Model-Based Approaches for Validating Business Critical Systems

J. Augusto, Y. Howard, A. Gravell, C. Ferreira, S. Gruner, M. Leuschel,
Declarative Systems and Software Engineering Research Group,
Department of Electronics and Computer Science,
University of Southampton, S017 1BJ Southampton, UK
Email: {jca, ymh, amg, cf, sg, mal}@ecs.soton.ac.uk

***Abstract*— Developing a business critical system can involve considerable difficulties. This paper describes part of a new methodology that tackles this problem using co-evolution of models and prototypes to strengthen the relationship between modelling and testing.**

We illustrate how different modelling frameworks, Promela/SPIN and B/ProB/AtelierB, can be used to implement this idea. As a way to reinforce integration between modelling and testing we use model-based tests and trace-driven model checking. As a result we were able to anticipate problems and guide the development of our software in a safer way, increasing our understanding of the system and its reliability.

I. Introduction

A business-critical system is a software, or software/ hardware, system whose correct operation is crucial to a business or enterprise. Developing such systems involves dealing with considerable difficulties such as distributed systems that interact via a/synchronous communication, and security issues (see for example [1] and [2]). On the other hand, product development constraints and experience strongly suggest that a component-based approach is useful and that more agile methodologies should be used to produce software in the area. The Declarative Systems and Software Engineering Research Group is investigating a novel combination of methods for developing software focused on these particular features.

One of the main aims of the research project is to encourage the development of reliable software by using formal methods in industry and business oriented companies. Some of the activities carried out by the group that were oriented towards the research aims consisted of exploring different ways to use and extend existing tools as well as exercising different methods that can increase reliability on the software production process. We report here on how, by extending and modifying the way we use existing tools and methodologies for validation, we successfully improved the outcome of the validation process.

The validation process we are reporting about is focused on the design of models, tests and their interaction. We provide some details about all these interacting parts in section II. We used several case studies to explore our ideas, in section III we describe those that have been developed in more detail: the travel agency and the mortgage broker case studies. Some of the modelling frameworks for the validation process are Promela/SPIN ([3], [4]), B/ProB/AtelierB ([5], [6], [4], [10]), CSP/FDR ([7], [8]), CSP(LP)/CIA/XTL ([9], [11]), and StAC/CIA/XTL ([12], [11]). However we will just exemplify our proposal based on the two first frameworks of the list. See sections IV-A, IV-B and IV-C for an account on how we use these modelling frameworks to successfully validate different aspects of our systems.

We illustrate how models are very helpful to guide the development and validation process. They can be "executed" through animation/simulation, and can be comprehensively checked, at least for specific configurations, by model checking. Here we give a brief description of some of the successful applications of these tools but particularly on how emphasizing the interaction between modelling and testing has proved useful in guiding the development process on critical aspects and to increase the confidence in our system.

A brief description of other case study we considered (section V-A), the component-based side of our work (section V-B), and some lessons learnt and future work (section VI) are included at the end.

II. Modelling, testing and their relation

As part of a much broader methodology we are exercising different ways to use modelling techniques and the way that models relate to the rest of the software development process. From a methodological point of view, new ways to relate the model with the prototype were exercised: new terms like *trace-driven model checking* and *model-based testing* were incorporated to our technical jargon. They are now part of our new approach to develop business critical systems. These methodological considerations have shown to be useful uncovering subtle errors when the systems became more complicated and the chances of having hidden subtleties both in the model or the code increased.

On one side we take full advantage of existing tools by exploiting their simulation and model checking facilities in the traditional way. We also propose novel uses of those tools so that we can use a model as a continuous guide to the developing process, aiming to minimize corrections and revisions in the implementation. So we favour a more integrated relation between models and prototypes that can co-evolve in a synchronized way.

Our models are in widely used notations that have defined semantics and tool support. These notations are capable of dealing with essential notions for e-business applications like

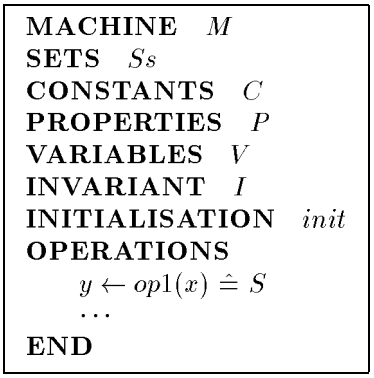

Fig. 1. Abstract machine structure

concurrency and synchronous/asynchronous communication. These modelling frameworks allow the creation of simple and abstract models that can be simulated and rigorously checked.

We used our models and tools as a guide to focus from the very beginning on critical issues: e.g. absence of deadlocks, interaction, message passing and synchronization in a/synchronous communication scenarios. Initially we used SPIN and ProB for simulation at early stages of the system while later on we made a more sophisticated use on verification by using SPIN and AtelierB for property checking.

