Model Based Testing of an Embedded Session and Transport Protocol

Vesa Luukkala, Ian Oliver

Nokia Research Center
Helsinki, Finland
{vesa.luukkala,ian.oliver}@nokia.com

Abstract. We describe an experience in applying model based testing in verifying especially the parallel behavior of a device level service and discovery protocol. Our approach is two phased: we first define a high level domain model in B and use cases in CSP that can be verified and then create a more detailed reference model that we use for testing the implementation on-the-fly. The use cases are used to drive both the B model and the reference model.

1 Introduction

This paper documents our experiences in applying formal methods and model-based testing approaches in an industrial, semi-formal environment. We were tasked with testing an embedded session and transport protocol for mobile devices based on SOA [1] principles. It is expected that subsystems connected by such a protocol would be provided by an external vendor and must be tested as black box implementations.

We were expected to construct the tester during development of the system, the implementation work had already started and there were no formal specifications for the system. Our main goal was to find bugs arising from concurrency. Experiences of the modeling work and some empirical evidence related to this is described in [2]. During the development the requirements and the the environment evolved forcing us to attempt a less formal and more pragmatic approach.

It is well known that parallel systems are hard to verify and test. Exhaustive verification and proving of correctness are options for systems that are constructed in well controlled environments and often with specialized languages. When testing parallel systems the above problems are augmented by the fact that it is not possible to force an executing parallel system to a certain state as timing and scheduling issues that cannot be influenced from outside of the implementation affect the behavior of the system under test.

We attempted a rigorous approach where we specified requirements at a high level, dividing them into use cases and system model and then refined that model strictly to a concrete model that could then be used as basis for automated testing which would partly alleviate the problems of testing a parallel

system. Especially we planned to rely on on-the-fly testing technique to cope with parallelism.

Our experience was that maintaining the refinement between a high level model and a more concrete model that was used for testing was too laborious, thus maintaining that was abandoned. Use cases remained the only link between those two models. To measure the coverage of the use cases we initially limited the model used for testing so that it would execute according to the use cases. We also post-processed traces of random execution of the model to recognize use case behaviors. Eventually tool support allowed us to drive the model directly with use cases.

We attempted to measure the quality of the testing by comparing function call counts from an instrumented implementation that was tested by our approach and with existing "traditional" style testers, but we noted that this kind of coverage values do not reflect quality of testing. Although on-the-fly testing typically gives higher values it is easy to create a traditional test suite that produces values of same or higher magnitude by blind repetition. For same call count values the model-based on-the-fly tester is able to order the sequence of calls in more different ways, which is useful especially in uncovering errors arising from concurrency. At the same time we note that it is usually not feasible to be able to exhaustively test all possible parallel combinations, thus we propose that a way to ensure uncovering parallel errors is to drive the model with a use case to a situation with high parallelism and then letting the tester tool execute unguided after that. The produced traces can be postprocessed to obtain a measure on how many of the possible parallel executions have been covered.

The rest of the paper is arranged as follows: section 2 describes our approach, section 3 describes the system that we are testing, section 4 elaborates on the construction of the tester, including modeling and other decisions during development, section 5 describes the results of testing and finally section 6 presents our conclusions.

2 The Approach

Our initial approach is outlined in figure 1: the system *requirements* (1) are split into *use cases* (2) and an *abstract system* (3). We have chosen to design the use cases as *CSP* (4) and the abstract *domain model* of the system in B [3] (5).The B model can be checked for consistency by theorem proving and further validated against the use cases by model checking. Another view of the same thing is that the CSP can drive the B model, which acts as an oracle.

Then we *ideally* construct a concrete *reference model* (6) based on the abstract B model using the B method and refinement. The CSP can be used to drive the reference model as well so in addition to the construction method the CSP is used as another mechanism for ensuring the validity of the reference model.

The reference model is concrete enough that it contains necessary information for testing an implementation of the defined system without adding behavior outside of the model. Note that the reference model is really a model of the

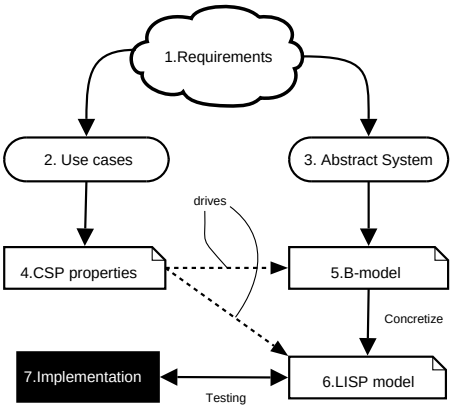

Fig. 1. The attempted process.

system rather than model of the tests, we rely on tools to be able to infer the tests from the system model. We assumed that it would be less effort to operate on the same terms as the system design rather than having a completely separate tester. Also we are especially interested in testing of the parallel features and thus we wanted to perform *on-the-fly testing* [4] under the assumption that it would be easier to have the tester adapt to the parallel behavior of the system rather than to have test cases that would contain essentially the same functionality. These assumptions led us to use the commercial Conformiq Qtronic [5] tool for testing.

