Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine

Doron Drusinsky
Time Rover, Inc.
Cupertino, CA, USA
www.time-rover.com

Garth Watney
NASA JPL
Pasadena, CA, USA
Garth.J.Watney@jpl.nasa.gov

Abstract

Run-time monitoring is a lightweight verification method whereby the correctness of a program’s execution is verified at run-time using executable specifications. This paper describes the verification of the fault protection engine of the Deep-Impact spacecraft flight software using a temporal logic based run-time monitoring tool.

1 Introduction

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. Pnueli [11] suggested using LTL for reasoning about concurrent programs. Since then, several researchers have used LTL to state and measure correctness of concurrent programs, protocols, and hardware (e.g., [10]). LTL is an extension of propositional logic where, in addition to the well-known propositional logic operators, there are four future-time operators (Eventually, Always, Until, Next) and four, dual, past-time operators. Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real-time systems [1]. MTL extends LTL by supporting the specification of relative time and real-time constraints. All four LTL future time operators can be constrained by relative time (cycles) and real-time constraints specifying the duration of the temporal operator.

Runtime Execution Monitoring (REM) is a class of methods of tracking the temporal behavior of an underlying application. REM methods range from simple print statement logging methods to run-time tracking of complete formal requirements (e.g., written in LTL/MTL) for verification purposes. Indeed, first applications of REM were verification oriented where REM was used to track how formal specification requirements are conformed to by the actual executing system. Temporal Rover [2] and DBRover [6] are such systems. Recent adaptations of on-line REM methods enable run time monitoring for non-verification purposes such as temporal business rule checking and temporal security rule checking [5], as well as for model checking [7] and rapid prototyping [6].

In April 2002 a three-month task was funded to exchange technologies and explore methods for the infusion of run-time verification into actual mission flight software at JPL. This task was collaboration between JPL, Ames Research Center and Time-Rover. The specific goals of this task were to generate and test Flight Software Correctness Properties in the Flight Software domain, and to capture a process for the development and testing of these Correctness Properties. For this task we used REM-based verification techniques. This paper described the problem, methods, and tools used in the process.

2 Run-time Monitoring with the DBRover

The DBRover is a temporal logic editor, simulator, and run-time monitor. Development using the DBRover is performed using the following Graphical User Interface (GUI) tabs:

1. Editor, for graphically editing temporal logic rules (assertions). The tool supports future time and past time linear temporal logic, including real-time constraints (MTL) and time-series constraints [4]. Support is also provided for special counting operators described in [2].

2. Builder, for launching an automatic build process after which rules are run-time monitored, as described in the monitor tab. The DBRover builder builds rules in sets
consisting of one or more rules; such a rule-set is built as a dynamically linked library that hooks onto the DBRover GUI. In addition, the DBRover generates a small code snippets which is used to instrument the target; this snippet is responsible for sending Boolean information from the target, as described below. The DBRover also supports a No Snippets protocol with which the target’s instrumentation is perfumed once and does not change every time the rules change.

3. **Monitor**, once a rule-set is built, the DBRover listens to messages from target applications pertaining to these rules. Messages contain Boolean information about basic propositions in the temporal logic rules. For example, the rule \( \text{Always } x>0 \implies \text{Eventually } y>0 \) consists of two basic propositions, \( x>0 \) and \( y>0 \); the target application is responsible for sending those two bits of information to the DBRover every cycle. In fact, a cycle is defined by the arrival of a new set of basic proposition values. Additional information sometimes sent from the target is real-time clock readings and time-series values.

4. **Administration**, for controlling how information about the success and failure of a rule are delivered to the end user. For example, the end user can decide to be notified upon the success of one rule via email while for another rule (s)he decides to execute a custom program every time it fails. Note that as described in [2], the DBRover and Temporal Rover actually provide two Boolean results per rule, the first describing whether the rule succeeded or failed so far, and the other describing whether this success or fail status might potentially change in the future, i.e., whether the first Boolean result is final or not.

