Applications of Model Checking at Honeywell Laboratories ⋆

Darren Cofer, Eric Engstrom, Robert Goldman, David Musliner, Steve Vestal

Honeywell Laboratories, Minneapolis MN 55418, USA
darren.cofer@honeywell.com

Abstract. This paper provides a brief overview of five projects in which Honeywell has successfully used or developed model checking methods in the verification and synthesis of safety-critical systems.

1 Introduction

Embedded software in control and communication systems is becoming increasingly complex. Verification of important safety or mission-critical properties by traditional methods of test and design review will soon be impossible or prohibitively expensive. The only way developers of complex systems will be able to manage life cycle costs and yet still field systems with the functionality that the market demands is to rely on mathematical models and analyses as the basis for our designs.

For several years Honeywell has been investigating the use of model checking techniques to analyze the behavior and correctness of a variety of safety and mission-critical systems. This paper provides a brief overview of five projects in which we have successfully used or developed model checking tools and methods.

2 Automatic Synthesis of Real-Time Controllers

Unmanned Aerial Vehicles (UAVs) under development by the military and deep space probes being developed by NASA require autonomous, flexible control systems to support mission-critical functions. These applications require hybrid real-time control systems, capable of effectively managing both discrete and continuous controllable parameters to maintain system safety and achieve system goals.

We have developed a novel technique for automatically synthesizing hard real-time reactive controllers for these and other similar applications that is based on model-checking verification. Our algorithm builds a controller incrementally, using a timed automaton model to check each partial controller for correctness. The verification model captures both the controller design and the

⋆ This material is based in part upon work supported by Rome Labs (contract F30602-00-C-0017), AFOSR (contract F49620-97-C-0008), and NASA (cooperative agreement NCC-1-399)

semantics of its execution environment. If the controller is found to be incorrect, information from the verification system is used to direct the search for improvements. Using the CIRCA architecture for adaptive real-time control systems [6], these controllers are synthesized automatically and dynamically, on-line, while the platform is operating. Unlike many other intelligent control systems, CIRCA's automatically-generated control plans have strong temporal semantics and provide safety guarantees, ensuring that the controlled system will avoid all forms of mission-critical failure.

CIRCA uses model-checking techniques for timed automata [11] as an integral part of its controller synthesis algorithm. CIRCA's *Controller Synthesis Module* (CSM) incrementally builds a hard real time reactive controller from a description of the processes in its environment, the control actions available, and a set of goal states. To do this, the CSM must build a model of the controller it is constructing that is faithful to its execution semantics and use this model to verify that the controller will function safely in its environment.

CIRCA employs two strategies to manage this complex task. First, its mission planner decomposes the mission into more manageable subtasks that can be planned in detail. Second, CIRCA itself is decomposed into two concurrently-operating subsystems (see Figure 1): an *AI Subsystem* including the CSM reasons about high-level problems that require powerful but potentially unbounded computation, while a separate *real-time subsystem* (RTS) reactively executes the generated plans and enforces guaranteed response times.

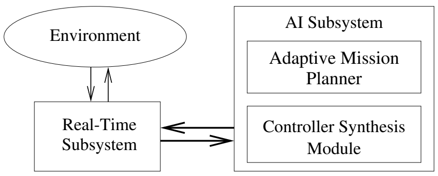

Fig. 1. Basic CIRCA architecture.

The CIRCA CSM builds reactive discrete controllers that observe the system state and some features of its environment and take appropriate control actions. In constructing such a controller, the CSM takes a description of the processes in the system's environment, represented as a set of transitions that modify world features and that have worst case time characteristics. From this description, CIRCA incrementally constructs a set of reactions and checks them for correctness using a timed automaton verifier.

The real-time controllers that CIRCA builds sense features of the system's state (both internal and external), and execute reactions based on the current state. That is, the CIRCA RTS runs a memoryless reactive controller.

Given the above limitation on the form of the controller, the controller synthesis problem can be posed as choosing a control action for each reachable state (feature-value assignment) of the system. This problem is not as simple as it sounds, because the set of reachable states is not a given — by the choice of control actions, the CSM can render some states (un)reachable.

Indeed, since the CSM focuses on generating safe controllers, a critical issue is making failure states unreachable. In controller synthesis, this is done by the process we refer to as preemption. A transition t is preempted in a state s iff some other transition t' from s must occur before t could possibly occur.