3 NoTA Architecture, Session and Transport Protocol Layer

The system under test is the "high interconnect" (H_IN) part of the Network on Terminal Architecture (NoTA) [6, 7]. The driving force behind NoTA is to produce a device which supports "horizontalisation" of technology in the most extreme form currently available while adhering to the constraints of very small, embedded devices - namely, but not limited to, the mobile phone as we currently know today. The two goals that drive NoTA are modularity and service orientation.

Modularity is seen in the way devices are physically (and possibly logically) constructed while service orientation allows the functionality of the device to be abstracted away from that functionality's implementation and physical location. What this achieves is a complete separation of the functionality from the construction of the device. In other words the whole product line [8] based upon the Network on Terminal Architecture concepts becomes simply the choice of what

functionality is required and then the choice of the most suitable implementation technologies; or even possibly vice versa.

A NoTA device is constructed in a modular fashion from components which are termed subsystems. A subsystem is simply a self-contained, independent unit of the device which provides processing capabilities. Typically a subsystem manifests itself as a unit containing a processor, local memory and a set of local devices, for example, a camera, solid state storage and so on. A subsystem must also provide communication to the NoTA Interconnect allowing the services that run upon that subsystem to communicate with other services elsewhere in the whole NoTA device. Figure 2 shows this pictorially with a device containing two subsystems of various configurations.

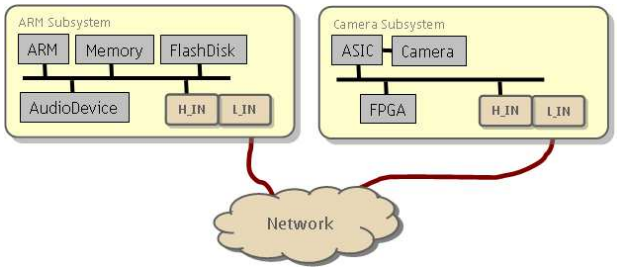

Fig. 2. Pictorial Representation of a NoTA Device

The Interconnect can be any suitable communication medium, although in a mobile device this means some kind of high speed bus. In the simplest design, the interconnect is either of star or bus topology, although any particular network topology is possible.

The interconnect is divided into three layers: High Interconnect (H_IN), Low Interconnect (L_IN) and Network (TCP/IP, MipiUnipro etc) layer.

The H_IN provides services with resource discovery and management, session and upper level transport facilities. It is into this layer that services and applications request communication channels to other services and applications, while services themselves announce and register their existence and provide functionality to the world. The communication mechanism provided by the H_IN is connection oriented and provides asynchronous message based passing and streaming data communication possibilities:

Typically the asynchronous message based type of communication is reserved for control commands and general interaction between services, while large quantities of data, eg: multimedia, are sent via the streaming connections. The L_IN is more of a device driver level which abstracts the underlying communication network away from the H_IN.

While devices are constructed out of subsystems and an interconnect, without services the device is not capable of providing any end-user functionality. The

subsystems and interconnect serve to support the provision of services. The notion of service in NoTA is an abstraction of some logical grouping of functionality [9].

4 Modeling the system and constructing the tester

There are two requirements for testing of a NoTA system that we focus on: the parallel nature of the system and testing of third party provided subsystem implementations. Each subsystem must be able to communicate with entities (services or applications) on the H_IN network regardless of which subsystem they reside on. Each subsystem may be a separate computing entity and there is no global scheduling mechanism for the whole system, which means that even if there is a deterministic behavior for a single subsystem, the behavior of all the subsystems executing in parallel most likely does not have one. We wanted to flush out bugs in the H_IN arising from this parallel complexity. Secondly we wanted to test that a third party subsystem implementation would work properly in the H_IN network.

While not covered in this work, we also wanted to be able to test the services themselves as they also may have parallel behavior and hopefully use the same approach as described in this work.

We decided to model the H_IN layer itself rather than the protocol between the subsystems or the individual services and applications using the H_IN. The reason for this was that we wanted to concentrate our effort on the system itself and then derive the possible correct behaviors from the system model automatically to cope with the expected complexity. The downside of this decision is that the system model as such treats all of the entities using H_IN the same data-wise, that is, the model of the system (or the real implementation itself) does not contain any specific information about the content of the data that it passes. When needed, we planned to provide this data by use cases or as a more drastic measure, model the particular services in the same model as the H_IN.