5. **Simulator**, for simulating rules before they get built for monitoring. The simulator is capable of simulating multiple scenarios per rule, including real-time and time-series simulation for rules that contain such constraints. Fig. 1 illustrates the simulator tab for a rule with real-time and time-series constraints. Simulation scenarios can be exported and imported to facilitate group level rule development and for later use within a test suite. Also, results displayed in the monitor tab can be copies and pasted into the simulation tab; this enables off-line analysis of rules using run time generated scenarios.

![DBRover GUI](image)

**Figure 1. The DBRover Simulator Tab**
themselves. The DBRover then visualizes those potential deadlocks as Message Sequence Chart (MSC) diagrams.

7. TLCharts, supports a new hybrid of temporal logic and Harel statecharts, described in section 4.

The DBRover is a remote monitor capable of monitoring applications in most design phases, such as (i) prototyping, in cooperation with prototyping tools such as Matlab and CAPS [6], (ii) cross platform simulation, the level of the work described in this paper, and (iii) on an embedded target such as VxWorks™. Communication between the target and the DBRover is performed via sockets, serial-port, or http.

3 Deep Impact Fault Protection Verification

3.1 Deep-Impact Mission and Fault Protection

The Deep-Impact spacecraft will make a football-stadium-sized crater 15 stories deep in the comet Tempel 1. The spacecraft consists of two autonomously controlled vehicles - the Flyby Spacecraft and the Impactor, which separate 24 hours before encounter. The Impactor is a battery-powered spacecraft with the capability to navigate towards the comet for 24 hours. As the Impactor approaches the comet it will transmit images back to the Flyby Spacecraft seconds before impact. The Flyby Spacecraft will fly within 300 miles from the impacted comet recording the impact event. The launch date is set for December 2004 with an encounter date of July 2005.

The Deep-Impact Flight Software, written in C++, was chosen as a good candidate for this task for the following reasons: 1) The runtime environment was actively maintained at JPL and easily available. 2) Some key modules of interest that pertain to automation had been developed at JPL – notably the Fault Protection (FP) subsystem and Auto-Navigation subsystem. The behavior of the Fault Protection subsystem in particular was well understood by the runtime verification task lead at JPL. 3) The source code and build procedure of the Fault Protection subsystem was easily accessible in order to insert monitor points into the application code and perform rebuilds. All of the above mentioned reasons were found necessary for the successful completion of this task.

3.2 Fault Protection Engine Module

The Fault Protection Engine is a reusable, non-mission specific, subsystem within the Deep-Impact. This component is responsible for the management of all system level fault identification and recoveries. The Engine provides the capability for the spacecraft to recover from a single fault and continue its mission. Multiple faults are handled sequentially, where only one response is active at a time. At system startup, calls are made to the Engine to register symptoms, faults, recovery scripts and their corresponding mappings. At runtime, the Engine is responsible for the orderly handling of incoming symptoms and the execution of appropriate recovery scripts – called responses. All responses are classified into a 2 level priority scheme. The Engine has the capability to interrupt the execution of a low priority response with a high priority response. Responses can in-turn invoke sub-responses. Responses are queued on 2 separate queues and managed in an orderly manner. The Engine maintains its internal state and the internal state of each registered Symptom, Fault and Response (see table 1). Responses may vary from a single command (reset a hardware device) to a long running sequence (orient the spacecraft to a safe attitude). The Engine ensures that responses are handled in an orderly fashion and are not run unnecessarily (triggered by an overly sensitive monitor). After each response is run through to completion, the Engine clears the fault and sends a cleanup signal back to the offending monitor.

