In Proceedings of the International Conference on Performance, Computing and Communications, April 2001, pp.145-150, IEEE.

Embedded Software for a Space Interferometry System: Automated Analysis of a Software Product Line Architecture

Gerald C. Gannod*†‡
Department of Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
(480) 727-4475
gannod@asu.edu

Robyn R. Lutz§
Jet Propulsion Laboratory
4800 Oak Grove Drive
M/S 125-233
Pasadena, CA 91109-8099
(515) 294-3654
rlutz@cs.iastate.edu

Marian Cantu
Department of Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
(480) 441-0847
Marian.Cantu@motorola.com

Abstract

This paper describes the analysis of the embedded software for an interferometry system using model checking as a means for achieving various analysis goals. The contribution of this paper is to demonstrate how the use of lightweight formal methods can be applied to software for an embedded system via analysis of the behavior of a software architecture.

1 Introduction

Embedded systems are computer-based systems that are used to control, monitor, and facilitate the operation of devices or other machines. The fact that these systems are "embedded" indicates that they are an integral part of an overall system. For instance, many of the popular hand-held devices that permeate popular culture such as digital cell phones and personal digital assistants are examples of devices that make use of embedded systems.

In the space exploration domain, there has been an effort to make use of embedded systems for several years to control robotic spacecraft. Currently, there is an interferometer product line (e.g., a family of spacecraft and ground-based systems) that makes extensive use of embedded systems, where the systems in the product line share both common hardware and software architectures.

This paper describes the analysis of the embedded software for an interferometry system using model checking as a means for achieving various analysis goals. The contribution of this paper is to demonstrate how the use of lightweight formal methods can be applied to software for an embedded system via analysis of the behavior of a software architecture.

The remainder of this paper is organized as follows. Section 2 provides background relating to software architecture, product lines, and the interferometer application. Section 3 describes the case study, including the overall baseline architecture of the interferometer product line and the subsequent analysis of a portion of the embedded software. Finally, Section 4 offers concluding remarks and suggests some future research.

2 Background

This section describes background material in the areas of software architectures, software product lines, and interferometry.

2.1 Software Architectures

A *software architecture* describes the overall organization of a software system in terms of its constituent elements, including computational units and their interrelationships [7]. In general, an architecture is defined as a *configuration* of *components* and *connectors*. A component is an encapsulation of a computational unit and has an interface that specifies the capabilities that the component can provide. Connectors, on the

*This research was performed while this author was a visiting researcher at the Jet Propulsion Laboratory.

†This author supported in part by a NASA/ASEE Summer Faculty Fellowship.

‡Contact Author.

§Mailing address: Dept. of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA 50011-1041.

0-7803-7001-5/01/$10.00 © 2001 IEEE

other hand, encapsulate the ways that components interact. A configuration of components interconnected with connectors determines the topology of the architecture and provides both a structural and semantic view of a system, where the semantics are provided by the individual specifications of the components and connectors.

2.2 Product Lines

Bass, Clements, and Kazman define a *software product line* as "a collection of systems sharing a managed set of features constructed from a common set of core software assets" [2]. These assets typically include a base architecture and a set of shared software components. The software architecture for the product line displays the commonality that the systems share and provides the mechanisms for variability among the products. The systems in the product line are referred to as *members* or *derivatives* of the baseline architecture or architectural style.

2.3 Interferometers

Interferometers will be used to explore the origins of stars and galaxies and to search for Earth-like planets around distant stars. An interferometer combines the starlight it collects from telescopes in such a way that the light "interferes" or interacts to increase the intensity and increase the precision of the observation. The product line of interest in this work is a set of interferometer projects under development by NASA's Jet Propulsion Laboratory. An interferometer, in this context, is a collection of telescopes that act together as a single, very powerful instrument. Three spaceborne interferometers are either under development or planned for launch in the next eleven years, with additional formation-flying interferometers envisioned for subsequent years [5, 6]. Two ground-based interferometers in the product line are currently operational, with at least two more planned.