Also, we are modeling the system (H_IN layer) and we want to test based on the same or refined model of the *system* rather than construct a separate model of the *tester* for the system. This is because after constructing a valid model of the parallel system we want to reuse this effort in automatically deriving the tests from this model. The difference between a tester model and a system model is that the tester exists outside of the system acting as a user whereas the system model describes the behavior from the system point of view. So when a tester model sends a message, the system model expects a message. One way of thinking about this is that we want to *compare* the system model with the implementation. Here the system model acts as a reference implementation. Another view is that the tester is an "inverse" of the system.

The model communicates with the outside world by sending messages to named ports and receiving messages from them. The number of ports that allow communication to the outside world is fixed, but easily parameterizable. Each of these ports corresponds to an entity offering a service or using some service on top of the H_IN layer, so the chosen number of ports determines the maximal parallelism for the model. When the model sends a message to the outside world,

it means that we expect the system to produce that kind of message and when the model receives a message we expect the implementation to accept that kind of message.

For high level modeling of the H_IN layer we chose the B language because we had previous experience in building systems using that methodology [10] and using CSP for use cases followed from the available B tool support. Furthermore the B method has the notion of refinement which allows stepwise generation of less abstract models until B0 subset of B for which there exist mappings to imperative programming languages. We used the ProB animator and model checker [11] and the commercial ClearSy Atelier B for analyzing the B.

Also, since we expected to be testing a parallel system, we wanted to be able to perform testing on-the-fly, that is we wanted to "execute" the model in parallel with the implementation and adapt to the behavior of the implementation. It is possible to use the system model to generate a set of linear test cases, but using this mechanism to attempt testing of parallel systems is cumbersome. Firstly because attempting to generate all possible linear test cases covering all parallel interleavings becomes infeasible for nontrivial systems due to the state explosion problem. Secondly, a parallel system may execute correctly but differently than what a test case expects, which means that the test case signals "inconclusive" or "fail" and the effort spent for that test case is wasted. Finally, while typically testing languages have the possibility of branching based on replies from the implementation, constructing a test case that would attempt to adapt to the implementation would lead to implementing an on-the-fly tool using the testing language. We expect that on-the-fly testing alleviates these problems. There is existing work on generation of test cases from a model [12, 13] and also on various *test selection* heuristics [14, 15] which attempt to produce a subset of possible testing for best possible coverage of the model. We expect that the most successful of these heuristics will be applicable for test case generation as well as on-the-fly testing and that support for them will appear in tools.

The two particular requirements of the ability to base its testing on the model of the system and ability to execute testing on-the-fly led us to use the Conformiq Qtronic tool for testing. At the time of this decision Qtronic supported a variant of LISP to define the models. Qtronic executes the the LISP model symbolically in parallel with the executing system under test causing messages to be sent to the system under test. The feedack from the tested system is taken into account by the symbolic executor and influences the permissible behavior later on. The tool also has possiblity of guiding the testing by various coverage criteria: there is structural criteria, such as branch, condition and boundary value coverage over the LISP model and coverage over user specified checkpoints that are entries in the LISP model. In addition there is notion of coverage over use cases, which are also defined in LISP.

4.1 The models

The B machine contains operations for each H_IN primitive and the CSP is then used to express the use cases that specify the desired behavior of the system over those primitives. The CSP can be then verified against the B model as described in [16]. For example, figure 3 shows the B operations for registering a service

and the CSP for a use case that shows that after a service has registered under some service identifier, another entity may connect to that particular identifier. Effectively this is one possible and desired linear trace in the system.

The first two events in the CSP are internal to the used ProB tool and thus implementation details, but the `register_with_ResourceManager?Sname?Sid!REGISTRATION_OK` event is the first H_IN specific one and expresses that the particular event can occur successfully for some service id `Sid` and service name `Sname`. The register event is then followed eventually by a `connect` event that uses the same service id `Sid` to connect successfully. The `RUN` process that is executed concurrently states that all the events that are passed as its parameters do not constrain the CSP process (the list of events here is truncated). In effect we are saying that the events in the `RUN` process are ignored until the desired `connect` event with desired arguments is encountered. This same mechanism is also used to hide operations that are purely internal to the B machine, such as the `rm_register` above, so that only H_IN primitives are used to build the use cases. This way the use cases should be applicable to any system that has the same set of primitives.

```
sid,err <-- register_with_ResourceManager(nn) =
PRE
       nn : SERVICE_NAME &
      icnode_state = RUNNING
THEN
    sid,err <-- rm_register(nn,icnode_address)
END ;
```

```
ss,err <-- rm_register(nn,aa) =
 PRE
    nn : SERVICE_NAME &
    aa <: ICNODE_ADDRESS
 THEN
   CHOICE
      err := REGISTRATION_ERROR ||
      ss :: SID
   OR
   ANY
        newsid
        WHERE
         newsid : SID - rmsids
        THEN
          ss := newsid ||
          rmsids := rmsids \/ { newsid } ||
          err := REGISTRATION_OK
        END
   END
 END ;
```

```
CSP:  MAIN = initialise -> notify_resource_manager_location -> REGISTER;;
      REGISTER = register_with_ResourceManager?Sname?Sid!REGISTRATION_OK ->
                 (CONNECT ||| RUN[register_with_ResourceManager,send,nm_register,...]);;
      CONNECT(SS) = connect!SS!CONNECT_OK -> skip;;
```

Fig. 3. The "register" primitive expressed in B and a CSP property.

