Software Model Checking of ARINC-653 Flight Code with MCP

Sarah J. Thompson
SGT Inc., NASA Ames Research Center
MS269-1, Moffett Field, California
sarah.j.thompson@nasa.gov

Guillaume Brat
CMU, NASA Ames Research Center
MS-269-1, Moffett Field, California
guillaume.p.brat@nasa.gov

Arnaud Venet
SGT Inc., NASA Ames Research Center
MS269-1, Moffett Field, California
arnaud.j.venet@nasa.gov

Abstract

The ARINC-653 standard defines a common interface for Integrated Modular Avionics (IMA) code. In particular, ARINC-653 Part 1 specifies a process- and partition-management API that is analogous to POSIX threads, but with certain extensions and restrictions intended to support the implementation of high reliability flight code.

MCP is a software model checker, developed at NASA Ames, that provides capabilities for model checking C and C++ source code. In this paper, we present recent work aimed at implementing extensions to MCP that support ARINC-653, and we discuss the challenges and opportunities that consequentially arise. Providing support for ARINC-653's time and space partitioning is nontrivial, though there are implicit benefits for partial order reduction possible as a consequence of the API's strict interprocess communication policy.

1 Introduction

NASA missions are becoming increasingly complex, and, more and more of this complexity is implemented in software. In 1977, the flight software for the Voyager mission amounted to only 3000 lines. Twenty years later, the software for Cassini had grown by a factor of ten, and more strikingly, the software for the Mars Path Finder mission amounted to 160 KLOCs (thousands of lines of code). Nowadays, the software for the Space Shuttle has reached half a million lines of code (0.5 MLOCs). Moreover the software for the International Space Station has exploded to 3 MLOCs and it is still increasing. Some experts have estimated that the software required by the Constellation project will reach at least 50 MLOCs. Yet, NASA is still relying on traditional (and expensive) testing and simulation to verify its software.

Naturally, NASA is now exploring new ways to speed up the testing process, reduce its cost, and increase its efficiency. Model checking is seen as a way of helping verifying software, especially for multi-threaded code. Our colleagues at JPL developed the well-known SPIN model checker [12]. Unfortunately, it requires translating code into a modeling language called Promela, which is not always feasible or practical. However Promela models are compiled into C programs that perform the model checking activity. JPL therefore developed *model-driven verification* [13]; it consists of embedding C code fragments into the compiled Promela models. This technique allows them to partially check C programs. NASA Ames has taken a different approach called software model checking. For example, the JPF (Java PathFinder) model checker can check Java programs without any translation [11].

Experience from the JPF project has demonstrated the utility of *software model checking* (i.e. model checking that acts directly on a program, rather than on a model that has been manually extracted from it). However, current flight software is mostly implemented in C, not Java, and in the future it seems increasingly likely that C++ will become the platform of choice. The MCP model checker is being developed to fill the requirement for an explicit-state software model checker, in the style of JPF and

SPIN, that fully supports the C++ language. Moreover, we envision designing a continuum of tools for the verification of C and C++ programs and test case generation. The tools will span techniques ranging from static analysis to model checking.

1.1 ARINC-653

The ARINC-653 standard [1, 3, 2] specifies the software interface to be used by developers of Integrated Modular Avionics flight software. It is comprised of three parts:

Part 1: *Avionics software standard interface (APEX).* This part defines the operating system's interface to avionics code, with specific focus on partition management, process management and error handling.

Part 2: *APEX extensions.* Since Part 1 is insufficient on its own to support many kinds of practical code, Part 2 extends the standard to provide interfaces for file handling, communications and a few other commonly used operating system services.

Part 3: *Compliance testing specification.* Part 3 does not define extra functionality in and of itself – rather, it specifies how compliance testing should be carried out for implementations of parts 1 and 2.

In this paper we concentrate on ARINC-653 Part 1. In a very real sense, Part 1 can be thought of as occupying the same part of the software food chain as POSIX threads [15], though its execution model is somewhat different.

1.1.1 Partitions

The key defining feature of ARINC-653 is its inclusion of *partitions.* A partition is analogous to a Unix process, in the sense that it runs in its own, separate memory space that is not shared with that of other partitions. Partitions also have strictly protected time slice allocations that also may not affect the time slices of other partitions – the standard's aim is to ensure that if a partition crashes, other correctly functioning partitions are unaffected. It is not possible, in standards-compliant code, to define areas of shared memory between partitions – all interpartition communication must be mediated via the APEX API.