Extensive documentation of the requirements and design for these software components, as well as C code for the component prototypes, were available for our analyses. In addition, we used whatever project-unique documentation was available. Predictably, more documentation exists for projects farther along in their development. System descriptions are available for all the missions; software requirements and design documents are still high-level and informal for later missions; and code is not yet available for any of the spaceborne interferometers.

3 Approach and Analysis

Figure 1 shows the topology of the architecture that served as the basis for all analysis, both manual and automated. The architecture, as mentioned earlier, was derived using information recovered from documentation, source code, and developer interviews. In the diagram, hardware components are shown as shaded and round rectangles while the software components are shown as sharp rectangles. The connectors, represented by lines between components, depict the relationships between components in the architecture. This particular diagram represents the software that exists within an "arm" of an interferometer, where a standard interferometer has two arms.

Among the components shared by the interferometer systems and discussed in this paper are the Delay Line, the Fringe Tracker, and the Internal Metrology. The Delay Line component compensates for the difference in time between the arrival of starlight at the separate mirrors. The Fringe Tracker component provides constant feedback to the Delay Line regarding needed adjustments to maintain peak intensity of the fringe (patterns of light and dark bands produced by interference of the light). The Internal Metrology component provides input to the Delay Line regarding small changes in distances among parts of the interferometer that must be included in its calculations.

Behavioral parts of interest were identified as meriting more rigorous analysis based on previous close reading of the systems' documentation and prior architectural analysis. The intent in this study was not systematic verification of all common behaviors but added assurance that some specific architectural features, flagged by the analysts as possibly vulnerable, were correct. The approach that we used to analyze the system was to utilize the Spin Model Checker [4] to specify the algorithmic and communication behaviors. The Spin is a model checker that has been used for verifying the behavior of a wide variety of hardware and software applications. Promela, the input language for Spin, is based on Dijkstra's guarded command language as well as CSP. In addition, we used the Wright ADL [1] for preliminary specifications that were later translated into Promela.

While our focus was primarily upon the software of the interferometer system, a corresponding hardware and hardware/software analysis is both possible and part of our future investigations. A key element of the interferometer architecture was the use of the "Target Buffer" connector. This connector, both in the design and in the implementation, is a non-locking buffer used to communicate target trajectories to the Delay Line component by several other components. A target is a specified position for the Delay Line controller to achieve. The target trajectories from multiple sources are combined by the Delay Line's target generation