By the time we completed this model, the specifications had already changed somewhat and the initial test model had turned out to be complex enough that we felt that constructing the chain of refinements between these models would be too time consuming, especially as this process might have to be repeated. Thus we decided to develop the models separately but nevertheless make an effort to make sure that the CSP use cases would be compatible with both. We felt that it was still worth the while to continue, as a correct - in terms of

verification - system does not guarantee that the system would have behaved in accordance with the customer's wishes - hence the need for testing at all levels of development.

It can be argued that correct development of the system would have been achieved if we had followed the refinement rules and constructed a concrete specification in the B0 subset of B which is translatable to a 'normal' imperative language (B0 sequentializes actions and removes non-determinism). There do exist code generators from B0 to C, C++ (and also Java and Ada). We faced three problems here, firstly the time spent developing and refining the specification would have been considerable and secondly we would have to have developed a code generator to LISP and ensured that it preserved the semantics of the model. Finally we felt that there would certainly be changes that would result from connecting the LISP system to the tester and all of these changes could not be done at the LISP level but higher up in the refinements chain. These facts together outweighed the potential benefits - of course if B (and it has now been superseded) would have been taken into more general use then this route might pay off in the future. Additionally, strict refinement based approaches do not cope with change in the specification well resulting in techniques such as retrenchment to preserve the mathematical link between now differing specifications.

The LISP model was constructed essentially as an event loop: the system reads in messages from incoming ports and these events are then processed. These events match the H_IN primitives. Notably, initially all H_IN events were accepted by the event loop. Since this model is used for testing this meant that given this model to a model based tester, it would generate any of these primitives to be sent to the implementation. The effects of this are described further below in section 4.2.

Figure 4 shows the LISP code that corresponds to the "register" primitive in figure 3. This function is called from the event loop after an event `Hactivate Service` with one parameter `sid` of integer type has been received.

If the parameter `sid` has been registered before (1), the event `HactivateService_ret` is sent back with parameter value zero signaling failure (3) and the event loop is re-entered. The model has been augmented with a tool specific checkpoint mechanism (2), which allows tagging parts of the model so that the tags are reported in test traces, but they can also be used as coverage guides. If the `sid` is new, it is then associated with the port from which the original message came from (4) and a new internal interface is created (5) and also associated with the `sid` (6) for later use. Then a random value non-zero is generated (7) and after some more bookkeeping and checkpoints (8), the return message is sent back (9) containing the random value. Again, since this is a model of the system, emitting a message with a random value means that the tester expects the implementation to produce that message with any value in place of the random value.

Each H_IN primitive has a similar function and furthermore there are functions for bookkeeping and internal data structures as well as functions whose purpose is to enforce the typing of the primitive parameters.

The use cases are also written in LISP and they are similar to the CSP ones in that they are essentially sequences of events. Figure 5 shows the same use case as earlier in figure 3: the `main` function calls three functions, that perform the

```
(define HandleActivateService
  (lambda (msg env in_port ret_port)
    (let*
        ((msg_name (ref msg 0)) (sid (ref msg 1))
1        (known_sid (r_known_sid? (env_rmap_port env) sid)))
         (if (known_sid)
           (begin
2            (checkpoint (tuple 'unable-to-register sid)) ; a named checkpoint
3            (output ret_port (tuple 'HactivateService_ret sid 0)) ; sending a message
             (h_in_router env))
         (begin
4              (r_dict_add (env_sid_to_outport env) sid ret_port)
5              (let ((sid_port (make-interface)))
                 (begin
6                  (r_dict_add (env_rmap_port env) sid sid_port)
7                  (let ((pid (+ (random 254) 1))) ; make up a value
                     (begin
                       (r_dict_add (env_s2p_port env) sid pid)
                           (env_incr_service_ctr env)
8                          (checkpoint (tuple 'service-activated (print-pname in_port) sid))
9                          (output ret_port (tuple 'HactivateService_ret sid pid))))
                     (h_in_router env))))))))
```

Fig. 4. The LISP code corresponding to the "register" primitive.

communication. This use case is expected to be running as an observer in parallel with the system model. The way this use case either influences the testing or adapts to it is elaborated below.