A. Promela/SPIN

SPIN [3] has been a particularly successful tool that has been widely adopted for performing automatic verification of software specifications. SPIN offers the possibility of performing simulations and verifications. Through these two modalities the verifier can detect absence of deadlocks and unexecutable code, check correctness of system invariants, find non-progress executions cycles and can verify correctness properties expressed in propositional linear temporal logic formulae.

Promela is the specification language of SPIN. It is a C-like language enriched with a set of primitives allowing the creation and synchronization of processes, including the possibility to use both synchronous and asynchronous communication channels. We refer the reader to the extensive literature about the subject as well as the documentation of the system at Bell Labs web site for more details: `http://netlib.bell-labs.com/netlib/spin/whatispin.html` Due to its widespread use we assume some degree of familiarity with this framework from now on.

B. The B Model

B AMN is a model-oriented formal notation that is part of the B-method developed by Abrial [5]. In the B-method, a system is defined as an abstract machine which has the structure presented in Figure 1. The sets clause presents user defined sets that can be used in the rest of the machine; those sets can either be enumerated or deferred. The properties are used to define logical properties of the constants or sets of the machine. The variables describe the state of a machine, they are described using set-theoretic constructs, as for example, sets, partial functions, and sequences. The invariant is a set of first-order predicates. The invariant provides typing constraints for the variables of the machine, *e.g.* $basket \in CLIENT \rightarrow \mathcal{P}(BOOK)$, and may also include logical properties that must be preserved by the variables. The initialisation describes the initial values for each variable of the machine. The initialisation must establish the invariant. Operations act on the variables while preserving the invariant and can have input and output parameters. Initialisation and operations are written in the generalised substitution notation of B AMN, which includes constructs such as assignment, guarded statements, and choice. In the assignment statement $x := E$, x is a variable and E is an expression that may use any of the available variables. Simultaneous assignment $x := E \parallel y := F$ is equivalent to $x, y := E, F$. In the guarded statement **SELECT** G **THEN** S **END**, the guard G is a condition on the state variables and S is an AMN statement. This statement will be enabled only when G holds. The nondeterministic choice between two statements is written **CHOICE** S **OR** T **END** that will be enabled when either S or T are enabled. The unbounded choice **ANY** x **WHERE** P **THEN** S **END** nondeterministically chooses some value x satisfying P and then behaves like S. This is a subset of the language, as we only presented the AMN constructs that will be used through this thesis. The B method has two supporting tools, *Atelier-B* [6] from Steria and *B-Toolkit* [13] from B-Core.

ProB [Leuschel01], [Leuschel03] is a new tool that provides animation, visualisation and model-checking for B machines. It is implemented in SICStus Prolog, exploiting advanced logic programming features such as co-routining and constraint solving. This gives performance comparable to more mature tools such as FDR and Spin. The input for ProB is the XML encoding provided by Bruno Tatibouet's jbtools B-to-XML parser. Animation (or interactive state exploration) uses a TCL/TK interface, and the dot open source graph drawing software. Model-checking (or exhaustive state exploration) can be used to check, for example, that a machine's invariant is not violated. ProB is currently still under active development. Improvements planned include the generation of test-cases from B machines, and model-checking for temporal logic properties and infinite state systems.

III. Some Case Studies

We considered several case studies in order to test the methodological aspects of our proposal. Typical scenarios like *order fulfilment*, *E-bookstore*, *travel agency*, and *mortgage broker* were considered. In the following sections we describe those that have been given more attention as a way of exercising all aspects of our methodological proposal.

A. The Travel Agency Case Study

One of our case studies is a travel agency that offers travel booking services to its customers via the Internet. A

customer logs on to the travel agency welcome page, chooses the booking service they require (a hotel room reservation, or a rental car etc) and offers the credit card they will use to pay for the service when booked. The travel agency contacts appropriate suppliers and attempts to book the customers requirements, reporting back to the customer what has been booked on his behalf.

This case study is interesting for a number of reasons; it has components (travel agents, hotels and car rentals) that are distributed over the internet. There are many possible instantiations of the network of components. Each of the participating components has its own version of the booking data and yet the versions need to be consistent. And lastly, the case study is an e-business application and therefore uses the services of a complex layer of 'middleware'; web servers, object databases and java component technology through offered interfaces.

A typical instantiation has a number of travel agents, hotels, car rentals and customers. From the customer's perspective, they only interact with the travel agent, the suppliers of their hotel rooms or rental cars are hidden. Suppliers only see the travel agent; they have no direct contact with a customer. From the travel agent's perspective they can see their customers and their suppliers; however bookings made with suppliers by other travel agents are hidden. The distributed nature of the travel agency and the complexity of the possible interactions means that it is not easy to test.