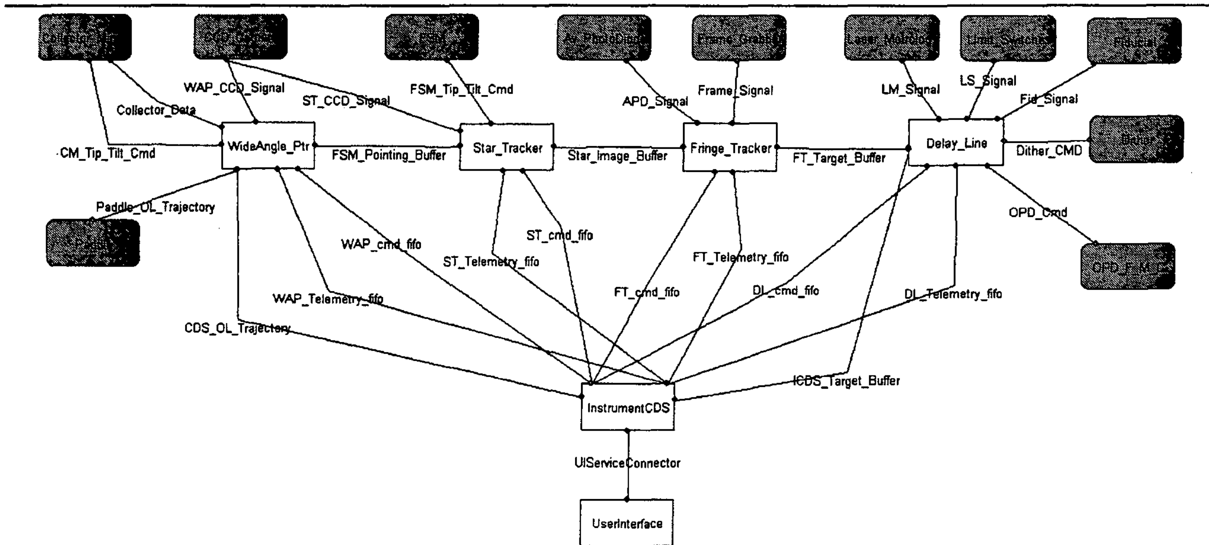

Figure 1. Interferometer Software Architecture

software to calculate a target. The Target Buffer connector was viewed as a possible concern, especially in light of the non-locking feature. It was determined that behavior involving this connector should be formally specified in order to study its impact on the system.

There are several components that are either directly or indirectly impacted by the non-locking nature of the Target Buffer connector: Target Sources, a Command Controller, and a Target Generator component. The Target Generator uses the values written to the Target Buffer by various Target Sources to compute a target position for the interferometer. The Command Controller provides control for the computation by enabling or disabling the Target Sources. Target Sources write a timestamped value to the Target Buffer, with the timestamp determining a time that the target value becomes valid.

The Target Generator uses the following four-step sequence for calculating the target position:

1. Promote waiting targets to active status if the current time is greater than or equal to the timestamp
2. Read new targets from enabled target sources
3. Pend (assign to *wait status*) or activate new targets based on timestamps
4. Compute the total target

The Wright specification of the interaction between the Target Generator and the potential sources of data that are written to the Target Buffer is shown in Figure 2. The Source specification models the fact that a source internally decides whether or not to write a new value to the Target Buffer. Finally, the Target Generator specification models the target-position algorithm described above.

```
Style TargetComputation
Connector TargetBuffer
   Role Writer = writetarget!x -> Writer |~| Tick
   Role Reader = readtarget?x -> Reader |~| Tick
   Glue = Writer.writetarget!x -> Glue []
        Reader.readtarget!x -> Glue [] Tick
Component Source
   Port CDSCommand = enable -> CDSCommand |~|
                disable -> CDSCommand |~| Tick
   Port DLTarget = write!x -> DLTarget |~| Tick
   Computation = (CDSCommand.enable -> Generate) []
      (CDSCommand.disable -> Computation) [] Tick
   where { Generate = DLTarget.write!y -> Generate []
           Generate [] Tick }
Component TargetGenerator
   Port Input = readtarget?x -> TargetBuffer |~| Tick
   Computation = (_promote ->
               Input.read_target?x ->
               _pend_or_activate ->
               _compute -> Computation [] Tick )
end Style

Configuration TargetComputationInstance
Instances
   tb1 : TargetBuffer
   src1 : Source
   dl : TargetGenerator
Attachments
   src1.DLTarget as tb1.Writer
   dl.Input as tb1.Reader
End Configuration
```

Figure 2. Subset of the Wright Specification

From the Wright specification, we constructed a Promela specification, portions of which are found in Figures 4 and 5. A sample of a message sequence chart

for this model is shown in Figure 3. In the diagram, the vertical bars represent processes, messages received or sent by a process are shown as arrows between the bars, and the relative time between sending and receiving the various messages is shown by the length and angle of the arrows. Here a controlling process (icds) sends message to a pair of sources, corresponding to enabling or disabling commands. Since the delay_line process (e.g., the target generator) receives data only through shared target buffers, only messages indicating which sources are enabled are sent. The message and timestamp processes are merely function calls that are used to randomly generate data (message) and timestamps.

It was our intention to use the model of the target generation process to determine whether or not the following situations could occur.

