Some Guidelines for Formal Development of Web-based Applications in B-Method

Abdolbaghi Rezazadeh and Michael Butler

School of Electronics and Computer Science,
University of Southampton,
Highfield, Southampton SO17 IBJ, United Kingdom
{ar02r,mjb}@ecs.soton.ac.uk

Abstract. Web-based applications are the most common form of distributed systems that have gained a lot of attention in the past ten years. Today many of us are relying on scores of mission-critical Web-based systems in different areas such as banking, finance, e-commerce and government. The development process of these systems needs a sound methodology, which ensures quality, consistency and integrity. Formal Methods provide systematic and quantifiable approaches to create coherent systems. Despite this there has been limited work on the formal modelling of Web-based applications. In this paper our aim is to provide researchers with some guidelines based on results from ongoing work to model a Web-based system using the B-Method. Session and state management, developing formal models for complex data types, abstraction of distributed database systems and formal representation of communication links between different components of a web-based system are the main issues that we have examined.

1 An Introduction to Web-Based Systems

Web-based applications are distributed systems that can be accessed using a Web browser. During recent years the extent and scope of their use has grown rapidly, significantly affecting all aspects of our lives. Industries such as manufacturing, travel and hospitality, banking, education, and government are Web-enabled to improve and enhance their operations. E-commerce has expanded quickly, cutting across national boundaries. Even traditional legacy systems have migrated to the Web. The scope and complexity of current Web applications varies widely: from small-scale, short-lived services to large-scale enterprise applications distributed across the Internet and corporate intranets and extranets.

Although numerous Web-based systems are in use now and many of us rely on them, the manner in which they are developed raises serious concerns [1–3]; they need to be reliable and perform well. To build such systems, Web-based system developers need a sound methodology, a disciplined process and a set of good guidelines. Due to the high amount of new demands, Web applications are evolving continually and the complexity of these systems is increasing rapidly. Therefore the use of a rigorous method becomes more important.

Formal methods use mathematical notation to describe systems in a clear and rigorous manner. Abstraction and stepwise refinement employed by formal methods is a valuable approach for developing complex Web-based systems. The B-Method is a well-known formal method [4] which has been applied to several software development missions including academic and industrial projects [5–7].

Our aim in this paper, through the modelling of this specimen Web-based system, is to identify some challenging aspects of these types of systems and propose an approach to their formal representation. We hope to provide a set of guidelines which could serve as a basis for further work. In the rest of this paper we present the travel agency case study and briefly discuss its initial aims and objectives. The chosen case study has been selected to be inclusive enough to represent the main properties and functionality of typical Web applications. By developing formal models in B we have extracted some generic and essential patterns. These patterns are considered to model some common properties and functionality shared by a broad category of Web applications. In the next step we have tried to find some appropriate formal refinements for these abstract patterns which could be provable within the framework of the B prover tool [8–10]. As Web applications are distributed systems, the decomposition of primary refinement models into subsystems and introducing suitable formal models for communication links are other objectives. The last section concludes the paper with recommendations for further work and discussions.

2 Informal Representation of the Case Study

Here we outline the main requirements and sketch the overall architecture of the system. The aim is to develop a Web-based Travel Agency system to enable potential users to access it through an Internet connection using a standard Web browser to perform one or more of the following tasks:

- Book a flight or Cancel a booked flight
- Book a room or or Cancel a booked room
- Hire a car or Cancel a hired car

The Travel Agency Web-based system is hosted on the Travel Agency Server which is responsible for processing the Web-clients' requests. These messages are produced and sent by the client browser through Internet links and based on HTTP or other similar standards. The travel agency system relies on a group of secondary agencies' servers like flight agencies to accomplish the client requests. The travel agency system uses Internet links to communicate with the secondary servers. A simple architecture of this system is depicted in Figure 1.