There are business critical features of the system; we are asking for a credit card payment, we want to be sure that a reservation for a customer recorded by the travel agent is also recorded by the supplier of the service, and we want to preserve commercial privacy of the information.

We implemented the case study as a web application for Apache Tomcat web servers using Java Server Pages (JSP's), Java Servlets and Java Database Connectivity (JDBC). Booking data for the travel agent and supplier components was stored in Microsoft SQLServer2000 DBMS via JDBC, which also provided a 'name discovery' service for the interacting components to locate the suppliers of services they need. The complete Promela model, fully documented, can be seen as an appendix to [14].

B. The Mortgage Broker Case Study

This case study is currently under development and although it shares some basic structure with the Travel Agency case study, the protocol by which transactions are completed is different. Another very important distinction is that while for the previous case study we enforced a synchronous communication hypothesis, for this case we wanted to exercise our development methodology by considering asynchronicity as the underlying communication paradigm.

The mortgage broker scenario we considered consisted of a user interface that allows customers to make requests for a mortgage to a mortgage broker on the basis of salary, amount to be borrowed and location of the property to be bought. The location is passed to several insurers that will assess the risk of insuring that property and, possibly, offer themselves as insurers for that particular mortgage request. Other data like the salary and the amount that is being requested is passed to several lenders that can potentially offer themselves as lenders.

Although the scenario can be initially shortly described there are quite a few logical subtleties that have to be considered. For example, the procedure of making an application and guaranteeing an answer, which at the same time should be consistent between all the parts in a distributed environment, is a well-known source of problems. How can we ensure that the answers to different requests do not get mixed? How can we ensure that an acknowledge request is answered and the answer reaches the intended destination when in the middle there could be disruption of services?.

We opted to simplify the protocol to the following extent. A typical session will demand that the user, by using a browser, apply for a mortgage. The user uses a browser to login in the system and make the request so this is a synchronous communication. For each new request the mortgage broker will broadcast the application to a network of insurers and lenders. Each of them periodically will check for possible new applications. After evaluation they will ignore it or will make an offer. The model broker will periodically check also for insurers' and lenders' offers. These offers in turn will be passed to the user as they are collected. Each time the user check the virtual basket, s/he will be presented with the list of insurers and lenders offering their services for a particular mortgage application. The user will have the opportunity of selecting offers provided that the choice comprises a pair insurer-lender. This selection is passed by the mortgage broker to those service providers that have made an offer to tell them if their offers have been accepted or rejected. Insurers and lenders cannot keep on hold their resources indefinitely so they will be forced to impose some deadlines on their offers. Hence, by the time they are contacted by a mortgage broker about their successful application they probably already dismissed that offer so that they can use the released resources to make offers to other customers.

There are plenty of interesting combined behaviours to explore under this settings. For example, a lender can probably timeout one offer to make the resources available again but in principle it could be the case that will always repeat a request for those same customers. Detecting that will motivate the decision to prevent it or not and in the first case will trigger the question of how to do it, e.g. by keeping record of customers' accept/decline history.

Lenders and Insurers have their own strategies to decide under which conditions they should offer services and resources. These possible behaviours have to be mixed with the possible reactions from the user, in order to maximize their investments. The combination of possible behaviours to be allowed/implemented is so interesting as difficult. As we mentioned above we made several simplifications in order to keep the case study interesting and close enough to reality but, at the same time, with a level of complexity such that can be developed within the usual tight time slots assigned to research experiments.

IV. Synchronizing Modelling and Testing

Keeping models and prototypes close together demands inter-leaving of modelling and prototyping as opposed to developing a complex model that then has to be followed by the subsequent implementation discovering mismatches between idealized abstractions with actual tools available.

In this section we use Promela/SPIN and B/ProB/AtelierB to illustrate the advantages of incorporating this synchronization idea into the development process and the possibilities that each one provides for implementing such a step. The relation between modelling and testing can be explored in two obvious directions:

1) using the model to generate information that should prove consistent with the prototype behavior when they are transformed into test cases.
2) using the prototype to generate traces that can be then model checked. We explored this idea in two ways by transforming traces into sub-models and temporal formulas to be checked.

All these options are usually subject to a possible intermediate step for renaming traces to bridge notational differences, e.g. names for storage structures as used in languages with different levels of abstraction.

A. Using SPIN to Generate Tests Traces

There are plenty of suggestions in the literature to guide the process of generating tests cases to test a prototype or implementation. Here we focus in some options for automatic generation of tests. Consistently with our proposal, we let the model to guide the generation of tests so that we can, not just test correctness in the prototype, but also synchronicity between the model and the prototype. We have found at least two ways to inform the prototype by using the model: by filtering simulations and by transforming temporal formulas into test cases.

1) Extracting Tests from Simulations: SPIN allows users to simulate the behavior of the systems modelled in several ways: a) randomly generated b) user-guided (interactive), and c) system-guided (by using an error trail after model-checking).