> **Data From Disabled Sources.** Is there a potential for calculating the target position by using data from sources that are currently disabled?
>
> **Best Data from Enabled Sources.** Is there a potential for calculating a target position by using data that is less current than data currently in the target buffer?

```
proctype source_1 (chan cds){
   chan cmd;
   chan ts = [1] of { int };
   chan msg = [1] of { int };
   int active_or_inactive;
   cds?cmd;
   cmd?active_or_inactive;
   do :: (msgs_generated < max_msgs) &&
         (active_or_inactive == true) ->
         if :: run message(msg);
               msg?o_tb1;
               run timestamp(ts);
               ts?s1_ap;
               msgs_generated = msgs_generated + 1;
            :: skip;
         fi;
      :: (done != true) -> cmd?active_or_inactive;
      :: (done == true) -> break;
   od
}
```

Figure 4. Promela Specification of Target Source

In the first case, we were interested in determining whether or not it was possible to generate a target position by using data from inactive sources. In essence, a target position input can be read by the Target Generator, pended due to the timestamp (e.g., the timestamp indicates that the target value is not to be used until

```
proctype target_generator (chan valid)
{
   int v, active_sum;
   int prev1 = s1;
   int prev2 = s2;

end_tg:

   do
   :: (msgs_generated < max_msgs) ->

   /* "activation" of pended targets achieved by
         maintaining previous value of s1 or s2 */
      prev1 = s1; prev2 = s2;

   /* read new targets from active target sources */
      valid?v;
      if
      :: (v == NONE) -> active_sum = 0;
      :: (v == S1_ONLY) -> s1 = o_tb1; active_sum = s1;
      :: (v == S2_ONLY) -> s2 = o_tb2; active_sum = s2;
      :: (v == BOTH) -> s1 = o_tb1; s2 = o_tb2;
                        active_sum = s1 + s2;
      fi;

   /* if either of the following doesn't hold, then
      less current data is being used for the target */
      if
      :: (s1_ap < now) -> assert(prev1 == s1);
      :: (s2_ap < now) -> assert(prev2 == s2);
      fi;

   /* check if pended or not */
      if
      :: (v > NONE) ->
         if
         :: ((s1_ap <= now) && (s2_ap <= now)) ->
            sum = s1 + s2;
            assert( active_sum == sum );
            /* if not, then data is taken from
               inactive sources */
         :: ((s1_ap <= now) && (s2_ap > now)) ->
            sum = s1;
         :: ((s1_ap > now) && (s2_ap <= now)) ->
            sum = s2;
         :: ((s1_ap > now) && (s2_ap > now)) ->
            skip;
         fi;
      :: (v == NONE) -> skip;
      fi;

   /* compute target */
      printf("MSC: sum = %d\n", sum);
      printf("MSC: active_sum = %d\n", active_sum);

      /* reset sum */
      s1_ap = now; s2_ap = now;
      sum = 0; active_sum = 0;
   :: (msgs_generated >= max_msgs) -> break;
   od;
   done = 1;
}
```

Figure 5. Promela Specification of Target Generator

Figure 3. Message Sequence Chart for Target Generation

some time in the future), and subsequently promoted into use when the timestamp matches (or precedes) the current time. The potential inconsistency occurs during the time that the target is pended and is caused by the fact that a source can be disabled during this waiting period. To verify the existence of this behavior within the model, we used an assertion that compared the value of the target as computed from the currently active sources and the sum computed from the values stored in the target buffer. If, at the time that values from multiple target buffers are being used to compute a target results in a value that is not identical to the sum that can be computed from the currently active sources, then the data is inconsistent.

Within the specification in Figure 5, the statement `assert( active_sum == sum )`, is used to verify this scenario. Figure 6 shows the output of the Spin model checker for the corresponding safety assertion. Specifically, the figure shows that the assertion failed as well as statistics on the verification run.

The second case involves the following situation. As before, a target from a source is read, potentially pended, and eventually promoted. Because of the sequencing of events, a new target value from the source can overwrite the recently promoted target and, based on the timestamp, be valid for immediate use. Again we used a safety assertion to verify the existence of this behavior, this time checking to see whether recently promoted data has been overwritten with older data.