<table>
<thead>
<tr>
<th>Engine</th>
<th>Id:le</th>
<th>Run_Response_NoWaypoint</th>
<th>Run_Response_Waypoint</th>
<th>Run_Interrupting_Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symptom</td>
<td>Green</td>
<td>Red</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault</td>
<td>Green_Queue</td>
<td>Green_Unqueue</td>
<td>Red</td>
<td>Disabled</td>
</tr>
<tr>
<td>Response</td>
<td>Id:le</td>
<td>Running</td>
<td>Disabled</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Fault protection engine state table

Other missions that have used similar fault protection capabilities include Galileo, Mars Pathfinder and DeepSpace 1. Deep-Impact continues with this legacy but has incorpo-
cation and monitoring tool – DBRover. At the time that this task was started, the Engine had completed its traditional module testing and was in the process of being integrated with the rest of the Flight Software. The Engine was in a moderate state of maturity with no known problems. We still believed however that there was a good possibility of uncovering latent bugs in the software since there are many aspects of the application that lend themselves well to a runtime verification approach. During runtime, the developer/tester may be unaware of inconsistencies within the internal state of the Engine. Faults may be locked into a state where they are unable to trigger a response. Due to the nature of the application, there are hundreds of possible symptoms that may be reported to the Engine and dozens of possible faults that can trigger responses. It would be difficult for a test engineer to be constantly checking internal state consistency. Using the runtime verification approach, the DBRover monitor is run in parallel with the application and alerts the user to any violation of pre-defined correctness properties.

3.3 Run-Time Verification Process

Besides simply finding defects in the application code, this task was primarily interested in establishing a process for generating and applying Correctness Properties to mission flight software. The inputs we used were 1) flight software requirements, 2) software design documentation, and 3) source code. Correctness properties had to be formally defined in Linear Temporal Logic (LTL) and/or Metric Temporal Logic (MTL) with the aid of the DBRover tool and stored in the tool’s database. One of the initial difficulties encountered was how to formally define a correctness property from a requirement. Most software requirements do not lend themselves easily to the generation of correctness properties. Requirements are typically general enough as to not enforce a particular implementation. Some typical FP requirements are stated as: Recovery sequences shall be initiated sequentially and shall not be initiated unnecessarily and Every initiated recovery sequence shall execute through to completion. The problems with generating correctness properties were 2-fold: 1) How to formally translate the requirement to LTL/MTL, and 2) How to implement the LTL/MTL statement into real events and variables applicable to the software under test. To solve these problems, we had to dig into the software design documentation to gain the knowledge of how the requirements were implemented and then dig into the source code to find variables and events that could be monitored by the DBRover tool. By using this process we were able to generate correctness properties, run the application and perform monitor checking through the DBRover tool. But many of these monitor points were testing flight properties that were obvious and not particularly interesting. We were interested in checking complex interactions that were non-obvious to the developers and testers, were difficult to debug and were potentially catastrophic. Improved methods of generating meaningful correctness properties have been addressed in section 4.

3.4 Findings

To generate meaningful correctness properties, we tried some reverse engineering techniques. A detailed changelog of the source code had been maintained during module and subsystem testing. By using this changelog we could track defects to a particular software version in the repository. A subset of 6 defects were identified as non-obvious, difficult to debug and potentially catastrophic. For each identified defect, the question, what correctness property is violated? was asked. The answer to each question led to a description of a correctness property in an English sentence. This sentence could then be translated into formal LTL/MTL with the aid of the DBRover tool. The software pertaining to each identified defect was resurrected and a build of the application was made with the appropriate monitor points inserted. This build was then run through its testing exercises alongside the DBRover monitor tool. For each of these defect builds, it was verified that DBRover identified the appropriate violation of a Correctness Property.

The latest Engine software to-date had officially closed out all these 6 identified defects. This latest software was then monitored for each of the above-mentioned correctness properties. The Engine was put through its testing with the expectation that no violations of correctness properties would be raised. This was true for 5 of the 6 correctness properties. One of the correctness properties that were being monitored raised a violation during the test. This unexpected result had uncovered a case where the software had been “fixed” so that the immediate symptom of the problem was eliminated, but the underlying correctness property was still being violated (i.e., a “kludge” fix). This particular violated correctness property left the Engine in a state where it appeared to be functioning correctly but in actuality was in an undesirable state. This undesirable state had the potential to exhibit itself by failing to invoke a needed response script, which in turn could be catastrophic for the Deep Impact mission.