Simulations can be saved, so we complement our models with printouts whenever a key choice is taken in the model. These printouts act also as comments for the model and are practically harmless in computational terms as they are considered at simulation time but they are not considered by SPIN when doing model-checking. The simulation can be then saved and filtered by using a simple program. The filtering process can be tuned by taking a "dictionary" as a complementary input providing translation to accommodate the differences between the model and the prototype terminologies. As a result we extract a set of steps that can be used as a test to verify that going through those steps in the prototype will result into exploring the same, or at least equivalent, states and conditions than in the model. The following trace gives the description of part of a simulation where two users login on the system by using different browsers to book cars in different car rentals. Notice that their processing is interleaved.

```
(-->) loginID: 1
(-->) Credit Card MC provided by user
      with loginID: 1
(-->) Selection made by the system:
      book a car for User:1
(-->) loginID: 2
(-->) Shop informs: Successful request
      to book a car in CarRental00 for
      User :1
(-->) TA informs: Your booking of a car
      was successful from CarRental 00
      for User 1
(-->) Credit Card VISA provided by user
      with loginID: 2
(-->) For IndexTravelAgencyDB=0
      taDB[IndexTravelAgencyDB]=dbid,
      with dbid=1
      taDB[IndexTravelAgencyDB].shopID1=
      dbshopID1, with dbshopID1=0
      taDB[IndexTravelAgencyDB].shopID2=
      dbshopID2, with dbshopID2=0
(-->) TA is redirecting request for
      booking a car to Car Rental 00,
      request was made by user:1
(-->) Selection made by the system:
      book a car for User:2
(-->) Shop informs: Successful request
      to book a car in CarRental01 for
      User :2
(-->) TA informs: Your booking of a car
      was successful from CarRental 01
      for User 2
(-->) For IndexTravelAgencyDB=1
      taDB[IndexTravelAgencyDB]=dbid,
      with dbid=2
      taDB[IndexTravelAgencyDB].shopID1=
      dbshopID1, with dbshopID1=0
      taDB[IndexTravelAgencyDB].shopID2=
      dbshopID2, with dbshopID2=1
(-->) TA is redirecting request for
      booking a car to Car Rental 01,
      request was made by user:2
```

An equivalent sequence of steps can then be followed in the prototype to test if the behaviour obtained is correct and compatible with the one observed in the model.

2) Temporal Logic Properties as Tests: We used temporal logic properties as a way to emphasize requirements. Some properties checked in the Promela Model by using PLTL temporal logic formulas were:

> *"If a car is requested and there is a car available one is eventually booked"*
>
> *"If a car has been booked and its unbook is requested then it is eventually unbooked"*

"If a car is requested and there is no car available the request is rejected"

"If unbooking a car is requested and has not been booked the request is rejected"

"Booking do not produce unbookings and viceversa"

"It is true that if a car is booked that is what the user gets"

"Never a car is booked to more than one customer at the same time"

"All recorded bookings for cars and rooms are also recorded in the Travel Agents' DB"

"No record in the Travel Agents' DB is made if before it was not recorded in CarRental's and Hotel's DB"

"If a car is booked by user N and s/he request another car, and there are cars available in both shops then the previous shop will be selected"

"No operation goes ahead without Credit Card approved"

"The credit card check eventually finishes"

"No resource is booked for ever"

"A user cannot operate for another user"

"Is not true that first come first served"

EXAMPLE *1:* consider we want to check the following property: *if a car is requested and there is no car available the request is rejected.*

If we want to check the property by using SPIN we have to use PLTL (Propositional Linear Temporal Logic) notation which in ASCII, as used in SPIN, will be written using the following notational conventions for temporal and boolean operators:

```
[] means "always in the future"
<> means "sometime(s) in the future"
|| means "or"
&& means "and"
-> means "implies"
```

The mapping from propositions mentioned in the temporal formulas and conditions holding in the Promela model is defined by using definitions with the following general structure: `#define p (c)` means that proposition "p" mentioned in the temporal logic formula is true when condition "c" is true in the Promela model. Typical conditions are that a variable has a particular value. Conditions usually relay on two symbols: "==" for equal and "!=" for not equal but other usual relational symbols can be used.

Then we translate the original statement as the following temporal logic sentence:

```
[] ((bc43 && c0b && c1b) -> <> bc43rej)
```

which has to be supplemented by the following definitions:

```
#define bc43 (bookCarFor3==true)
#define c0b (cars00[0]!=0)
#define c1b (cars00[1]!=0)
#define bc43rej (bookRejected==true)
```

Δ

These definitions can be used to set the equivalent conditions in the prototype and to guide the testing procedure to explore that given the same conditions in the antecedent of the rule we obtain in the prototype the conditions described in the consequent of the rule.