One area where partitions differ considerably from Unix processes is in their strict adherence to a well-defined start up and shutdown mechanism, with strict prohibition of dynamic allocation and reconfiguration. On cold- or warm-boot[1], only the partition's primary process may execute. It then starts any other processes, creates and initializes interprocess and interpartition communications channels, allocates memory and performs any other necessary initialization. The primary process then sets the partition's state to NORMAL, at which point no further dynamic initialization (including memory allocation) is allowed. Setting the partition's state to IDLE causes all threads to cease execution.

1.1.2 Processes

ARINC-653 processes are analogous to POSIX threads[2]. A partition may include one or more processes that share time and space resources. Processes have strictly applied priorities – if a process of higher priority than the current process is blocked and becomes able to run, it will preempt the running process

[1] The exact meaning of cold and warm is left to the implementer.

[2] Arguably, the usage of the word 'process' in the standard is unconventional, and can be confusing.

immediately. Processes that have equal priority are round-robin scheduled. Memory is shared between all processes within a partition.

1.1.3 Error handling

Each partition has a special, user-supplied, error handling process that runs at a higher priority than all other processes in the partition. Normally it sits idle, consuming no time resources, until it is invoked as a consequence of an error detected by the operating system or explicitly raised by the running code. It then defines how the partition should respond, and can (for example) cause a partition to be restarted if necessary.

It is possible to define watchdog timeouts for processes that cause the error handler to be invoked automatically if time limits are exceeded.

1.1.4 Events

ARINC-653 Part 1 events are similar, though somewhat simpler than, the event synchronization facilities provided in most other threading APIs. Events may be explicitly in a *set* or *reset* state – when set, they allow all processes waiting on the event to continue. When reset, all processes waiting on the event are blocked. No support for self-resetting events, or events that allow only a single process to proceed are supported, nor is there explicit support for handling priority inversion.

1.1.5 Semaphores

Semaphores in Part 1 behave in the traditional way – they are typically used to protect one or more resources from concurrent access. A semaphore created with an initial resource count of 0 and a resource count limit of 1 behaves exactly like the mutex facilities found in other threading APIs.

1.1.6 Critical Sections

APEX defines a single, global, critical section that, when locked, prevents scheduling. This is a little different, and more extreme in effect, in comparison with the critical section facilities in POSIX threads and in the Windows threading API – rather, it is analogous to turning off interrupt handling.

1.1.7 Buffers and Queuing Ports

ARINC-653 *buffers* are actually thread-safe queues intended for interprocess, intrapartition communication. They are constructed with a preset maximum length and maximum message length which may not be varied at run time. Messages consist of blocks of arbitrarily formatted binary data. Processes attempting to read from an empty buffer block until another process inserts one or more messages. Similarly, attempting to write to an already-full buffer will cause the sending process to block until space becomes available.

Queuing ports are the interpartition equivalent to buffers, and provide queued communication between partitions.

1.1.8 Blackboards and Sampling Ports

Blackboards, ARINC-653's other interprocess/intrapartition communications mechanism, are analogous to buffers, except that they store exactly zero or one messages. Processes may write to blackboards at any time without blocking – the message that they send replaces the existing message, if any. Reading

from an empty blackboard causes the reading process to block until another process writes a message to the blackboard.

Sampling ports are the interpartition counterpart to blackboards. Their semantics are slightly different, in that neither sending or receiving processes ever block. They also add a timestamping facility that makes it easy for a receiving process to check whether the data it has retrieved has become stale.

1.2 The MCP Model Checker

The MCP (Model Checker for C++) project began as an attempt to reimplement the JPF architecture for C++: JPF implements an extended virtual machine interpreting Java bytecode with backtracking, and the first version of MCP had a similar architecture, substituting LLVM [16] for Java. MCP's current architecture is closer to that of SPIN than of JPF, however – rather than running code in an instrumented virtual machine with backtracking, we use program transformation techniques to instrument the program itself, then run it natively in an environment whose run-time system implements backtracking.

