Testing software suffering from hardware

Pieter J. Schoenmakers Jochen A.G. Jess
Department of Electrical Engineering
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, the Netherlands

Abstract

In this paper we propose a method for testing control software of embedded systems independent of the hardware that is required by the software for normal operation. This independence decouples, to some extent, software development from the development of the system's hardware, implying that (sub-) system level testing of the software can commence prior to hardware availability. In addition, the method makes stress, load, and frequent regression testing economically viable.

I. Introduction

In large co-designed embedded systems two kinds of software can be discerned. One kind is largely concerned with algorithmic data processing; the other kind is the software that handles the control of the whole system. This control software can be very complex; it is the result of a design effort by a possibly large team. In a concurrent design process, this team faces the problem of testing the control software, or parts thereof, while the hardware system to run it on is not yet available. In the project's maintenance phase, hardware will be available but testing every corner of the software while depending on the actual hardware might not be a feasible option. All in all, it is desirable, often mandatory, to be able to test the control software independent of the hardware system.

In this paper we propose a general method for testing control software independent of the hardware. More specifically, we describe the facilities that must be offered by the test environment and the system level specification language to support this method. A major goal of our work is to enable unattended test runs of load, stress, and other system-level tests. Unattended runs enable frequent (regression) testing, which increases the confidence in the system's correct operation.

In this paper we do not propose any new strategies for designing tests, nor do we introduce new kinds of tests. Numerous approaches to testing software, or specific details thereof, already exist [1]. We want to be able to apply those techniques to situations where the software to be tested can not meaningfully operate without the hardware with which it interacts. In this paper, we propose a method that enables running tests without that hardware.

The proposed testing method is a generalization of specific solutions to the problem of testing software independent of the hardware. An example of such a solution is GUI testing, where the hardware to be eliminated during testing consists of the mouse, keyboard, and screen. Various approaches to GUI testing exist. They all have in common that, at some level of abstraction, user control of the mouse and keyboard is run by a test driver, and, also at varying levels of abstraction, the program output or state is checked against the expected results as defined for the test. Our goal is to develop a generic framework that supports such hardware elimination and testing of the resulting system.

In the following sections, we will outline the design method and its requirements and briefly introduce the language we have developed in support of the method. Before concluding this paper, we will describe in detail the example of implementing and testing an elevator control program.

II. Outline

Testing software independent of the hardware with which it normally interacts, requires that during testing the hardware is replaced by a model of the hardware. This model can e.g. be provided by a hardware emulator, a simulator running a description of the hardware, but also by software written specifically for this purpose.

During normal operation, the interfaces between hardware and software consist of pieces of software which provide the control software with an abstraction of the hardware. We use the term *hardware abstractions* to denote these interfaces[1] (figure 1a). During test runs, each hardware abstraction must instead

[1]We use 'hardware abstraction' instead of the widely used term 'driver' to avoid confusion with the term 'test driver'.

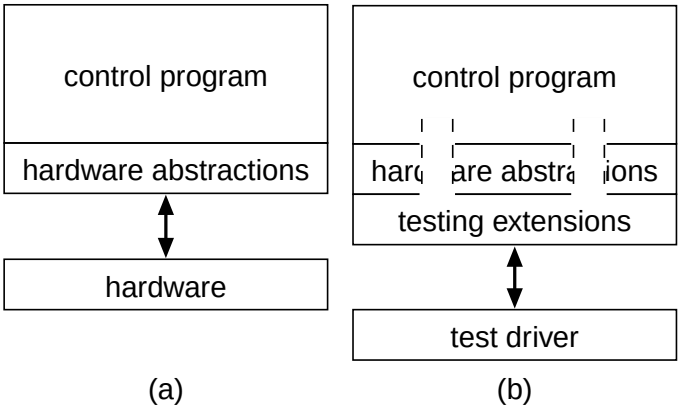

Fig. 1. *normal situation (a) and testing situation (b).*

provide an abstraction of a *model of the hardware*.