B. Using SPIN to Analyse Traces

We focus here on two different ways of using this model checker to process traces produced by the prototype. One approach we followed was rewriting the trace in Promela and then checking that the trace was consistent with an associated PLTL formula, see section IV-B.1 below. We also use trace sequences to build a PLTL formula that allowed to check if equivalent steps can be made by SPIN executing a Promela specification, see section IV-B.2 below.

1) Using Temporal Logic Formulas as Partial Specifications: One way to check consistency between the traces obtained from the prototype with the basic properties of the system is by first transforming the traces into Promela code, i.e., a sequence of steps that mimic those taken in the prototype and then checking they are consistent with a related PLTL formula. In this case the PLTL formula can be seen as a simplified specification of an aspect of the system.

EXAMPLE *2:* Lets suppose we consider the problem of asking a resource (e.g. a room from a hotel) for the second time when the first booking provoked exhaustion of the resource in the provider shop. Because the reservation strategy as implemented at some stage of the prototype was incomplete, the system was not able to attempt booking in a different shop after it failed to book in the one that was used for the first time.

The traces considered below focus on a sequence of two steps. The first step is a successful attempt to book a room in Hotel1 that as a side effect provokes the Hotel1 to be full. The second attempt to book a room will be addressed to the same hotel and will fail. The formula to be checked is:

```
[] ((<>(requested && available))
                          ==>  <> allocate)
```

more informally: "always when a resource is requested and there is one available, then one

should be eventually allocated". Here "requested" means `((BookType=0) or (BookType=1))`, "available" means `(RoomsAvailableHotel1>0)` and `(RoomsAvailableHotel2>0)` and "allocated" means that a room has been effectively booked (allocated=true). These definitions can be used to build the PLTL formula to be checked with SPIN. The specification below contains a trace with one booking and we can check with SPIN the sequence of states visited in the trace is consistent with the formula.

Some explanation of the variables involved follows. `UserID` is a number identifying a user. e.g. between 1 and 10, `BookType` is the choice made by the user (0:"car", 1:"hotel", etc), `CCType` is a credit card brand. (0:"VISA", 1:"MC", 2:"WRONG"), `SupplierName` is identifying a shop (1:"Hotel1", 2:"Hotel2", etc), `RoomsAvailableHotel1` is the number of available rooms in Hotel1 (0...), `RoomsAvailableHotel2` is the number of available rooms in Hotel2 (0...), `RoomBooked` is a number identifying a room, e.g., between 1 and 5, `ShopAnswer` is the shop can inform the TA about the status of the operation, e.g., 0:"impossible", 1:"done", etc.

```
byte UserID,
     BookType,
     CCType,
     SupplierName,
     RoomsAvailableHotel1,
     RoomsAvailableHotel2,
     RoomBooked,
     ShopAnswer;

bool requested, available, allocated;
/* boolean variables used to identify
   states of interest during the trace */

init{ /* first login */
       UserID=2;
       BookType=1;
          requested=true;
       CCType=1;
       SupplierName=1;
       RoomsAvailableHotel1=1;
       RoomsAvailableHotel2=1;
          available=true;
       RoomBooked=1;
       ShopAnswer=1;
          allocated=true;}
```

But, if we add to the above specification the rest of the trace registering that a second booking is requested and the situation is such that 1) a second booking is requested also to Hotel1 when it is full and 2) Hotel2 has a room available:

```
       /* second login */
       UserID=2;
       BookType=1;
          requested=true;
       CCType=1;
       SupplierName=1;
       RoomsAvailableHotel1=0;
       RoomsAvailableHotel2=1;
          available=true;
       ShopAnswer=0;
          allocated=false
```

SPIN will prove that the enlarged sequence of steps does not validate the formula, i.e. will detect that despite there is a room available in Hotel2 the system will not allocate that room to the user as it reaches the state "allocated=false". △

2) Using Traces as Temporal Logic Formulas: We can use information produced by traces to verify that we obtain consistent behaviour when we explore the corresponding paths in the Promela model. We have several options that will accomplish this. We can

1) use the results witnessed during implementation testing
2) use interactive (user guided) simulation
3) transform the testing conditions into a formula written in Linear Temporal Logic (LTL) and check its validity.

We have explored a method that combines these options in an automated or semi automated manner. We can transform a sequence of actions taken while testing the prototype into a temporal formula and then by using SPIN check that an equivalent sequence of actions can be taken in the model. In order to automate the link between the output traces from the implementation and the input traces required by the model checker, we created a file that allows us to automatically translate implementation trace variable names to names used in the model. For example;

```
corresponds([cctype,mc],
            [[ccbit1,1], [ccbit2,0]])
corresponds([cctype,wrong],
            [[ccbit1,1], [ccbit2,1]])
```

where, the first line should be read as "if variable `cctype` has value mc in the implementation then variables `ccbit1` and `ccbit2` in the model have values 1 and 0 respectively".