4.2 Use cases, data and control of the system

As mentioned earlier, the model of the system contains no information about the content of the data it is handling, the only properties the model deals with is the size of the data. This is consistent with the B model where the structure of the data was abstracted out using a generic `DATA` type.

For instance, a camera service has registered itself to the system under the name CAM. In order to use that service, an application must know that name and furthermore be able to send across correct commands, potentially in a certain order. For H_IN the name of the service is any sequence of characters with a maximum length that is associated with some port. If someone asks for that particular string then H_IN can freely choose a number within some bounds to represent a connection to that service and then transmit data, which from H_IN point of view is a sequence of items in a buffer.

For theorem proving and model checking purposes this generality is not a problem and also if the H_IN can be tested in a test bench where no real world entities are running it is possible to use essentially random data. However, if there is a service implementation in the network, it becomes necessary to know its identity and also how to maintain communication with it by sending compatible messages in right order. So there should be a mechanism of dealing with the data.

```
(define send-hactivate
  (lambda ()
    (let* ((oport (any-oport)) (sid (random 254)))
      (begin
        (output oport (tuple 'HactivateService sid))
        (tuple oport sid)))))
```

```
(define send-connect
  (lambda (sid)
    (let ((oport (any-oport)))
      (begin
        (output oport (tuple 'Hconnect_req sid))
        #t )))) ;; we have reached our goal
```

```
(define receive-hactivate_ret
  (lambda (send-port sid)
    (let ((receive-port (oport-to-iport send-port) ))
    (let ((inmsg (handshake receive-port #f)))
      (let* ( (iport (gp inmsg)) (msg (gm inmsg)))
        (begin
          ;; We make sure that message is what we want
          (require (equal? (ref msg 0) 'HactivateService_ret) #f)
          (require (equal? (ref msg 1) sid) #f)
          (require (> (ref msg 2) 0) #f)))))))
```

```
(define main
  (lambda ()
    (if |usecase:use case 2|
      (let ((a (send-hactivate)))
        (let* ((s_oport (ref a 0))
               (sid (ref a 1)) )
          (begin
            (receive-hactivate_ret s_oport sid)
            (send-connect sid)))))))
```

Fig. 5. The LISP use case corresponding to the B use case.

Also, the system is built with a certain purpose in mind: H_IN should connect entities and transport data between them. The information how to do this is contained in the model, but the sequence of primitives that performs the data transfer is just as probable as any other sequence. For example a valid sequence of primitives always starts with a "register" primitive and is then followed by a "connect" primitive, but the model may always enable sending a "send" message with bogus arguments first. Of course this is due to the way the model has been constructed: all primitives should always be accepted. This generation of unexpected behavior is partly the reason why model based testing is powerful, but it also has the risk that the intended behavior is never completely exercised as there is always a high possibility of choosing a bogus message.

Both the data issue and sequencing of primitives can be resolved by modifying the model. The expected data can be hard-coded in the model so that whenever a message is to be sent or received from a given address, it has the desired format. The major downside of this is that this is completely bound to the particular application. The sequencing can also be enforced in the model: a boolean flag in the event loop can ensure that a "send" can only occur after a successful "connect". This is more general than the data hard-coding but it seems clear that if more complex sequences than two messages are considered, the system quickly becomes complex.

Both of these clearly limit the state space of the system so it might be possible to consider these models a refinement of the "generic" model, but the loss of generality is unappealing.

To solve this we put the application specific information to use cases and then count on the testing tool to be able to utilize the information in them. The use cases are used in two ways: to influence the test execution and to observe the test execution.

Figure 6 gives an example of both cases. This is part of the use case shown in figure 5 and on the left hand side of figure 6 the use case adapts and observes

```
(define send-hactivate
  (lambda ()
    (let* ((oport (any-oport)) (sid (random 254)))
      (begin
        (output oport (tuple 'HactivateService sid))
        (tuple oport sid)))))
```

```
(define send-hactivate
  (lambda ()
    (let* ((oport port1) (sid 12))
      (begin
        (output oport (tuple 'HactivateService sid))
        (tuple oport sid)))))
```

Fig. 6. A use case that observes and another that influences.

the test run, here the `HactivateService` primitive that is used to register the sender of this primitive under the id given as parameter. There are parameter values that are both chosen randomly, the port where the message occurs and an integer variable corresponding to the service id. As we explained earlier, the random value indicates that any value produced by the test tool can occur here. So this states that we want to be able to observe primitive `HactivateService` for any port and service id and that we want to remember both for later use. The version the right hand side of figure 6 is similar, but both the port and service id have fixed values; we expect this use case to influence the tester tool so that it is able to generate the values given in the use case. For payload data, this is the mechanism we force the service commands to comply with possible existing systems.

