normal operation ("roll forward"), upon version switching, without re-sending the suppressed messages from its message log or suppressing any newly generated messages. This is because the global state satisfies the consistency property (as defined in [9]) when version switching is initiated.

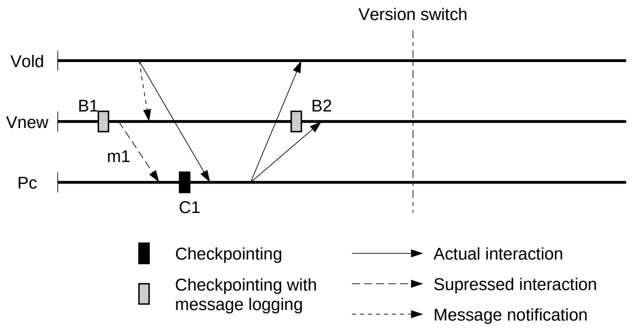

Figure 2: Version Switch without Message Re-send or Suppression

Figure 3 depicts the scenario of version switching (for the transition from on-board validation to guarded operation) in which V_{new} needs to re-send the suppressed message m2 from its message log, upon version switching. And the message will undergo an acceptance test before transmission, as the version switching brings the system to the guarded operation stage. In this case, message-logging (of V_{new}) assures recoverability (as defined in [9]) and prevents the system from losing messages during upgrade.

Figure 4 describes the scenario in which V_{new} needs to suppress message m2, after version switching and when it executes to the point where the message is generated. In this case, message sequence number plays a crucial role for avoidance of redundant messages.

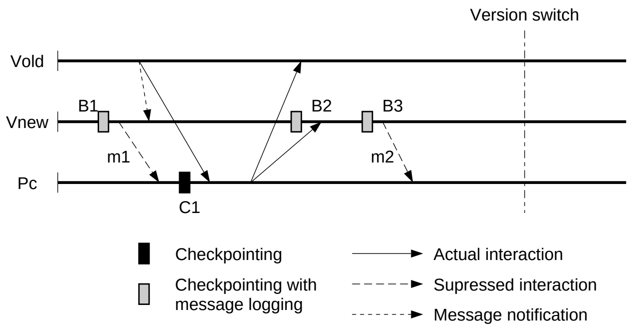

Figure 3: Version Switch with Message Re-send

2.3 Guarded Operation

During guarded operation, V_{new} becomes the active and V_{old} becomes the shadow; different processes in the system perform checkpointing and message logging as follows.

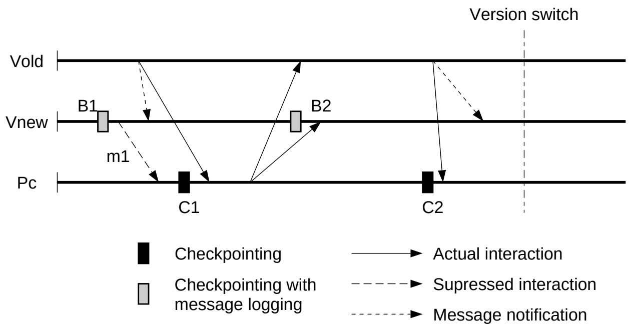

Figure 4: Version Switch with Message Suppression

- **V_{new}**

  Performs acceptance test when it needs to send a message to a device. When it sends a message to device or P_c, it sends the message sequence number to V_{old}.

- **V_{old}**

  Checks the correctness of every message it intends to send using acceptance test and logs it with a message sequence number, during its execution as a shadow.

  Deletes from the log the messages with sequence numbers smaller or equal to the sequence number piggybacked on the latest message from P_c that identifies the last message it received from V_{new} before its most recent successful acceptance test.

  Takes a checkpoint after it receives a message from P_c and before passing it to the application.

  Upon error recovery, it becomes the active version and the logged messages are sent to P_c or devices, if its message log is not empty.

- **P_c**

  Performs acceptance test on the message it intends to send to a device and takes a checkpoint immediately after it sends out the message which passes acceptance test.

  Takes a checkpoint which saves the message sequence number after it receives a message from V_{new} and before passing it to the application.

  When it sends a message to V_{new}, it also sends a copy of the message to V_{old}, such that V_{old}

can execute in the same data and control flows as V_{new} does.

Figure 5 illustrates the scenario in which V_{new} fails on acceptance T2 and V_{old} will subsequently take over without rollback, re-sending logged messages or suppressing newly generated messages. This is because the global state, at the time when V_{new} fails the acceptance test T2, satisfies the consistency property.