In Figure 1 we see that more than one client could communicate with the travel agency system simultaneously. The travel agency system will manage the status of different sessions using state variables, stored in a local database. For booking requests like flight booking, a message which includes details about the request will be broadcast to all related agencies' servers by travel agency system. Responses which the travel agency should expect could vary from zero to the

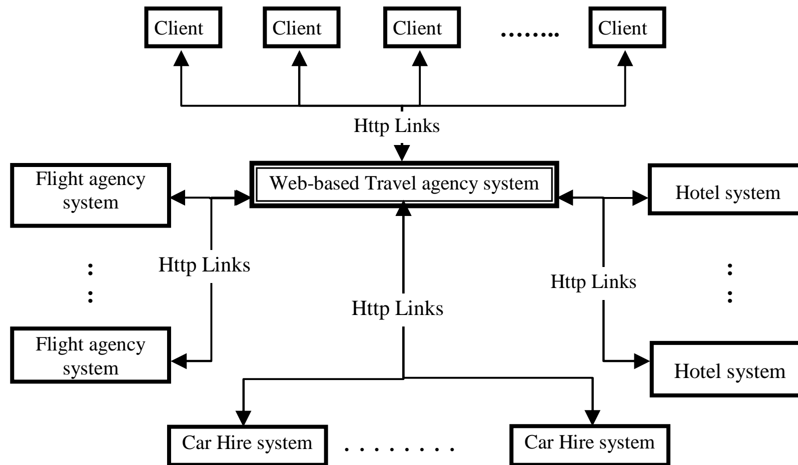

Fig. 1. A Simple Architecture of the System.

number of all secondary agencies in the best situation. The collected response will be sent by the travel agency system to the appropriate client. In other cases, like cancelling a booked flight, the request will be sent directly to the related flight agency. Also it is quite convenient to assume a local database in the travel agency server for representing all booked services. This database could reduce the amount of communication and complexity of un-booking process.

3 An Overview of Formal Development Process

As we mentioned previously our main objective, in applying formal method to this case study, was to identify some common challenging issues and propose some formal models for them. Therefore instead of detailed presentation of formal models, in this section we have summarised the formal development process.

This work is based on Event-B style for development of distributed systems [11, 12]. Unlike standard B, which is used to specify and develop software modules in B, Event-B was introduced for modelling of distributed systems. In the Event-B style operations are called "events" which may occur spontaneously rather than being invoked. Those events are no longer pre-conditioned, but guarded by a predicate, which express the condition under which the event can be enabled. When we refine a model, we either refine an existing event by strengthening the guard or/and the before-after predicate (removing non-determinism or applying data refinement), or add a new event which is supposed to refine the skip event. The introduction of new events is supported by superposition method [13, 14]. In superposition, new functionality is added to an existing model in the form of additional variables and assignments to these variables as new operations, while the original computation is preserved.

In the first stage of formal process an abstract model based on Event-B style has been produced. The abstract model is a single B-machine which encloses some operations to model the main functionality of the travel agency system from the viewpoint of the users. In the second step we have refined the abstract specification by introducing client side operations based on the superposition methodology. Operations of the abstract model have been classified as the server side operations at this stage. Some operations of the abstract model which are influenced by the introduction of client operations have been refined by adding extra guards and removing non-determinism.

Operations of the secondary agencies servers have been introduced in the second refinement model. In this stage some formal definitions for distributed databases have been added. Each secondary server has a local database which is contains information about available service that this agency can offer to its customers. Data distribution among secondary servers and the travel agency system leads to distribution of processing between servers. In other words, introducing new operations which finally reside on secondary servers for manipulating distributed data resulted in further refinement of the travel agency operations in this stage. In the second refinement we have operations of the clients, the travel agency system and the secondary servers.

Decomposition is the main strategy to tackle the complexity of the model in Event-B style. Introducing communication links between different parts is a pre-stage to the decomposition process. Therefore in the third refinement stage we have introduced communication links. The main challenging questions which we have identified during the above mentioned development processes are:

- Session and State Management in Both Client and Server side
- Inter-Server Interactions
- Refinement of Complex Data types
- Abstraction of Distributed Databases
- Formal Modelling of Communications Links

In the following sections we have examined these issues in detail and we have presented some solutions for them. Although we have used the travel agency case study to discuss the main properties of a Web application and to clarify the key issues in developing a B-model for them, the identified aspects and proposed solutions could be applied to a wide range of Web applications.