1.2.1 LLVM: Low Level Virtual Machine

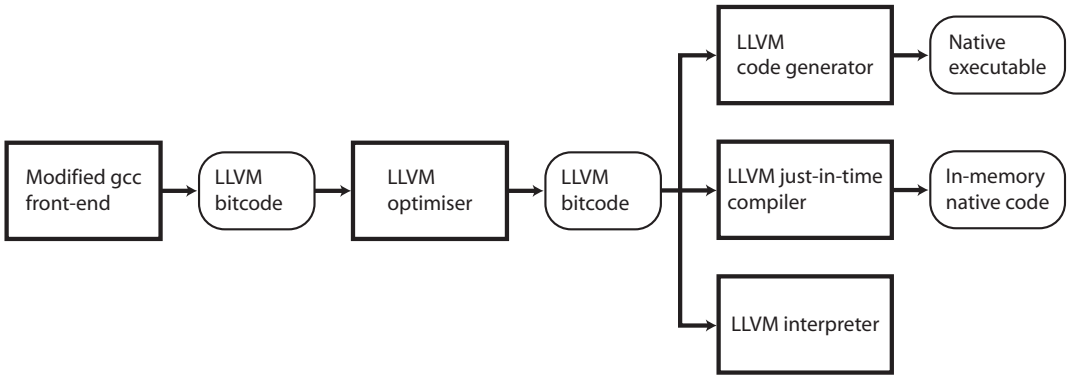

Figure 1: Simplified LLVM architecture

Fig. 1 shows a simplified version of the LLVM flow[3]. A modified version of the `gcc` front-end is used to parse the C++ source code and to lower most of the language's constructs to a level closer to that of a typical C program. The original `gcc` back-end is discarded in favour of emitting *LLVM bitcode*, which is then optimised and passed on to various alternative back-ends.

The LLVM bitcode format was specifically designed to support program analysis, transformation and optimization – a Static Single Assignment (SSA) representation [8] is adopted, making many analyses and transformations (including ours) far more straightforward than they might otherwise be.

1.2.2 The MCP Architecture

Fig. 2 shows an outline of the MCP architecture. Functionality is split across several subsystems:

Transformation Passes Several MCP-specific transformations that instrument the code under test are implemented as an extension module for LLVM's `opt` program transformation framework.

[3]Many LLVM tools have been omitted here for clarity – LLVM is a large, rich toolset, so we concentrate on the subsystems that are specifically relevant to MCP.

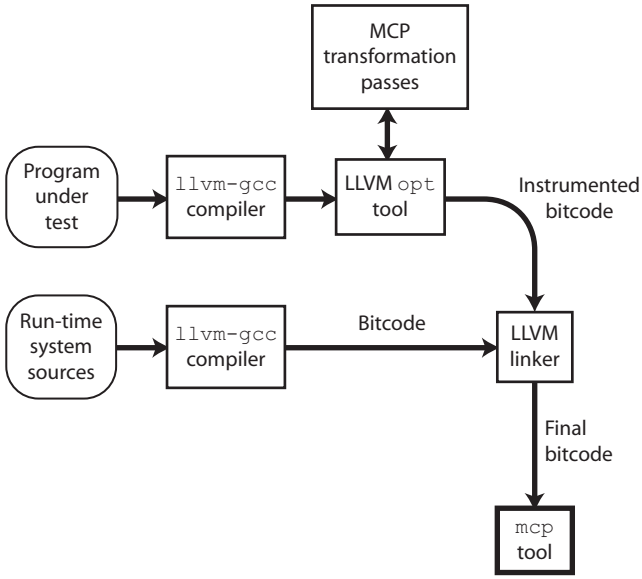

Figure 2: MCP Architecture

Run-time System A run-time system is implemented in C++, compiled with LLVM's `gcc` front-end, then linked with the program under test after it has been transformed. Its primary purpose is to intercept system calls that MCP needs to handle differently, *e.g.*, `printf`, `malloc`, `free`, `memset`, `memmove` and `memcpy`. This approach also provides a convenient place to implement compatibility wrappers that allow code written to specific operating system APIs to be handled without modification – the ARINC-653 subsystem is implemented as an extension to the run-time system.

JIT Environment/User Interface Model checking is initiated by users by running the `mcp` command-line application (see Section **??**). The `mcp` tool comprises an instance of the LLVM just-in-time (JIT) compiler environment, as well as MCP's implementations of memory versioning, hashing, state space searching etc.

1.2.3 Emulating threading

One of the trickier issues surrounding the model checking of C++ source code is the fact that the language standard specifically does not mandate any particular threading model. Real-time program semantics are therefore dependent upon the execution environment, so any attempt to analyse multithreaded code must inevitably make some kind of assumption about the underlying threading model.

MCP implements its own low-level API, on top of which arbitrary threading models may be constructed. This API is deliberately constructed to, as far as possible, serve as a superset of all of the threading models that are of interest.

1.2.4 Search Algorithms

MCP implements several built-in search strategies:[4].

[4]Since MCP's partial order reduction algorithm does not presuppose any particular execution order, MCP does not have a performance bias toward any particular search strategy.

Breadth-first search Paths through the search space are explored in depth order, beginning with the shortest first.

Depth-first search Paths are executed to completion before backtracking continues.