While testing, the software to be tested (the *system under test* or SUT) is controlled by a test driver. The test driver is responsible for executing the tests and verifying their outcome. To make the SUT ready for execution, it is extended with *testing extensions* which change the SUT's abstractions of the hardware into abstractions of the hardware models, making the SUT independent of the hardware on which it normally depends (figure 1b). These hardware models then provide the SUT with its inputs. They in turn receive directions from the *simulated world* in which they operate. The simulated world is maintained by the test driver, which can be customized to suit the SUT.

The test driver program provides a framework in which the simulated world can be created and scenarios can be generated and, if applicable, recorded and played back. Tests can be verified by comparing the outcome against the outcome dictated by the scenario or by checking the system's behavior against sets of invariants and postconditions. These conditions can apply to the SUT, parts thereof, or parts of the simulated world.

An important advantage of modeling the hardware purely by software is that this enables modeling at higher levels of abstraction than plain interfaces. Furthermore, the hardware models only need to be as good as necessary to achieve the testing goal. For example, when doing basic transaction testing on an elevator control, only the scheduling subsystem is being tested and timing information can be safely omitted from the models. Only when the tests are extended to include information on, for instance, time to service, the models must take into account the time it takes for the cage to move and for passengers to enter and exit. In effect this means that the testing models can be developed in parallel with the system: new models can be included as each subsystem is implemented and existing models can be refined when the testing needs increase.

It is possible for the outcome of tests to not be verifiable by observing the behavior of the hardware models. In such cases the testing extensions must include functionality to observe the SUT's internal state. Furthermore, they are also necessary to set the SUT's state at the start of a test. They should of course not alter the system's behavior as that would invalidate the test results or, even worse, violate the truthfulness of the tests.

It is important to note that the testing extensions are not an inherent part of the system's functionality. These extensions are absent during normal operation, and must therefore not be part of the system's design. If they were, they would unnecessarily increase the complexity of the system's specification. This observation signifies the difference between design for test (DfT) for hardware and for software: with hardware DfT, specific facilities must be incorporated into the design to, for instance, gain access to the hardware's internal state, whereas with software, access is essentially unrestricted, were it not for the specification language or paradigm used.

The target machine on which the control software is to run might not be capable of supporting the software with the testing extensions. The testing process must therefore be independent of the machine on which it is performed. To support real time testing, the test driver must support distinctive notions of elapsed real time, elapsed simulated time, and elapsed CPU time. With simulated time being unrelated to elapsed real time, simulated time can be sped up to decrease the real time needed to run the tests, and it can be slowed down if time is needed to compute (part of) the hardware model. Since the testing process is independent of the machine on which testing takes place, a suitable machine can be chosen on which, for instance, coverage and profiling tools are available.

III. Facilities

To support the testing method outlined above, we define the following requirements:

1. a test driver which provides an extensible simulated world, and control of and communication with the SUT and its testing extensions,

2. a framework to support testing extensions and their interaction with the test driver,
3. the possibility to extend the control software so as to make it suitable for execution under the test driver, i.e. to change the hardware abstractions into abstractions of models of the hardware, and
4. the possibility to extend the software in order to make its internal state available to the test driver, and to have the driver set the system's state.

Points 3 and 4 basically state identical requirements: the possibility to extend a program without those extensions being part of the program's design. If this extensibility does not require recompilation of the original program, it offers some interesting advantages:

- older versions of the program can be tested against the latest test extensions without needing to retrieve and compile the old version of the sources, and
- testing extensions can always be developed and applied, even in case (part of the) source code is unavailable.

IV. Language

In support of the proposed testing method, we have designed and implemented TOM: an object oriented programming language in the spirit of Objective-C [2] and Java [3] (ante factum), which supports the testing paradigm described in this paper. To this end, TOM provides the following facilities:

- classes are extensible entities: a class is fully defined by its main definition and any extensions defined for it,
- an extension can add and replace methods. In support of complex added behavior, an extension can add instance variables. To promote object reuse, an extension can introduce additional superclasses, and
- extensions can be added to a program at compile, link, or run time.