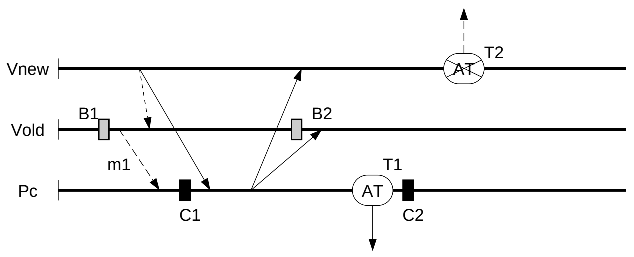

Figure 5: Roll Forward without Message Re-send or Suppression

Figure 6 illustrates the scenario in which V_{new} fails on acceptance T3 and V_{old} will subsequently 1) take over without rollback, and 2) re-send the suppressed message that passed acceptance test T2. Again, message logging at the sender's side assures recoverability.

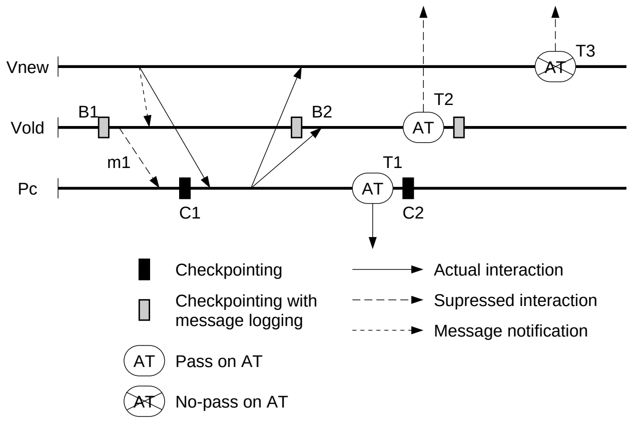

Figure 6: Roll Forward with Message Re-send

Figure 7 illustrates the scenario (at the guarded operation stage) in which V_{new} fails on acceptance T4. Since the global state that comprises the process states of V_{old} and P_c at the time of the failure satisfies the consistency property (as defined in [9]), V_{old} will subsequently 1) take over the control without rollback, and 2) suppress the message that is equivalent to the first message of V_{new} after version switching and when it executes to the point where the message is generated. Again, message sequence number enables the avoidance of redundant messages.

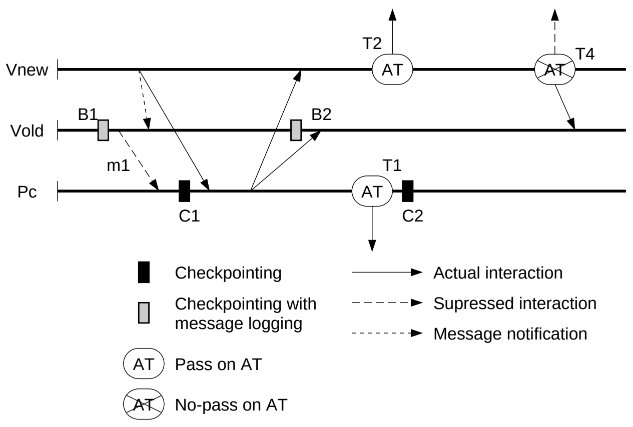

Figure 7: Roll Forward with Message Suppression

Figure 8 illustrates the scenario in which V_{new} fails on acceptance T3 and V_{old} rolls back to checkpoint B3, upon version switching, without sending out logged messages or suppressing any newly generated messages; and P_c rolls back to checkpoint C3. Note that B3 and C3 comprise the most recent recoverable consistent system state [10].

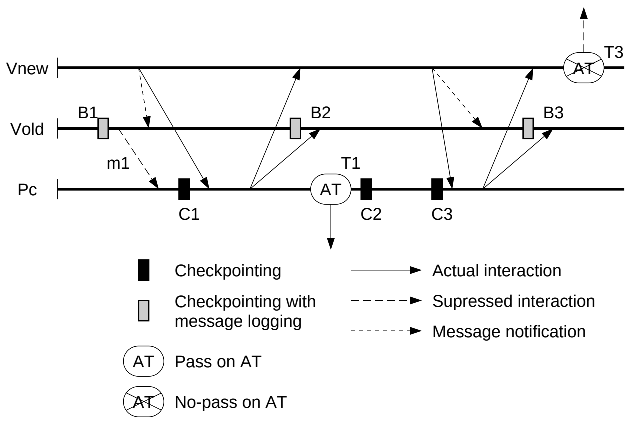

Figure 8: Rollback Recovery without Message Re-Send or Suppression

Figure 9 depicts the scenario (at the guarded operation stage) in which V_{new} fails on acceptance T2. Upon the version switching triggered by the failure,

V_{old} rolls back to checkpoint B4 and needs to re-send message m2, while P_c rolls back to checkpoint C3. In this case, B4 and C3 comprise the most recent recoverable consistent system state (as defined in [10]). And similar to the version switching case and roll-forward case, message logging assures recoverability.