Note that the question of whether a transition is preempted is not a question that can be answered based on local information: preemption of a transition t in a state s is a property of the controller as a whole, not of the individual state. It is this non-local aspect of the controller synthesis problem that has led us to use automatic verification.

3 Real-Time Scheduler of the MetaH Executive

MetaH is an emerging SAE standard language for specifying real-time fault-tolerant high assurance software and hardware architectures[8]. Users specify how software and hardware components are combined to form an overall system architecture. This specification includes information about one or more configurations of tasks, their message and event connections, information about how these objects are mapped onto a specified hardware architecture, and information about timing behaviors and requirements, fault and error behaviors and requirements, and partitioning and safety behaviors and requirements.

The MetaH executive supports a reasonably complex tasking model using preemptive fixed priority scheduling theory [1, 2]. It includes features such as period-enforced aperiodic tasks, real-time semaphores, mechanisms for tasks to initialize themselves and to recover from internal faults, and the ability to enforce execution time limits on all these features (time partitioning).

Traditional real-time task models cannot easily handle variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors. One of the goals of this project was to use dense time linear hybrid automata models to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory.

Figure 2 shows an example of a simple hybrid automata model for a preemptively scheduled, periodically dispatched task. A task is initially waiting for dispatch but may at various times also be executing or preempted. The variable t is used as a timer to control dispatching and to measure deadlines. The variable t is set to 0 at each dispatch (each transition out of the waiting state), and a subsequent dispatch will occur when t reaches 1000. The assertion $t \leq 750$ each time a task transitions from executing to waiting (each time a task completes) models a task deadline of 750 time units. The variable c records accumulated compute time, it is reset at each dispatch and increases only when the task is in the computing state. The invariant $c \leq 100$ in the computing state means

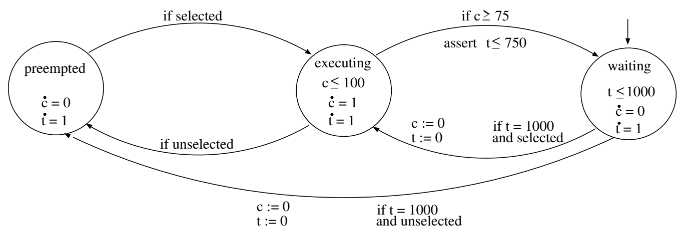

Fig. 2. Hybrid Automata Model of a Preemptively Scheduled Task.

the task must complete before it receives more than 100 time units of processor service, the guard $c \geq 75$ on the completion transition means the task may complete after it has received 75 time units of processor service (i.e. the task compute time is uncertain and/or variable but always falls in the interval $[75, 100]$).

In this example the edge guards `selected` and `unselected` represent scheduling decisions made at scheduling events (called scheduling points in the real-time literature). These decisions depend on the available resources (processors, busses, etc.) being shared by the tasks.

We began our work using an existing linear hybrid automata analysis tool, HyTech [4], but found ourselves limited to very small models. We developed and implemented a new reachability method that was significantly faster, more numerically robust, and used less memory. However, our prototype tool allows only constant rates (not rate ranges) and does not provide parametric analysis.

Using this new reachability procedure we were able to accomplish one of our goals: the modeling and verification of a piece of real-time software. We developed a hybrid automata model for that portion of the MetaH real-time executive that implements uniprocessor task scheduling, time partitioning and error handling. Results of this work are presented in [9].

Our hybrid automata model was not developed using a separate modeling language. Instead, statements were added to the code to generate pieces of the model as subprograms were unit tested. When unit testing of all subprograms was completed, the complete system model was then subjected to reachability analysis. This provided a high degree of traceability between code and model.

The conditions we checked during reachability analysis were that all deadlines were met whenever the schedulability analyzer said an application was schedulable; no accessed variables were unconstrained (undefined) and no invariants were violated on entry to a region; and no two tasks were ever in a semaphore locking state simultaneously. Assertion checks appearing in the code were also captured and verified in the model.

We discovered nine defects in the course of our verification exercise. In our judgement, three of these would have been almost impossible to detect by testing due to the multiple carefully timed events required to produce erroneous behavior.

There are limits on the degree of assurance that can be provided, but in our judgement the approach may be significantly more thorough and significantly less expensive that traditional testing methods. This result suggests the technology has reached the threshold of practical utility for the verification of small amounts of software of a particular type.