Heuristic search In this strategy, the test harness provides a ranking function that allows MCP to implement a best-first search strategy.

Randomised search Randomised search explores the search space in a random order. In practice, randomised search has a 'look and feel' somewhere between that of breadth- and depth-first search, but in practice it has less tendency to get stuck in local minima.

1.2.5 Assertions and Backtrace Generation

When an assertion occurs or when MCP detects an error (*e.g.*, due to a segmentation fault, deadlock, assertion failure or some other problem in the code under test), it generates a backtrace from the beginning of execution of the current execution trace until the most recently executed instruction is reached. Backtrace logs may optionally include all executed LLVM instructions, executed source lines and all memory contents that are read or written by the program. Only the actual trace from the currently executing code fragment is generated – the (usually enormous and irrelevant) logging information from other traces that did not lead to errors are ignored.

2 Model checking ARINC-653 code

Since MCP supports the entire C++ programming language, the special considerations required for checking ARINC-653 code are specifically related to the implementation of its peculiar threading semantics. Though much of ARINC-653 can be mapped to existing approaches, there is sufficient oddity that a simplistic approach such as mapping APEX to and existing POSIX threads implementation is not sufficient.

Time management Like most threading APIs, ARINC-653 Part 1 provides facilities for dealing with time, ranging from sleeping for a particular interval, waiting until a specific time, timeouts on waiting on blocking synchronization objects, etc. It also supplies some facilities that are squarely aimed at avionics code, such as watchdog timeouts and automatically detecting stale data.

Ideally, it would be preferable to be able to accurately model program execution time. However, since actual flight code is likely to be targeted at an entirely different CPU architecture [14, 22] and compiled with a different toolchain [9], this is impractical, and indeed it could be potentially dangerous to extrapolate results from one platform to another. Therefore, code is assumed to run arbitrarily (though not infinitely) quickly, unless it explicitly waits via an API call. We therefore concentrate on modeling time at the level of explicit waits and timeouts rather than at the level of instructions.

Since MCP backtracks, its time implementation must also be able to backtrack. Consequentially, time is emulated rather than taken from a real-time clock. This has a number of benefits, not least of which being that arbitrarily long wait intervals can be emulated without needing to actually wait for the specified length of time – modeling, for example, Earth-Mars communications links with very long packet round-trip times, becomes feasible and efficient.

Process management ARINC-653 Part 1 processes map fairly directly to MCP's existing thread model, so could be accommodated relatively straightforwardly. Partitions were trickier to support, because they required a new memory protection system to be implemented that could segment memory access and check that it is being accessed with appropriate partitions. This turned out to be the single largest change necessary to MCP in order to support the APEX API.

Partition start up/initialization control Most of the requirements of partition start-up and initialization control involve parameter checking within API calls, so this is dealt with largely with assertions in the API implementation. MCP's memory manager was extended to support a flag that causes memory allocation to throw an error, making it possible to detect accidental memory allocations while a partition is in NORMAL mode.

Sampling ports, queuing ports, buffers and blackboards These communications APIs were implemented fairly straightforwardly, and did not require alteration to the MCP core. Synchronization was implemented purely in terms of MCP's native event mechanism.

Events, semaphores, critical sections and error handling These APEX API features mapped more or less directly to MCP's existing facilities. Some changes were necessary, but these were mostly consequential to the partitioned memory model support.

3 Partial order reduction under a partitioned memory model

Partial order reduction is carried out by model checkers in order to reduce the impact of the exponential time complexity inherent in backtracking. Given N representing the size of the program fragment under test and a representing the amount of possible nondeterminism at each decision point, time complexity for explicit state model checking has an upper bound proportional to a^N. It therefore behooves us to try to get a down to as close to 1 as possible, because this has a dramatic effect on the size of program N that can be practically analyzed. Partial order reduction techniques typically attack this in two ways: making the execution steps bigger by bundling thread-local operations together (thereby effectively reducing N), and where possible bundling together nondeterministic choices that have equivalent consequences (reducing a where possible). MCP does both – it leverages LLVM's static analysis and optimization capability in order to make the steps between yield points as large as possible, and second mechanism tracks reads and writes to shared memory, suspending evaluation lazily when only a subset of currently running threads have touched the relevant locations.

Though MCP's existing partial order reduction strategy can be applied to ARINC-653 code, there are some potential benefits available as a consequence of the partitioned execution scheme. In particular, the decision to only allow partitions to affect each other via API calls has profound consequences. A partition with a single process, or with multiple processes none of which having the same priority, is inherently deterministic. Therefore, nondeterminism may only arise as a consequence of timing relationships between such partitions. The MCP APEX implementation optionally treats partitions as executing atomically between API calls, offering a huge speedup with minimal time or memory overhead. Initial results are encouraging, though at the time of writing this functionality is too new for it to be possible to quote performance statistics.