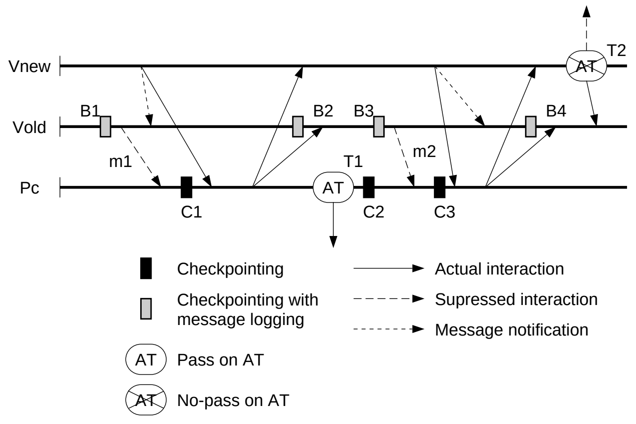

Figure 9: Rollback Recovery with Message Re-Send

Figure 10 describes the scenario in which V_{new} fails on acceptance T3 and V_{old} rolls back to checkpoint B3, and needs to suppress message m2 and the first message to the device, after version switching and when it executes to the point where the messages are generated; P_c rolls back to checkpoint C3. Here, B3 and C3 comprise the most recent recoverable consistent system state. And similar to the version switching case and roll-forward case, message sequence number plays a crucial role for the avoidance of redundant messages.

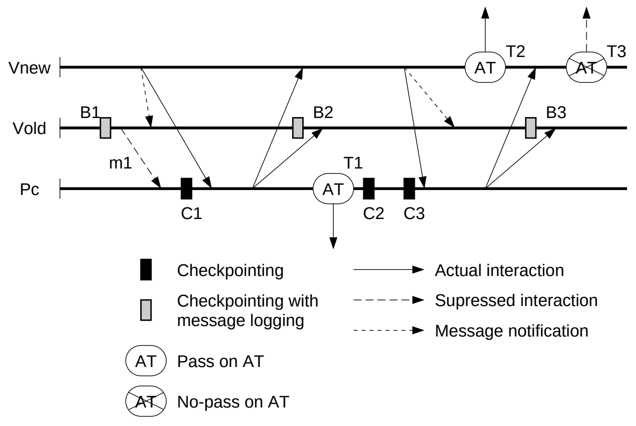

Figure 10: Rollback Recovery with Message Suppression

3 GSU Middleware

Figure 11 shows the relationships among the major components that comprise the GSU Middleware architecture, namely, how the invokable services, active agents and on-board tools collaborate to accomplish guarded software upgrading.

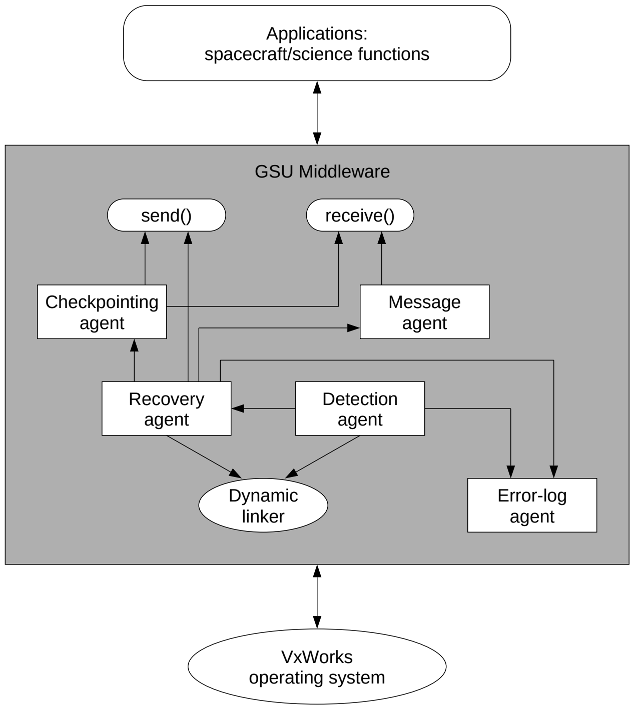

Figure 11: GSU Middleware Architecture

As mentioned earlier, invocations of the invokable services `send()` and `receive()` by an application program may result in different activities, depending upon the role of the invoking process and the execution environment, as described in Tables 1 and 2.

As shown in Figure 11, the invokable services are supported by a group of active agents, namely,

Message-log maintenance agent: Responsible for deleting the messages that are no longer necessary to remain in the message log of a process.

Checkpoint maintenance agent: Responsible for deleting the checkpoint that is no longer necessary to be maintained.