We will not further discuss TOM here; the interested reader is referred to [4].

V. An example

An example application of the testing method outlined in this paper is that of an elevator control. This is a rather simple application, without any hard real-time constraints. It does not require a very strict notion of simulation time to be tested, implying that the test driver can be simple.

The elevator control program is designed to control n systems, where system i controls l_i elevators that act as a group to service f_i floors; every elevator can service every floor.

The elevator control interacts with the system through various input and output devices. The input devices are all switches, the following types of which can be discerned.

request Two switches on each floor, except the top and bottom one, to request traveling up or down. The bottom and top floors each have only a single request switch. A request switch is pushed by a user who wants to travel in the indicated direction.

destination Every elevator has a destination switch for each floor. The user employs these switches to indicate the floor to which he wants to travel.

guidance Every normal floor has three guidance switches: one at the precise location of the floor, one just above, and one just below. When the first switch is pushed, the elevator doors can be safely opened. The other two switches announce a floor, allowing the cage to slow down before arriving at a floor.

limit Each end of an elevator shaft has a limit switch. These switches are supposed never to be activated.

In the output direction, the system controls:

displays the information displays in each elevator and on every floor, and

engines the engine of each elevator.

A. Design

The design of the elevator system is rather straightforward, given the preceding description of the hardware. The only addition to the following class descriptions is implicit information about the implementation of the scheduler.

System A `System` is a group of elevators. Each `System` maintains a set of `Elevator` objects and a set of `Floor` objects. A request from a floor is handled by the system, which dispatches it to an idle elevator when available. Requests that can not be immediately dispatched are stored, with a separate set for each direction.

Floor A `Floor` is a placeholder for the switches and displays present on each floor in the system. Each floor also maintains the status of the up and down requests on that floor. Basically, this is a copy of information also stored elsewhere. The redundancy in this case is, however, easy to maintain and helps checking the consistency between the

state of the world between various parts of the system.

Elevator An `Elevator` is the placeholder for the switches and displays it contains. An elevator travels in some general direction, servicing floors as indicated by the set of requests it maintains, plus any floor it passes that has a pending request at the system, which has not yet been assigned to an Elevator.

Engine Each `Elevator` has one `Engine` which abstracts the actual engine that moves the elevator's cage.

Switch To the software all switches are equal; they are instances of the `Switch` class. No subclasses are needed for different purposes, since to inform the appropriate object of a state change, the target/action paradigm as employed in OpenStep [5] is used. Using this paradigm, the target can be any class, without restricting the switch/target interface to certain methods or requiring a special switch subclass for each different target and action.

Display Every display in the system is represented by a `Display` object. The system does not differentiate between various physical displays.

The elevator control interacts with the hardware through a binary bidirectional stream: a high-level representation of a network used to communicate with the actual hardware. Incoming events are interpreted by a parser and dispatched to the appropriate hardware abstracting object. In the other direction, commands are issued onto the stream only from specific methods of the appropriate hardware abstractions. It is obvious that this interaction with the hardware is not totally realistic, but targeted at testing this 'virtual' interface from a tty.

B. Testing

In the testing situation the classes in the elevator control are modified to not rely on the actual hardware. This means that any interaction with the hardware and the bidirectional stream modeling the network is suppressed. Furthermore, a simulated world is maintained with the following properties:

- An elevator cage travels at constant speed, with infinite acceleration and deceleration.
- The virtual world is inhabited by virtual persons. These persons can travel by elevator; they issue commands to the system by operating the destination and request switches.

To create the simulated world in which to run the tests, the testing extension modifies the program's classes in the following way:

- The `Engine` class is modified to incorporate the model of the moving cage. This model is responsible for triggering the guidance and limit switches, a function normally performed by the event parser.
- The `Elevator` and `Floor` classes are modified maintain a set of persons as observer; they are informed of events concerning the entity they observe, such as the arrival of an elevator at a floor.
- A new `Person` class is introduced, to model the persons inhabiting the simulated world. They can observe a floor or elevator, and command them by pushing request and destination switches.
- Most of the functionality of the `Display` class is disabled, i.e. replaced with empty methods, the reason being that we're not interested in them in the testing situation.
- Some methods are added to various classes to make their inner state available for inspection or modification.