4 State Representation in Web-based Systems

The Web started as a means for sharing documents among scientists. Its designers have built the underlying technology (e.g., HTTP and HTML) with these goals in mind. Since then, people have realised the Web's potential as an application delivery medium and have started to exploit it. With the growth of e-business applications, the Web is rapidly being transformed into an application-intensive environment. In Web-based application the core functionality of system, the business logic, is handled by the server. Most web applications need

to maintain communication sessions with their client, and monitor each client's individual status and activities. Unfortunately, the communication protocol between web browser and web server (HTTP) is stateless and it does not provide the functionality on session control. Therefore it is not trivial to maintain information about each client interaction with server. The server-centric architecture of current Web applications makes a server-side session the natural choice. In the following sections we have examined this subject in detail.

4.1 Session Handling and State Management in Server side

State maintenance is one of the major issues in many applications, such as e-commerce and banking applications. As transactions between Web clients and Web servers occur in a stateless environment, state must somehow be passed from one transaction to the next in a Web application. Keeping state data on the server side is generally considered the safest and most appropriate technique when handling information of a sensitive nature.

The server uses a session's state variables to identify a user, process the input data provided by a client and determine user rights or the type of access to be offered to a user. Furthermore, based on the information which has been provided by the client, the server can set state variables to determine the next possible execution path.

Challenge: How do you represent the state information related to a user's interaction with a Web application?

Guideline: We have used explicit state variables to represent sessions state information on the server side. By defining two reference sets for state and sessions ID and a mapping function from a session ID to session state we can manage each session in the server side identically. So each session has a session identifier *"sid"* which could be used as an index to access session information on the server side. A new *"sid"* could be allocated to a new client as soon as it establishes a connection with server and afterward the client can use this *"sid"* on subsequent interactions.

To clarify the guideline we have presented a snapshot of the specification machine for the Travel agency case study in Figure 2. We have introduced the set *"STATE"* and *"SESSION"*. The first definition represents the possible states for a client session and the second one serves as a typing reference for sessions' ID. The *"session_state"* variable maps each client session to its related state. The variable "*session*" represents the set of all current active sessions. The operation "*StartNewSession*" models the creation of a new session by the travel agency system. This operation allocates a free session ID for the newly created session and sets the necessary environmental variables for it. Any changes in a session's state variable could enable a operation and execution of an operation could resulted in some changes in a state variable. For example, the *"SelectService"* operation is enabled when the session state is *"fresh"* and its execution changes the state of related session to one of *"booking"*, *"unbooking"* or *"signed_in"* state. The *"SelectService"* operation models the interaction of the clients with the system, when they select an available service.

```
MACHINE TravelAgency
SETS
   SESSION;
STATE={fresh,booking,unbooking,service_selct,options_ret,choice_made,
             signed_in,certified,valid,invalid,booking_ret,unbooked_sel};
DEFINITIONS
   freshSESSION ≙ SESSION - session;
VARIABLES
   session, session_state,
INVARIANT
   session ⊆ SESSION ∧
   session_state∈ session → STATE ∧   ...
INITIALISATION
   session := ∅  || session_state := ∅  || ...
OPERATIONS
  StartNewSession ≙
   ANY sid WHERE  sid∈ freshSESSION  THEN
     session := session ∪ {sid} ||
     session_state(sid) := fresh
   END;
  SelectService ≙
  ANY sid  WHERE sid∈session ∧ session_state(sid)=fresh  THEN
    SELECT (……. ) THEN
      session_state(sid):= booking
    WHEN (…….) THEN
       session_state(sid):= unbooking
    WHEN (…….)THEN
           session_state(sid):= signed_in
    END || ...
  END;
  FlightRequest ≙
   ANY sid WHERE sid∈ session ∧ session_state(sid)= booking THEN
     session_state(sid):= service_selct
   END;
```

Fig. 2. Abstract model of the travel agency system.

4.2 State Management in client side

In Web based application Web clients generally are classified as thin clients. This implies that processing in the client side usually is not significant. Web clients take input from users, perform type checking and simple data validation and in some cases carry out data encryption if necessary. Web clients use the application through Web browsers, over the Internet. They interact with system concurrently, independently, in an asynchronous manner. You can't control what they're doing and when they do it. Although the browser and underling mechanism do not support sate handling, still some coordination mechanism and state passing between server and client operations is necessary.

Challenge: How do you maintain the state information in the client side and perform coordination between different clients and the Web server.