Error-detection agent: Responsible for carrying out acceptance test and raising the exception flag

Table 1: Invokable Services During On-Board Validation

	V_{old}	V_{new}	P_c
`send()`	Actually transmits a message and sends the message sequence number to V_{new} (message notification).	Suppresses the message and saves it in the message log.	Sends the message to both V_{old} and V_{new}.
`receive()`	Passes it to application directly.	Saves the message sequence number if it is a message notification from V_{old}; otherwise 1) takes a checkpoint and does message logging, and 2) passes it to application.	Takes a checkpoint before passing it to the application.

upon a failed acceptance test.

Error-recovery agent: Responsible for monitoring the error flags and making decision on recovery mechanisms upon a failed acceptance test, e.g., decisions on roll-forward or rollback, the rollback distance for each process, and whether message suppression or re-sending is required.

Error-log agent: Responsible for the maintenance of an on-board error log and classification of error events/symptoms (e.g., single or recurrent error type) for downlinking to the ground to facilitate statistical modeling and heuristic trend analysis.

The agents for checkpoint and message-log maintenance are important for cost-effective memory utilization and assuring only the information necessary for version switching or error recovery are maintained. Specifically, for the on-board validation stage during which V_{new} runs as a shadow and thus its error will not contaminate other processes, V_{new} needs only to keep its most recent checkpoint for the rollback recovery if an error is detected; while P_c needs only to keep its most recent checkpoint for keeping track of incoming messages and rejecting redundant messages after version switching. As to the guarded operation stage during which errors of the active V_{new} will contaminate other processes, when V_{new} or P_c passes an acceptance test, all the processes need only to keep their most recent checkpoints; when V_{old} passes an acceptance test, the previous checkpoints of its own and those of P_c taken prior to its most recent message received by V_{old} (that has proved to be correct via the acceptance test of V_{old}) can be deleted, excluding the most recent checkpoint.

4 Summary and Future Work

We have presented an approach to guarded software upgrading through scenario-based descriptions. Our objectives are to avoid or minimize the unavailability and performance loss of spacecraft/science functions due to software upgrading activities and due to system failure caused by residual faults in an upgraded version. Our approach emphasizes

1) the utilization of nondedicated or inherent system resource redundancies such as an earlier software version and a processor that otherwise would be idle in the mission phase during which software upgrading activities are conducted, and

2) the low-cost efficient error containment and protection mechanisms based on adaptation/extension of the enabling technologies such as checkpointing and message logging.

In addition to NASA's X2000 missions, the GSU methods could benefit a wide variety of commercial application domains. In particular, 1) the guarded software upgrading techniques will be useful for many other applications which are subject to frequent software upgrading and require high availability and/or safety, such as Internet services, transportation systems, airline reservation systems, telephone systems and medical systems, and 2) the methods of utilizing inherent, non-dedicated resource redundancies and

Table 2: Invokable Services During Guarded Operation

	V_{old}	V_{new}	P_c
`send()`	Suppresses the message and saves it in the message log; if the message destination is a device, performs acceptance test before logging the message.	Actually transmits a message and sends the message sequence number to V_{old}; if the message destination is a device, performs acceptance test before sending the message.	If the destination is V_{new}, sends the message to both V_{old} and V_{new}; if the message destination is a device, performs acceptance test before sending the message.
`receive()`	If it is a message notification from V_{new}, save the message sequence number; otherwise 1) takes a checkpoint and does message logging, and 2) passes it to application.	Passes it to application directly.	Takes a checkpoint before passing it to the application.

our low-cost flexible error containment and recovery algorithms will enable the transferring of the state-of-the-art fault tolerance techniques from research domain to real applications. Currently, we are formalizing the error containment and protection methods and conducting formal algorithm verifications. We also plan to carry out a series of case studies based on analytic methods and testbed experiments to investigate the effectiveness of the GSU methodology for a variety of software applications in space systems.

References

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

[2] A. T. Tai and L. Alkalai, "On-board maintenance for long-life systems," in *Proceedings of the IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET'98)*, (Richardson, TX), pp. 69–74, Apr. 1998.

[3] R. Baalke, Office of the Flight Operations Manager, "Mars Pathfinder update," Mars Pathfinder Weekly Status Report, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, June 1997.

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

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

[6] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[7] K. S. Tso and A. Avižienis, "Community error recovery in N-version software: A design study with experimentation," in *Digest of the 17th Annual International Symposium on Fault-Tolerant Computing*, (Pittsburgh, PA), pp. 127–133, July 1987.

[8] Y. M. Wang *et al.*, "Checkpointing and its applications," in *Digest of the 25th Annual International Symposium on Fault-Tolerant Computing*, (Pasadena, CA), pp. 22–31, June 1995.

[9] 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), Sept. 1998.

[10] 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.