Identifying correctness properties by using the defect logs had proved to be a very fruitful exercise. These correctness properties not only found a software defect still lurking in the code, but now there was the potential to extend each of these correctness properties in new ways to find still more hidden and subtle defects. As this task had come to an end, the identified process could now be utilized and extended by the Deep Impact software development team.
4 Recent Developments: Monitoring TLCharts

The Deep Impact verification task was performed in the summer of 2002. While the feedback was positive, a concern raised by some users was the difficulty of developing complex assertions in temporal logic. As a solution, the hybrid language of TLCharts was developed, combining Harel statecharts and temporal logic [3].

The following infusion pump example illustrates the language of TLCharts and will be used to contrast TLCharts with temporal logic. The infusion pump consists four conditions: infusionBegin, infusionEnd, keyPress and alarmSound. The informal requirement is:

Between every infusionBegin and infusionEnd a keyPress must be repeatedly sensed within 2-minute intervals, or else an alarm must sound within 10 seconds; also, once alarm sounds according to this specification then the assertion has succeeded.

The following MTL assertion was developed by an engineer with experience using LTL and MTL:

\[ \text{Always}(\text{infusionBegin} \implies ((\text{infusionBegin Or keyPressed}) \implies (\text{Always}_{≤}120 \text{Not alarmSound}) \text{And} (\text{Next Eventually}_{≤}120 \text{keyPressed}) \text{Or} (\text{Not keyPressed Until}_{[120,130]} \text{alarmSound}))) \text{ Until infusionEnd}) \]

This assertion suffers from several deficiencies:

1. The assertion is arguably non-trivial while the natural language requirement is straightforward. For example, the term \text{infusionBegin Or keyPressed} is confusing; the author of this paper actually used a simpler keyPressed term in an earlier version of this rule, a version that is incorrect because after infusionBegin unless keyPressed actually happens the assertion will not require it to happen.

2. The assertion might fail under the following scenarios, assuming the LTL-model cycle time is one second, i.e., the assertion is evaluated every second.

   a) An interval of 122 seconds between two consecutive keyPress events followed by an alarm sounding 1 second later, followed by no keyPress or alarm for 140 seconds. The assertion fails though the requirement is that following the first alarm the assertion must succeed.

   b) An interval of 122 seconds between two consecutive keyPress events followed by an alarm sounding 4 seconds later. The assertion fails because LTL’s \(\rho \text{Until} \varphi\) requires \(\rho\) to repeatedly succeed until \(\varphi\) succeeds, namely Not keyPressed must be true until the alarm.

3. An intuitive expectation is that an infusionEnd event will terminate the need for a flow of keyPress events. However, if keyPress occurs at time \(t\) and infusionEnd at time \(t+20\) then one additional keyPress will still be required after time \(t+20\). In other words, there is no simple way to explicitly truncate the requirement once infusionEnd other than to conjunct infusionEnd with the inner parts of the rule. Separate research on LTL with truncated paths has been published in [8].

The TLChart specification of the infusion pump requirement is illustrated in Fig. 2. The TLChart differs from a conventional Harel statechart in two primary ways:

1. Some transition are annotated with temporal logic conditions; e.g., the transition labeled \(\text{Always}_{≤}120 \text{!keyPress}\).

2. A TLChart is inherently non-deterministic. For example, in Fig. 2 non-determinism exists in two forms:

   a) The traditional form where two or more conflicting state changes might coincide, such as the case when both the keyPress and infusionEnd conditions are true while in state Wait.