For control flow, we can explicitly guide the implementation event by event, but we'd prefer to encode the same use case as in figure 3 and let the tester tool find the path to such a state. This allows us to guide the implementation to a state of high concurrency or otherwise unlikely situations and then let the tester tool proceed (semi-)randomly. Unfortunately the tester tool did not support this kind of use cases at the time so we could not try this approach. We expect this feature to appear in future. The potential downside of this feature is that it might require heavy computations, which are problematic when performing on-the-fly testing.

While not obvious from the use case listings, we expect that tool support will remove the work needed to write the use cases as code. The primitives for communicating with the implementation can be derived from interface specification, resulting in a user interface that allows specifying the use cases in an MSC-like format. The user only needs to fill in the sequence of actions and constraints on the data values.

4.3 Test configuration

There are multiple parallel entities that operate over the H_IN layer, but the H_IN layer itself may cover several subsystems as described in section 3, so there needs to be a way of connecting the tester tool to all of them. The model considers H_IN as a single system: there may be several points of communication, but they all reside on the same H_IN layer and are independent of each other.

In practice each entity on the H_IN layer must reside on some subsystem. Also, every subsystem must implement part of the H_IN network that offers primitives to the users and is able to communicate to other similar subsystems within

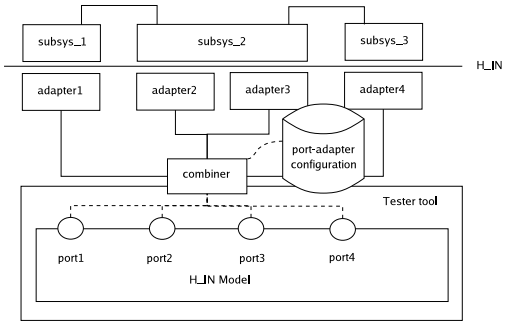

Fig. 7. Test configuration for a H_IN system consisting of three subsystems.

the H_IN network. There exists a C-language interface for these primitives and when model communicates with the outside world via port, this communication in terms of the tool run time system must be mapped to these C data types and function calls and back. Our aim was to construct the model used for testing at detailed enough level that we could claim that no information loss or information generation would be needed in these adapters. The only part where we are not entirely convinced about meeting this goal is the handling of asynchronous messages, which required its own bookkeeping mechanism.

Qtronic has the notion of an "adapter process" which communicates with the tester tool using a specific protocol, in this case over TCP/IP, and contains the user produced adaptation between the implementation and the Qtronic run time system. We have decided to make each adapter process correspond to one model port, rather than have one adapter process for each subsystem. This gives more flexibility in at the price of more overhead, which may be a problem for subsystems with low processing capabilities.

The traffic between different adapter processes is routed via a special "combiner adapter" that is configured with the address information of the port specific adapters. This configuration has to match the real world subsystem configuration. Figure 7 shows an example configuration, where the model has four ports and the system under test consists of three subsystems. Two of the ports have been mapped on the same subsystem, while other two are mapped to two different susbsytems.

We assume that we are able to execute the adapter processes on the subsystems, which is a valid assumption for in-house testing. However, if we want to test a subsystem implementation we have two possibilities: either demand the producer of the subsystem that there is an access to H_IN or then we must be able to exercise the service that exists on the subsystem using its own primitives. Use cases that contain the service specific data is one possibility, another one is to model the service in addition to the H_IN model.

5 Results and Conclusions

The abstract B model has 800 +/- 100 lines of code, the LISP model has 2100 +/- 200 lines of code and the C implementation of the H_IN has 20000 +/- 1000 lines of code. As usual, the modeling phase already uncovered some errors and assumptions in the implementation. The modelling process (including requirements elicitation and revision, plus various versions of the model) took 4 man months to produce the first major release of the specification and the first feature complete tester also took about 4 man months.

During testing, one bug was found that essentially was due to the implementation not expecting out-of-band messages, for example a "send" before a "connect" had been made. These could have been found by writing a tester that would have produced messages randomly.

We found four bugs that were of a concurrent nature, the earliest one is shown in figure 8, where two services on same subsystem register themselves and then an application tries to connect to both producing a connect request only for the other one. This trace is the shortest one to that error.

Another error of a similar kind, occurred with the same configuration where the indication of arrived data was given to the wrong entity. The third error was related to closing down of connections when two entities on different subsystems closed down the connection at same time and the fourth error occurred when two connections were sending data in both directions requiring four entities where connections were across subsystems. Furthermore we found errors that had no concurrent cause and most likely would have been caught by any kind reasonable testing. Typically the faults were such that they manifested themselves as multiple reported errors and identifying the root cause took some human work. Also, at times it was necessary to modify the model so that these errors could be circumvented and testing could be continued.