EXAMPLE *3:* We carried out a test to determine whether it was possible to make the same number of credit card inputs as in the implementation. Firstly, we manually tested in the implementation that we can enter first three "`wrong`" credit card brands before entering a valid one, "`mc`". Then we used a program to build an LTL formula that allows us to check if that is a possible scenario in the model. The resulting file, "`formula.ltl`", will have the following content:

```
#define p1 (ccbit1==1 && ccbit2==1)
#define p2 (ccbit1==1 && ccbit2==1)
#define p3 (ccbit1==1 && ccbit2==1)
#define p4 (ccbit1==1 && ccbit2==0)
/* * Formula As Typed:
   <>(p1 && (<>p2 && (<>p3 && (<>p4)))) */
```

Finally, we used the formula in the LTL Property Manager section to generate the "never claim" (by pushing just two buttons). Using SPIN's "No Executions (error behaviour)" option to force a counterexample that confirms the sequence is possible. Δ

The main problem faced whilst exercising the interaction between the prototype and SPIN is the synchronization of output/input traces. Filters and translators can be written to help in making these steps as automatic as possible but, as discussed earlier, close co-operation of modellers and implementers in naming reduces the amount and complexity of the translation.

C. B model of the Travel Agency Case Study

For the travel agency we have defined an abstract machine where the state contains information about user sessions, hotel and car rental bookings, and most operations replicate the interface presented to the user by the travel agency, like login and enterCard. Besides those the machine has operations that perform in a single step the service requested by the user (for example: bookRoom and unbookCar), which abstracts from the complexity of the implementation that requires several operations to implement a request.

The abstract machine has several invariant clauses; each of the clauses describes a property that must be preserved by the travel agency at all times. Next, we describe informally each one of the six invariant clauses:

1) *"if a user has a booking, then it must have an assigned hotel where all the bookings have to be done. Conversely, if a user has an assigned hotel, then s/he must have a booking"*
2) *"the same property is valid for the car rentals"*
3) *"all hotel bookings of a user must be done in the hotel assigned to that user"*
4) *"the same property is valid for the car rentals bookings"*
5) *"if a session does not have a valid card, the travel agency will not perform any service (booking or unbooking)"*
6) *This clause states that if a session has a valid card, the travel agency will attempt to provide the requested service"*

A useful property that does not appear in the invariant is the one that states that:

> *if a user wants to book a room and there are available rooms, a room will be booked for that user*

Modelling this property in B raises problems, as it contains a temporal ordering of events: if in t_1 there is a room available, then in t_2 one of the available rooms will be booked. To overcome this problem we modelled instead a similar property:

> *if the request for booking a room was not fulfilled, then there must not be any rooms available*

By modelling the implementation, rather than a specification, we capture the behaviour of the implementation and can use the model to check that behaviour. If we want to use a model and model checker to explore the behaviour of our implementation, we should be sure that the models faithfully represent some abstraction of the implementation. Directing the modelling to aspects of the behaviour of the implementation and instrumenting our implementation to produce traces of tests, we were able to use the model checkers to test whether those traces violated any of the desired properties.

We ensure convergence of the behaviour of model and implementation by

1) co-evolution of model and implementation
2) traditional quality assurance methods like formal inspection
3) and, finally, checking implementation traces in the ProB model checker to compare the behaviour of the model and implementation

An example of the use of co-evolution of the model and implementation is described by the following description of convergence of the implementation and the model.

Changes in state in the B machine are caused by B operations. By manual inspection, we inserted trace beans in our instrumented version of the implementation at points where the change in state of our implementation corresponded to changes in state in the B machine. This means that we capture a trace in our implementation at the same point that a B operation will be enabled in the B machine.

Originally designed for theorem proving rather than model checking and animation, the B language and method on its own cannot provide the trace checking for our higher level testing approach. We used the animation and model checking capabilities of ProB to check our B model. Our novel trace-checking method is as follows:

1) Execute the application, either by traditional manual testing, or by an automated test harness simulating randomised user interaction
2) Select the relevant trace data from the database using SQL to provide input for a ProB animation of the B model
3) Load the model B machine and the trace into ProB and allow the animator to re-play the steps of implementation trace in terms of the currently available operations in the ProB animation of the B machine.
4) The higher level test is *passed*, if the ProB animation was possible. The higher level test *failed* if there was no combination of available B-machine operations that could reproduce the test run described by the implementation trace

EXAMPLE *4:* An example of a set of trace data selected from the trace database and the subsequent history of its animation in ProB follows. Note that the name of the B machine and its initialisation has been added.

```
machine('TravelAgency').
initialise_machine.

login(user1).
choice(ss1).
chooseService(ss1,H).
```

```
enterCard(ss1).
redoCard(ss1).
choice(ss1).
chooseService(ss1,U).
enterCard(ss1).
pickShop(ss1).
'-->'(respUnbookCar(ss1),_).
choice(ss1).
chooseService(ss1,U).
enterCard(ss1).
pickShop(ss1).
'-->'(respUnbookCar(ss1),_).
logout(ss1).
```