   b) Whenever a transition annotated by temporal logic condition is enabled, such as the case during a computation which visits state Wait: a transition annotated with the temporal logic condition \(\text{Always}_{≤}120 \text{!keyPress}\) is enabled, namely, it is a candidate for traversal. From the TLCharts formal semantics described above it follows that this transition does not consume any inputs from the input sequence, i.e., it is a pure constraint. Note that additional, alternative computations leading out of state Wait do exist, such as the computation which consumes the input symbol alarmSound and changes state to the Alarm state. Such transitions are similar to \(\varepsilon\)-transitions in a non-deterministic automaton, except that they are guarded by temporal logic guards. Temporal logic guards can also appear in conjunction with conventional statechart transition labels (e.g., events) in which case the usual 1-cycle delay is imposed.
The TLC hart of Fig. 2 uses the following priority-based final state marking scheme: the Error state is a rejecting state with the highest priority (1) while all other states are accepting and have a lower priority. Using this semantics the TLCart rejects an input sequence if there exists a computation that ends in the Error state.

In addition to their visual and intuitive appeal TLCharts are useful in the following context. While the UML is widely acceptable, formal LTL and MTL specifications are not widely used beyond a limited circle of researcher and limited segments of one industry (hardware verification). Hence, it is quite common to expect organizations to have design some statecharts available when attempting to formulate LTL/MTL specifications. TLCharts can serve as a vehicle for a simple and safe upward migration path from design statecharts to formal specifications.

TLCharts solve the truncation problem in a visual manner. In Fig. 2, the Init state has priority 1, the highest priority, the Error state has priority 2, and all other states have the lowest priority 3. Hence given an input string s, if any computation ends in Init state then the TLCart accepts s, else if any computation ends in the Error state then the TLCart rejects s, otherwise the TLCart accepts s. In other words, once the computation reaches the Init state following an infusionEnd event, the input string is accepted. To assure that reaching the Init state does not forgive prior errors, e.g., a prior visitation to the Error(2) state following a premature alarm, we mark the Init state as priority 1^f, the f subscript denotes future, indicating that a success generated when visiting this state does not forgive (override) a previously generated error; however, Init having priority 1^f (i.e., a special 1) still implies that no future, lower-priority, Error can override the Init state.

The DBRover TLCharts tab supports TLChart verification using the following technique. A TLCharts converted into a set of related temporal logic reachability rules that assert about reaching individual states in the TLChart. For example, the temporal rule Eventually(infusionBegin And Eventually ≤120 alarmSound) describes the reachability rule for the Error state. Each such rule has an associated priority level describing the priority of the state being reached. During run-time, the DBRover monitors all reachability rules and aggregates the monitored results into a single statement for the entire TLChart.

5 Conclusions

Runtime Verification is a powerful but difficult process to implement. Some of the discovered strengths of this process are the following:

1) Automatically removes defects from the application code that slip through conventional testing; automation is particularly important for repeated (e.g., regression) testing of application code.
2) A practical debugging tool for the software developer at the coding and module testing level.
3) Enables the focus to stay on internal state consistency, not just the symptoms of the perceived problem.
4) Builds internal verification within the software application. This built-in verification extends the traditional
role of the C/C++ “Assert” statement by specifying powerful temporal logic properties, including real-time constraints.

5) Enables verification during the three primary development phases: prototyping (e.g., using Matlab simulation), cross platform simulation, and on the embedded target under true real-time constraints.

To alleviate some of the difficulty in this process, it would be best to incrementally build a library of correctness properties through the phases of software development – requirements generation to design to coding and testing. Once these correctness properties are applied to the application code, they should remain in the code to form a built-in verification capability guarding against the introduction of any software changes that may violate the stated property. One other improvement is to enable correctness property specification using a visual and familiar notation, as suggested in section 4.

The selected tool for runtime verification is very important for the success of this process. The tool will need to provide the following capabilities:

1) Aid in the generation and debugging of LTL properties
2) Aid in the instrumentation of the application code through auto code generation of monitor points and communication to the monitor tool without impacting the timing of the application code.
3) Interface with the user to start/stop monitoring and to notify the user of the status of each monitored Correctness Property.

Continual improvement in the quality of runtime verification tools should eventually enable the process of runtime verification and monitoring to become part of the overall software development process for spacecraft flight software development.

References