In order to determine whether these cases were also present in the code, we examined source files and were able to verify that the situations, as documented and as specified with Wright, did in fact exist in an early, pre-flight version of the source code.

In each of these cases, the use of a non-locking buffer coupled with the target-generator algorithm provided the potential for intermittent values that are inconsistent with the desired and current target. The interferometry project engineers confirmed that the Spin model checker accurately modeled the software behavior in both situations. In the first case, a target from a currently disabled target source may still be activated. In the second case, a newly received target with a less-current timestamp can overwrite an active target. However, in neither case is the software behavior contrary to intent, given the underlying assumptions about the operational use of the software.

4 Conclusion

The technique applied in this paper is part of a larger approach that we have developed for software product lines that includes both manual and automated analysis [3]. Here we have demonstrated how the use of a lightweight formal method can assist in the analysis of embedded software in order to determine correctness. Specifically, we showed how the use

```
pan: assertion violated (active_sum==sum) (at depth 411)
pan: wrote pan_in.trail
(Spin Version 3.2.3 -- 1 August 1998)
Warning: Search not completed
+ Partial Order Reduction
+ Compression

Full statespace search for:
never-claim         - (not selected)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid endstates - (disabled by -E flag)

State-vector 164 byte, depth reached 434, errors: 1
  891047 states, stored
1.2657e+06 states, matched
2.15674e+06 transitions (= stored+matched)
       3 atomic steps
hash conflicts: 1.13363e+06 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
156.824 equivalent memory usage for states
        (stored*(State-vector + overhead))
19.440 actual memory usage for states
        (compression: 12.40%)
State-vector as stored =
        10 byte + 12 byte overhead
2.097  memory used for hash-table (-w19)
0.240  memory used for DFS stack (-m10000)
21.896 total actual memory usage

nr of templates: [ globals procs chans ]
collapse counts: [ 18038 4 4 13 13 156 14 ]
0.83user 33.51system 0:34.33elapsed 100%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (108major+5316minor)pagefaults 0swaps
```

Figure 6. Spin Output of Safety Verification for the Disabled Source scenario

of a formal model can be used to reveal issues that may not be accessible by standard testing. For space-based systems, the increased confidence in the correctness of the behavior facilitates avoidance of situations where fielded products must be operated at a degraded level of functionality.

Increasingly, organizations are developing systems using a product line approach, where single software architecture serves as a baseline for building a family of similar systems. The baseline architecture must thus implement or support the requirements for certain quality-attributes in the product line system. In making architectural choices, the software engineering community is interested in understanding how to pick architectures or architectural styles that implement or support the requirements for quality attributes deemed critical to the system. Hence, our future work focuses on 1) formalizing a mapping between quality attributes of a system and software architectural elements, 2) finding techniques for analyzing characteristics of quality attributes and how they map to architectural styles, 3) Creating techniques for constructing quality-attribute specific product line derivatives and 4) Industrial case study analysis of the effectiveness of the entire approach. In addition, we are investigating how the use of reverse engineering can be applied to existing code in order to determine behavior of existing systems with the intent of facilitating better maintenance and evolvability of embedded software and embedded software product lines.

Acknowledgments

We thank Dr. John C. Kelly for his continued support of this work. We thank Dr. Braden E. Hines, Dr. Charles E. Bell, and Thomas G. Lockhart for helpful discussions and explanations regarding the reuse of interferometry software. Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was provided under NASA's Code Q Software Program Center Initiative, UPN #323-08.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[3] G. C. Gannod and R. R. Lutz. An Approach to Architectural Analysis of Product Lines. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 548–557. ACM, 2000.

[4] G. J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[5] Jpl interferometry projects. http:// huey.jpl.nasa.gov /ice/ice_projects.html.

[6] Origins program. http://origins.jpl.nasa.gov.

[7] M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.