This is a subset of the trace data from an execution of the implementation, the whole trace is richer, but only this subset is required for B and ProB, and matches exactly the input requirements for each B operation and outputs from a B operation. The first part of each record is the name of the B operation, the part in brackets is the input parameters or outputs. The parameters are derived from captured trace data. For example: `chooseService(ss1,H).`

The B operation 'chooseService' takes the session number (ss1) and booking service (H) as parameters. 'H' is directly derived from other trace data and indicates that the user has chosen to make a hotel booking, but the session numbers automatically assigned to the different web service sessions that track a customer's transaction, at 32 characters long, are cumbersome for animation so we use a translation program to provide a B operation parameter with the same format as its initialised set of sessions. Δ

Early results show that we can successfully use model-based trace-checking in two ways:

1) Firstly, to ensure behavioural correspondence between the model and implementation in preparation for
2) Secondly to detect failures and property violations in the implementation

As an example of the first case, we detected a mismatch between the B model of the Travel Agency and the implementation; a failure of convergence.

EXAMPLE *5:* The following is a fragment of an implementation trace that proved to be unsuccessful when checked in ProB. In this example, a user has already logged in, we show the trace from the point that they choose the booking service they require (in this case a room). They then enter their credit card details, which were invalid and the user is returned to the point at which they re-input their booking choice followed by another attempt at entering a valid credit card. On the second occasion they are successful and go on to book a room.

```
chooseService(session1,br).
    enterCard(session1).
    again(session1).
    chooseService(session1,br).
    enterCard(session1).
    pickShop(session1).
    `-->'(respBookRoom(session1),_).
```

Upon investigating the reasons for the trace failing in ProB, we discovered that whereas the 'again' record in the implementation trace returns users to the state where they may re-input the booking service (chooseService) they require after a failed credit card attempt (enterCard). The 'again' operation in the B machine returns a user to the state where they may make another booking *after a booking has been completed.* There is no equivalent operation in the B machine to the 'again' method in the implementation. This means that the implementation and the B model do not have the same behaviour; they have not converged, even though the implementation and B model had been previously formally reviewed.

The team of modellers and implementers used the results of the trace checked through ProB to change the B model to match the behaviour of the implementation and ensure behavioural correspondence. The name of the again method in the implementation was changed to 'redoCard' (to avoid any further confusion) and an operation with the same name and equivalent behaviour was added to the B machine. Δ

Our continuing work focuses on using ProB in the second manner, to check implementation traces for invariant violations of the B model which, because of the convergence of the behaviour of the model and implementation, will indicate that the desired properties of the implementation have also been violated.

We have only shown fragments of traces to illustrate this paper, but we have successfully run large traces generated from automated testing.

V. ONGOING AND FUTURE WORK

A. Achievements in the Mortgage Broker Case Study

With this case study we wanted to make a complementary exercise which involved similar conceptual entities than in the Travel Agency case study but, enriched with a more difficult interaction protocol and an emphasis on asynchronous communication.

Again, different models and modelling approaches were used. One of the model has been developed under the Promela/SPIN framework and another by using B/ProB/AtelierB. These two models focused on different aspects of the system. The Promela model focused in the traditional synchronization of activities between the many parts of the system and the general protocol for requesting services, making offers and reaching agreement between the parts, while the B model focused on the subtleties of message communication by exploring the possibilities arising from message loss.

Both models were used to guide the prototyping activities by anticipating problems to come and supporting crucial decisions that increase the balance between desirable and feasible features. At an earlier stage, by using Promela, we detected the complexity of the communications protocol considered and from the B model the need to compromise on some

crucial steps of that communication. These facts helped us to outline the general agreement protocol imposing synchronous communication between the mortgage broker with lenders and insurers at the time of making a final confirmation to commit on a previous offer.

One other important practical challenge is related to computational complexity associated to model checking. It is well-known that one of the weaknesses in the model checking approach is state space explosion. The complexity of this particular case study, characterized by heavy communication between the different units (user, mortgage broker, lenders, and insurers) makes mandatory to model check the specification by partitioning the problem on functional subunits, e.g. the communication between user and mortgage broker or the communication between the mortgage broker and either one generic lender or insurer. Once some local properties of these subsystems have been checked some compositional verification should be applied in order to infer properties about the system in general.

It must be observed that this complexity issues are more closely related with model checking but not with the simulation facilities we use at early stages of the system to study the general behaviour or when we use simulations to extract traces from models. Using traces extracted from the prototype to confront them with the models can be affected by computational complexity depending on the size of the trace.