4 Related work

Structurally, MCP probably bears closest resemblance to JPF (Java Pathfinder) [11], though at the time of writing it does not approach JPF's maturity. The most significant differences between JPF and MCP

stem from the differences between Java and C++; for example, JPF takes advantage of reflection and the standard threading package in Java, which MCP can not since those features are not present in C or C++.

Since JPF is a explicit-state model checker "a la SPIN", MCP is also a close cousin to SPIN [12]. Besides the explicit-state model, MCP also shares with SPIN the concept of compiling the model checking problem into an executable program. SPIN starts with Promela models while MCP performs transformations on C/C++ programs to embed the model checking problem into the original program.

Another model checker directly addressing C++ is Verisoft [10], which takes a completely different approach. Verisoft follows a stateless approach to model checking while MCP follows a conventional explicit-state model similar to SPIN [12].

CBMC is a bounded Model Checker for C and C++ programs [5]. It can check properties such as buffer overflows, pointer safety, exceptions and user-specified assertions. CBMC does model checking by unwinding loops and transform instructions into equations that are passed to a SAT solver. Paths are explored only up to a certain depth.

There are, however, several model checkers that address C. SLAM [4] is really more of a static analyzer than a model checker. It relies heavily on abstractions, starting from a highly abstracted form and building up to a form that allows a complete analysis. CMC [17] uses an explicit-state approach, but it requires some manual adaptation when dealing with complex types (*pickle* and *unpickle* functions).

Finally, there have been some attempts within NASA to use the Valgrind tool [18, 20] as a model checker. Unfortunately, it implies using very crude steps between transitions.

Other approaches to model checking code involve a translation step, be it automatic or manual. For example, Bandera [7] provides model checking of Java programs by translating automatically the program into a PVS [19], Promela [21] or SMV [6] model.

5 Conclusions

The ARINC-653 Part 1 support in MCP is very new and still under development, so the purpose of this paper is to provide a first look at the capability within NASA's wider formal methods community.

5.1 Future work

Other than a requirement for detailed testing and attention to detail with respect to standards compliance, the Part 1 implementation is largely complete at the time of writing. We hope, over the next few months, to put together a larger demonstration of the technology based on a moderate-sized, ARINC-653-based flight code model example, which will help us further tune our APEX implementation. If sufficient interest is shown, extending our implementation to encompass Part 2 would be feasible.

Acknowledgments

The work described in this paper was supported by the NASA ETDP (Exploration Technology Development Program) Intelligent Software Design Project

References

[1] Aeronautical Radio Inc. *Avionics application software standard interface Part 1 – required services*, 12 2005. ARINC Specification 653P1-2.

[2] Aeronautical Radio Inc. *Avionics application software standard interface Part 3 – conformity test specification*, 10 2006. ARINC Specification 653P3.

[3] Aeronautical Radio Inc. *Avionics application software standard interface Part 2 – extended services*, 12 2009. ARINC Specification 653P2-1.

[4] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers, 2006.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *Proc. TACAS*, pages 168–176, 2004.

[6] CMU. The SMV system. http://www.cs.cmu.edu/ modelcheck/smv.html.

[7] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[9] GHS Inc. *Green Hills Software Inc. web site*. http://www.ghs.com/.

[10] P. Godefroid. Model checking for programming languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[11] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, V2(4):366–381, March 2000.

[12] G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, September 2003.

[13] G. Holzmann and R. Joshi. Model-driven software verification. In *Proceedings of the 11th SPIN Workshop*. ACM SIGSOFT, April 2004.

[14] IBM. *PowerPC 740, Power PC 750 RISC Microprocessor User's Manual*, 2 1999. GK21-0263-00.

[15] IEEE. *IEEE Standard for Information Technology – Portable operating system interface (POSIX). Shell and utilities*, 2004. 1003.1-2004.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] M. Musuvathi, A. Chou, D. L. Dill, and D. Engler. Model checking system software with CMC. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*, pages 219–222, New York, NY, USA, 2002. ACM Press.

[18] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI 2007*, June 2007.

[19] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 411–414, London, UK, 1996. Springer-Verlag.

[20] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, April 2005.

[21] SPIN web site. Promela language reference. http://spinroot.com/spin/Man/promela.html.

[22] Synova Inc. *Mongoose-V MIPS R3000 Rad-Hard Processor web site*. http://www.synova.com/proc/mg5.html.