The implementation under test had simple test programs that set up a service and then a client would connect to that service to transfer simple data and this could be repeated. There were three scenarios: client connects and sends data, client connects and receives data and client connects and both client and service send and receive data.

Our assumption was that model-based testing would be able to exercise the system better and that our approach would be more efficient when compared to taking the "traditional" approach of writing linear test cases or test programs, especially for errors that arise from concurrency.

Bugs were found and the mentioned parallel errors were such that save for the first one there were no explicit requirements that would have led to test cases uncovering the errors. In the first case, the requirement existed, but there were multiple potential configurations of subsystems which were not explicitly noted down. We feel that it is unlikely that there would have been hand written sequential test cases that would have caught these errors. Furthermore, we note that they would not have been repicated by repeated executions of the existing test programs.

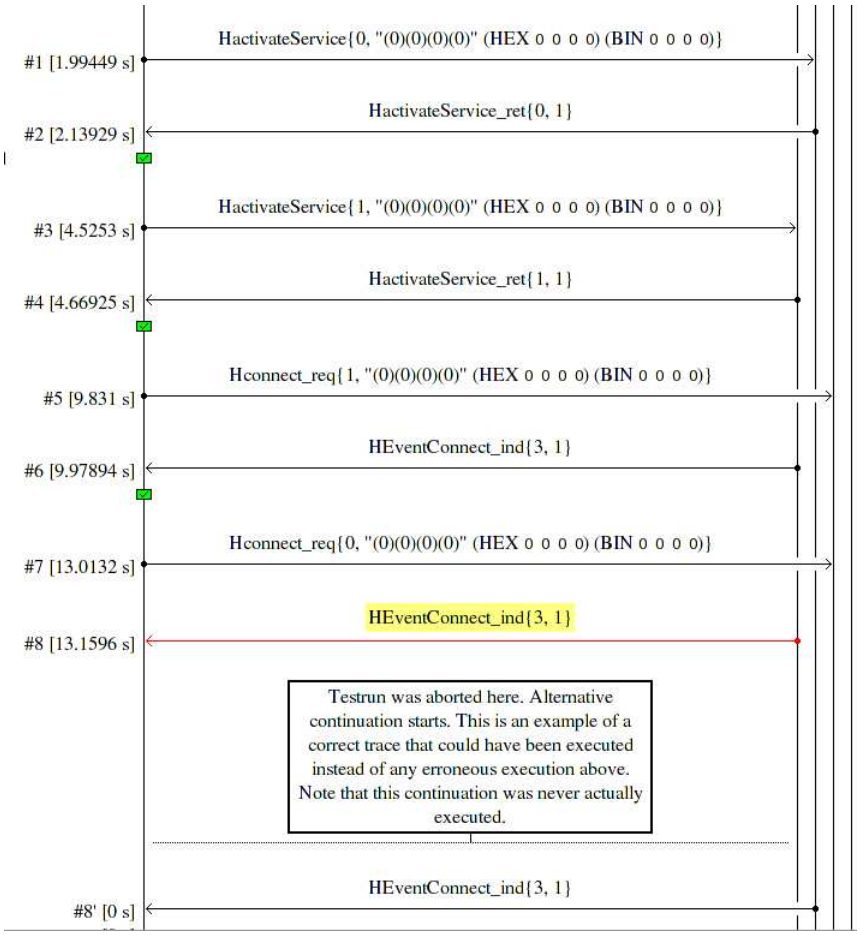

Fig. 8. An error trace involving three entities.

However measuring the parallel goodness of the testing is not straightforward as complete coverage is most likely not going to be achieved. Given this, would it be possible to identify the part that was not tested and guide later testing to cover that?

We used `gprof` utility to obtain call information for C functions for the individual H_IN components on subsystems, but it seemed that while the results of the longer on-the-fly produced more coverage data, this could always be matched simply by running the existing simple tests more times. It may be that a coverage analysis with smaller granularity than function level is needed to note the changes. However, the C implementation relies on threads and callbacks, which means that the branching is not detectable on the C-code level. It seems that measuring the parallel quality of the testing based on this kind of metrics is not good enough.

Another possibility we considered is using the traces produced by testing to deduce how much of the potential state space has been exercised. We implemented a prototype tool that takes in a description of a use-case and then attempts to show how many of the potential parallel executions of those were seen in the testing. Other approach would have been feeding the traces back to the B model checker and obtaining relevant information there. However, both of these approaches required tool development for which we didn't have resources.

Yet another way is to add the desired information to the model or the use cases. The Qtronic test tool has its own coverage criteria which aims to cover the model as well as possible and as the model has been constructed so that observable events are part of the coverage, the tool is able to produce meaningful testing and report checkpoints that may have metadata associated with them. It is possible to modify the model with auxiliary constructs that keep track of its own parallel state and produce that as output. The downside here is that another model is embedded in the model of the actual system.