B. Components

We are also interested on exploring the correspondence between components as seen in the modelling languages, e.g. proctypes in Promela, and components as seen in the prototype, e.g. classes. We included component identifiers in the traces obtained from the prototype in order to easily identify faulty components when tracking errors. The models are built following the components structure used in the prototype as suggested by the meaningful classes and the traces are produced in a way that makes this relation explicit but this side of our proposal has not been yet fully exploited.

VI. Conclusions

This work reports on our experience using models to validate software. In our approach we prefer to use models in connection with testing. This increases the role of the model on guiding the development and foreseeing problems ahead. In this scenario the way that success of a model is measured is not by discovering big problems in the prototype but by warning with anticipation of difficulties and by guiding in an iterative process of small steps the development of the final product.

Instead of developing a complex model that then has to be followed by the subsequent implementation discovering big mismatches between idealized abstractions and the implementation, we inter-leaved modelling and prototyping. Here pair-programming and pair-modelling by a team that combines programmers and modellers came naturally. In that way both, model and prototype, co-evolve keeping conceptually close and the development team has higher chances to discover problems when it is cheaper to fix them. This strategy is especially well equipped to do the cheapest of all problem fixings: preventing them from occurring.

Here we described how to use mainly two modelling frameworks in that way, Promela/SPIN and B/ProB/AtelierB, but other modelling frameworks can be used as well. [14] [15] On one side we used the model to inform the prototype by generating traces but we also used the prototype to inform the model by what we call trace-driven model checking. [4] Under this framework, one extra source of requirements is used by considering temporal logic properties specification as declarative requirements.

VII. Acknowledgements

This work forms part of the ABCD project, which is funded by the EPSRC (GR/M91013/01), whose support we gratefully acknowledge. The idea of model and prototype co-evolution is mostly the intellectual achievement of Yvonne Howard, Andy Gravell and Peter Henderson.

References

[1] P. Henderson, *Systems Engineering for Business Process Change (Collected Papers from the EPSRC Research Programme)*, P. Henderson, Ed. Springer Verlag, 2000.

[2] ——, *Systems Engineering for Business Process Change (New Directions)*, P. Henderson, Ed. Springer Verlag, 2002.

[3] G. Holzmann, "The spin model checker," *IEEE Trans. on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[4] Y. M. Howard, S. Gruner, A. M. Gravell, C. Ferreira, and J. C. Augusto, "Model-based trace-checking," 2003, Proceedings of UK Software Testing Workshop, York, United Kingdom, 4-5 September.

[5] J. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University, 1996.

[6] *Atelier-B User Manual*, Steria – Technologies de L'Information, 1998.

[7] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.

[8] M. Leuschel, T. Massart, and A. Currie, "How to make fdr spin: Ltl model checking of csp using refinement," in *Proceedings of Formal Methods Europe FME'2001 LNCS 2021(DSSE-TR-2000-10)*, 2001, pp. 99–118.

[9] M. Leuschel, "Design and implementation of the high-level specification language csp(lp) in prolog," in *Proceedings of PADL'01*. editor I. V. Ramakrishnan, LNCS 1990, Springer Verlag, 2001, pp. 14–28.

[10] M. Leuschel, M. Butler, "ProB: A Model Checker for B," in *Proceedings of FME 2003: Formal Methods*, editor K. Araki, S. Gnesi, D. Mandrioli, LNCS 2805, isbn 3-540-40828-2, Springer Verlag, 2003, pp. 855-874.

[11] J. C. Augusto, C. Ferreira, A. Gravell, M. Leuschel, and K. M. Y. NG, "Exploring different approaches to modelling in enterprise information systems," Electronics and Computer Science Department, University of Southampton, Tech. Rep., 2003, technical Report, http://www.ecs.soton.ac.uk/∼jca/rm.pdf.

[12] M. Chessell, C. Griffin, D. Vines, M. Butler, C. Ferreira, and P. Henderson, "Extending the concept of transaction compensation," 2001, technical Report. Declarative Systems and Software Engineering Research Group, Dept. of Electronics and Computer SCience, University of Southampton. To appear in IBM Journal of Systems and Development.

[13] *B-Toolkit User's Manual*, B-Core (UK) Ltd, 1996.

[14] J. C. Augusto, C. Ferreira, A. M. Gravell, M. A. Leuschel, and K. M. Y. NG, "The benefits of rapid modelling for e-business system development," In Proceedings of 4th International Workshop on Conceptual Modeling Approaches for e-Business (eCOMO2003), pp. 17-28. Chicago, Illinois, USA. October 13-16, 2003.

[15] J. C. Augusto, M. Leuschel, M. Butler, and C. Ferreira, "Using the extensible model checker xtl to verify stac business specifications," in *Pre-proceedings of 3rd Workshop on Automated Verification of Critical Systems (AVoCS 2003), Southampton (UK)*, 2003, pp. 253–266.