4 Fault-Tolerant Ethernet Protocol

Large industrial control systems require highly reliable communcation services, especially in chemical processing applications. We have developed a Fault-Tolerant Ethernet (FTE) communication network for industrial control applications. The network is composed of dual redundant LANs implemented with standard commercially available hardware and software drivers. It is transparent to control applications, both in terms of application coding and communication latency.

The original FTE protocols performed failure detection and recovery for single point of network failure. We used the Spin model checker [5] to produce a simple model of the dual LAN fault detection algorithm and verify the correctness of the initial version of the protocols. We found one significant error and some minor ambiguities and potential design errors that were addressed in the final design and implementation of the system.

We first constructed a single node model to verify that faults are correctly detected by the fault detection algorithm, and that single message losses are tolerated. The model consisted of four processes: the two LANs, a state broadcast process (sends pairs "I'm alive" messages on the two LANs), and the fault detection algorithm.

Faults were injected by permitting single message losses in each of the two LANs. Messages could be lost at any time, as long as two consecutive messages on a LAN were never lost. In this situation, the fault detection algorithm should never enter its error state.

Verification of this model identified an execution in which the error state could be (incorrectly) entered, thus exposing an error in the algorithm. The problem occured because the algorithm assumed that a lost message always results in the arrival of two consecutive messages from the same LAN. However, message ordering is not sufficient to detect a missing message.

The fault detection algorithm was revised to add a short sequence number to each "I'm alive" message. The maximum sequence number must be larger than the number of consecutive lost messages that are to be tolerated, so 2 or 3 bits is sufficient. Each message in a pair is given the same sequence number so pairs of messages can be identified by matching sequence numbers. However, Spin identified an further counterexample for the revised design in which alternating messages are lost on each LAN. There are never two messages lost in a row on either LAN, but a complete pair of messages is never received.

Two possibilities were considered to deal with this situation:

1. Revise the robustness requirement to exclude the message loss scenario identified above.

2. Relax the fault detection algorithm to tolerate the message loss scenario. This could be done by clearing the missing message counters upon receipt of the missing message, even if its sequence number does not match.

The second approach was selected. This is reasonable since messages are in fact being received from both LANs, so there is no reason to declare a LAN failure.

5 Time Partitioning in Integrated Modular Avionics

The Digital Engine Operation System (DEOS) was developed by Honeywell for use in our Primus Epic avionics product line. DEOS supports flexible Integrated Modular Avionics applications by providing both space partitioning at the process level and time partitioning at the thread level. Space partitioning ensures that no process can modify the memory of another process without authorization, while time partitioning ensures that a thread's access to its CPU time budget cannot be impaired by the actions of any other thread.

The DEOS scheduler enforces time partitioning using a Rate Monotonic Analysis (RMA) scheduling policy. Using this policy, threads run periodically at specified periods and they are given per-period CPU time budgets which are constrained so that the system cannot be overutilized [3].

Honeywell engineers and researchers at NASA Ames collaborated to produce a model for use with the Spin model checker [7]. The model was translated from a core "slice" of the DEOS scheduler. This model was then checked for violations of a global time partitioning invariant using Spin's automated state space exploration techniques. We successfully verified the time partitioning invariant over a restricted range of thread types. We also introduced into the model a subtle scheduling error; the model checker quickly detected that the error produced a violation of the time partitioning invariant.

We attempted to verify the following liveness property, which is necessary (but not sufficient) for time partitioning to hold: If the CPU is not scheduled at 100% utilization, then the idle thread should run during every longest period. When verification was attempted with two user threads and dynamic thread creation and deletion enabled, Spin reported a violation. The error scenario results when one of the user threads deletes itself and its unused budget is immediately returned to the main thread (instead of waiting until the next period). This bug was, in fact, one which had been previously discovered by Honeywell during code inspections (but intentionally not disclosed to the NASA researchers performing the verification). Therefore, it would seem that model checking can provide a systematic and automated method for discovering subtle design errors.

Our current time partitioning model does not incorporate several important time-related features of DEOS. These include:

- The existence of multiple processes, which serve as (among other things) budget pools for dynamically creating and deleting threads. Time partitioning must be verified at a process level as well as a thread level.

- Several types of thread synchronization primitives provided by DEOS, including counting semaphores, events, and mutexes. These allow threads to suspend themselves or be suspended in ways not accounted for by the current model.
- The existence of aperiodically running threads, used to service aperiodic hardware interrupts.