In this example the test driver is fully contained in the testing extensions added to the program during testing. When, in the future, this testing method has been further developed, the test driver will be a separate, extensible, program. It will maintain the simulated world; the testing extensions of the SUT will be primarily concerned with communication with and making the internal state available to the test driver.

Activity in the simulated world is initiated by the persons inhabiting it. All activity is event based. For example, the moving cage hitting a switch is an event that is inserted, while processing the previous event concerning the cage's motion, into the event queue to occur at the appropriate moment in time.

The event dispatching mechanism is based on the TOM standard library event handling classes `RunLoop`, which monitors file descriptors, and `Timer`, which can be scheduled with a `RunLoop` to perform a method invocation at some moment in time.

In the testing situation, the elevator control program runs on a machine that is not necessarily identical to the machine used in the normal situation. A reason for this deviation can be the unavailability of testing tools such as a coverage analyzer on the original target platform, or simply because a UNIX machine has more to offer to the tester/developer than an embedded system.

With the test driver used in this example, any discrepancy in CPU speed, not to mention the overhead of the test driver itself, directly influences the program's execution speed: because elapsed real time and simulated time are synchronous, this also affects the program in simulated time. Any accurate deep sub-second simulation of the real world is therefore impossible. The unpredictability of CPU scheduling, for instance when a UNIX machine is used to run the tests, further increases this problem.

Fortunately, we do not require a very strict notion of time, since the precise timing of events is of no importance to the elevator control; their order is what really matters. Unfortunately, however, the variation of time means that an event trace can not be used to test the outcome of a test run: when an entity in the testing situation schedules an event to happen at some moment in time relative to the current moment, any variation in 'the current moment' for successive runs, can change the order of events, rendering a trace useless.

The proper execution of test runs of the example is ensured by condition checks. The location of the checks vary: some checks are done by method preconditions and postconditions; other checks are performed by explicit code in the testing extension.

If none of the checks fails during a run, the system has performed satisfyingly. Obviously, satisfaction is only as good as the checks.

C. Test execution

As described in a preceding section, event dispatching in the test driver is done by the event dispatcher as provided by the standard library. This reuse is convenient, were it not that this setup binds simulated time to elapsed time, making a test run simulating a day actually take a day to run. This is not acceptable, especially since the CPU can be observed to be almost idle during that time.

A solution to this problem follows from the observation that the elevator system in the testing situation is a closed system, interacting with its environment only to issue errors or similar messages; no input is received from the environment, not even a file is read. It is therefore safe to modify the event dispatcher to, instead of waiting, immediately continue execution after having changed the program's notion of time accordingly. Various runs of this setup show a reduction in elapsed real time by a factor of up to 2100 for a single elevator 10 person 10 floor situation, with the 'average' speedup being a factor of 600, e.g. for a 4 elevator 20 person 20 floor simulation[2].

VI. Conclusion

In this paper we have sketched a method and supporting development environment that enables testing of embedded system control software independent of the hardware with which the software normally communicates. Initial tests of the method are promising, and further investigation into its applicability will be performed. Future work will include the development of a test driver and testing framework, to ultimately provide a solid development and testing base.

References

[1] Boris Beizer, *Software testing techniques*, Van Nostrand Reinhold, 1990.

[2] NeXT Software, *Object oriented programming and the Objective-C language*, Addison-Wesley, 1993.

[3] James Gosling, Bill Joy, Guy Steele, *The Java language specification*, Addison-Wesley, 1996.

[4] Pieter J. Schoenmakers, *The* TOM *programming language*, `http://tom.ics.ele.tue.nl:8080/`.

[5] NeXT Software, *The OpenStep specification*, 1994.

[2] Tests were run on a 180MHz PA-8000 inside a HP9000/879.