Our preferred approach would have been to drive the model to a desired state with high concurrency using a use case and then let the tester tool proceed with a random walk. Unfortunately the tool support for this feature was not available at the time.

Our approach managed to uncover errors that would otherwise most likely not been found, but at the price of creating the system essentially twice. Constructing the models is a different skill when compared to writing test cases and this may be the greatest obstacle in adoption of this kind of testing. Nevertheless the possibility of using use cases to drive the model may be useful in demonstrating the value in terms that are understandable to traditional testers. The tool support is nearly there to allow the use cases to act as a loose template for test execution which would allow the test engineer to write testcase-like constructs to exercise the implementation.

We did not set out to do comparisons with other existing approaches and tools, rather we were looking for experiences in combining components in a toolchain. There are alternative approaches for both the specification and the tester tool side, especially ToRX [17] and the Spec Explorer [18], but we did not evaluate these.

It is inevitable that errors, especially of a concurrent nature, are introduced during development through decisions (primarily architectural) made while implementing. In addition we see errors introduced through requirements change which can not be adequately modelled and verified at the more abstract levels of modelling. Even though we have had to take a pragmatic approach which has compromised some "formal methods ideals" we have seen our approach uncover errors earlier and provide more detail about those errors.

6 Acknowledgments

This work has been made in cooperation with the EU Rodin Project (IST-511599). We would like to thank Kimmo Nupponen and Antti Huima from Conformiq for valuable help.

References

1. Erl, T.: Service-Oriented Architecture. Prentice Hall (2005) 0-13-185858-0.
2. Oliver, I.: Experiences in using B and UML in industrial development. In Julliand, J., Kouchnarenko, O., eds.: B. LNCS 4355, Springer (2007) 248–251
3. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
4. Vries, R.d., Tretmans, J.: On-the-Fly Conformance Testing using SPIN. In Holzmann, G., Najm, E., Serhrouchni, A., eds.: Fourth Workshop on Automata Theoretic Verification with the SPIN Model Checker. ENST 98 S 002, Paris, France, Ecole Nationale Supérieure des Télécommunications (1998) 115–128
5. Conformiq Software Ltd.: Conformiq Qtronic, a model driven testing tool (2006–2007) http://www.conformiq.com/qtronic.php.
6. Suoranta, R.: New directions in mobile device architectures. In: Ninth Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006), 30 August - 1 September 2006, Dubrovnik, Croatia, IEEE Computer Society (2006) 17–26
7. Suoranta, R.: Modular service-oriented platform architecture - a key enabler to soc design quality. In: 7th International Symposium on Quality of Electronic Design (ISQED 2006), 27-29 March 2006, San Jose, CA, USA, IEEE Computer Society (2006) 11–13
8. Savolainen, J., Oliver, I., Mannion, M., Zuo, H.: Transitioning from product line requirements to product line architecture. compsac **01** (2005) 186–195
9. Kruger, I.H., Mathew, R.: Systematic development and exploration of service-oriented software architectures. In: Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture WICSA 2004. (2004) 177– 187
10. Kronlof, K., Kontinen, S., Oliver, I., Eriksson, T.: A method for mobile terminal platform architecture development. In: Proceedings of Forum on Design Languages 2006. Darmstadt, Germany. (2006)
11. Leuschel, M., Butler, M.: ProB: A model checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods. LNCS 2805, Springer-Verlag (2003) 855–874
12. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE. Volume 84. (1996) 1090–1126
13. Gnesi, S., Latella, D., Massink, M.: Formal test-case generation for uml statecharts. iceccs **00** (2004) 75–84
14. Feijs, L., Goga, N., S., M., Tretmans, J.: Test Selection, Trace Distance and Heuristics. In Schieferdecker, I., König, H., Wolisz, A., eds.: Testing of Communicating Systems XIV, Kluwer Academic Publishers (2002) 267–282
15. Pyhälä, T., Heljanko, K.: Specification coverage aided test selection. In Lilius, J., Balarin, F., Machado, R.J., eds.: Proceeding of the 3rd International Conference onApplication of Concurrency to System Design (ACSD'2003), Guimaraes, Portugal, IEEE Computer Society (2003) 187–195
16. Leuschel, M., Butler, M.: Combining CSP and B for Specification and Property Verification, Springer-Verlag, LNCS 3582 (2005) 221–236
17. Tretmans, G.J., Brinksma, H.: Torx: Automated model-based testing. In Hartman, A., Dussa-Ziegler, K., eds.: First European Conference on Model-Driven Software Engineering, Nuremberg, Germany. (2003) 31–43
18. Veanes, M., Campbell, C., Schulte, W., Tillmann, N.: Online testing with model programs. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2005) 273–282