We plan to integrate these features into the model and verify that time partitioning still holds with these features present. The principal challenge here will be keeping the state space size manageable while increasing the complexity of the model by incorporating these new features. The current model has already approached the bounds of exhaustive verifiability on currently available computer systems, although subsequent optimizations have reduced the size of the model somewhat. Furthermore, the current model has only been tested on a small range of possible thread budgets and periods.

6 Synchronization Protocol for Avionics Communication Bus

ASCB-D (Avionics Standard Communications Bus, rev. D) is a bus structure designed for real-time, fault-tolerant periodic communications between Honeywell avionics modules. The algorithm we modeled is used to synchronize the clocks of communicating modules to allow periodic transmission. The algorithm is sufficiently complex to test the limits of currently available modeling tools. Working from its specification, we modeled the synchronization algorithm and verified its main correctness property using Spin [10].

The ASCB-D synchronization algorithm is run by each of a number of *NICs* (Network Interface Cards) which communicate via a set of buses. For each side of the aircraft there are two buses, a primary and a backup bus. Each NIC can listen to, and transmit messages on, both of the buses on its own side. It can also listen to, but not transmit on, the primary bus on the other side.

The operating system running on the NICs produces *frame ticks* every 12.5 msec which trigger threads to run. In order for periodic communication to operate, all NICs' frame ticks must be synchronized within a certain tolerance. The purpose of the synchronization algorithm is to enable that synchronization to occur and to be maintained, within certain performance bounds, over a wide range of faulty and non-faulty system conditions.

The synchronization algorithm works by transmitting special timing messages between the NICs. Upon initial startup, these messages are used to designate the clock of one NIC as a "reference" to which the other NICs synchronize; after synchronization is achieved, the messages are used to maintain synchronization by correcting for the NICs' clock drift relative to each other. The algorithm is required to achieve synchronization within 200 msec of initial startup regardless of the order in which the NICs start.

The synchronization algorithm must also meet the 200 msec deadline in the presence of malfunctioning NICs or buses. For example, any one of the NICs might be unable to transmit on, or unable to listen to, one or more of the buses; or it might babble on one of the buses, sending gibberish which prevents other messages from being transmitted; or one of the buses might fail completely at startup, or fail intermittently during operation.

The introduction of an explicit numerical time model, and the combination of that time-modeling capability and the message-transmission capability in the same "environment" process, allowed us to produce a tractable four-NIC model that includes most of the important features of the synchronization algorithm.

The environment process encapsulates all those parts of the system that provide input to the algorithm we wish to model (frame ticks, buffers, and buses), while the NIC process encapsulates the algorithm itself. The interface between the two is simple and localized. It allows faults to be injected and complicated hardware interactions to be added with no change required to the NIC code. Complicated tick orderings produced by frames of different lengths are explicitly and accurately represented in the model. Because the interface between environment and NIC includes all the data that must be shared between them, there is no need for global data structures. This allows Spin's compression techniques to reduce the memory required to store each state.

With this model we were able to verify the key system property as an assertion in the environment process that states that all NICs should be in sync within 200 msec of the startup time.

References

1. P. Binns. Scheduling Slack in MetaH. *Real-Time Systems Symposium*, December 1996.
2. P. Binns. Incremental Rate Monotonic Scheduling for Improved Control System Performance. *Real-Time Applications Symposium*, 1997.
3. P. Binns. Design Document for Slack Scheduling in DEOS. Honeywell Technology Center Technical Report SST-R98-009, September 1998.
4. T. Henzinger, P. Ho, H. Wong-Toi. A User Guide to HyTech. University of California at Berkeley, `www.eecs.berkeley.edu/~tah/HyTech`).
5. G. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, vol. 23, no. 5, May 1997, pp. 279-295.
6. D. Musliner, E. Durfee, and K. Shin. CIRCA: a cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics* 23(6):1561-1574.
7. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. *ICSE 2000.*
8. S. Vestal. An architectural approach for integrating real-time systems. *Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997.
9. S. Vestal. Modeling and verification of real-time software using extended linear hybrid automata. *Fifth NASA Langley Formal Methods Workshop*, June 2000 (see `http://atb-www.larc.nasa.gov/fm/Lfm2000/`).
10. N. Weininger, D. Cofer. Modeling the ASCB-D Synchronization Algorithm with Spin: A Case Study. *7th International Spin Workshop*, September 2000.
11. S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, vol. 1, no. 1/2, Oct. 1997.