Guideline: We have used a message-based mechanism for this purpose. Each message is mapped to a session ID which relates the message to a specific client session. The message-based mechanism could be considered as an implicit state representation in the client side. Therefore from this viewpoint we can assume that two different approaches have been taken for state representation in the server and the client side. We have found that the main advantage of this approach is to avoid shared state variables among clients and the Web server which in its turn could lead to further complication.

We have presented some operation of the clients along with the server's operations from first refinement of the case study in Figure 3 to illustrate the guideline. We have used comments to make a distinction between the server and newly introduced client's operations. The server operations use explicit state variables for state representation. On the other hand, the client operations employ an implicit message-based method for state representation and coordination with the server operations.

The session ID, "*sid*", plays a central role to convey state information between client and server. However there is a situation that a client has triggered a new session but it has not obtained a session ID yet. In this step the client should use a temporary identification mechanism which could be the IP address or any other similar mechanism. The "*Client_ReqSession*" operation in Figure 3 depicts this situation. We have defined a new variable named "*handle*" to use it as temporary index to represent a client request for a new session. When in the *"StartNewSession"* operation the server has processed this request it allocates a new session ID for this specific client session and replies to the client by placing the new session ID in the "*new_client*" message buffer. In the *"Get_SessionID"* operation the client receives this allocated *"sid"* and it will use it through the rest of session to communicate with the travel agency server. For example in the *"PicService"* operation we have a message buffer named "*reqsevice_buf*" which has been defined as a mapping from *"session"* to *"REQUEST"* to carry the client's requests to the Server.

As we have mentioned in section 3, we have used superposition refinement to introduce client operation. This means that we retain the variables and operations of the abstract specification and introduce new operations which have no effect on the pervious variables. Some new variables which can be exploited by both the clients' operations as well as the Web server have been introduced in this stage. New variables are used as message buffers to exchange data between client and server operations. The introduction of these new variables has some implication on the Web server's operations.

In the abstract model some operations use nondeterministicly chosen values which need to satisfy just some typing and basic state conditions. In the refinement model some changes have been made in these operations' guard. This is to refine the nondeterministic choices to the available values in the related message buffers which are provided by clients. By using superposition refinement instead of more general Event B refinement in this stage, we do not require any gluing invariant which implies an easier set of prove obligations.

```
SETS
   HANDLE
DEFINITIONS
 freshHANDLE ≙ HANDLE - dom(new_client)
INVARIANT
 /*  Client Variables  */
   new_handle ⊆ HANDLE ∧
   new_client ∈ HANDLE ⇸ SESSION ∧
   token ⊆ SESSION ∧
   fresh_session ⊆ SESSION ∧
   reqsevice_buf ∈ SESSION ⇸ REQUEST
OPERATIONS
 Client_ReqSession ≙       /* Client Operation */
   ANY handle WHERE handle ∈ freshHANDLE THEN
     new_handle:= new_handle ∪{handle}
   END;
 StartNewSession ≙       /* Server Operation */
  ANY sid, handle WHERE sid ∈ freshSESSION ∧ handle ∈ new_handle  THEN
    session:= session ∪ {sid} ||
     session_state(sid) := fresh ||
     new_client(handle):= sid ||
    new_handle:= new_handle - {handle}
  END;
Get_SessionID ≙       /* Client Operation */
   ANY sid WHERE sid ∈ SESSION ∧ sid ∈ ran(new_client) THEN
     token:= token ∪ {sid} ||
     fresh_session:= fresh_session ∪ {sid} ||
     new_client:= new_client ⩥{sid}
   END;
PicService ≙       /* Client Operation */
  ANY sid, req WHERE sid ∈ fresh_session ∧ req ∈ REQUEST  THEN
      reqsevice_buf(sid):= req ||
      fresh_session:= fresh_session - {sid}
  END;
```

Fig. 3. Some operation of the first refinement.

4.3 Conducting Inter-Server Interactions

Coordination and communication management is an important issue in modelling interactions between two or more servers. In the case of inter-server communications, unlike client and server communication, both parties which are involved in a session are providing some services. Interaction between the travel agency system and secondary servers is an example of such inter-server communication. For example the travel agency system can ask a flight Agency server for available flight options and the flight agency server will reply with available options.

Challenge: What is the best way to model inter-server interactions?

Guideline: Considering the fact that the servers are independent, any approach to modelling their interaction, should provide a solution with minimum possible cohesion between these subsystems. Using the message-based approach seems to be a good candidate for this purpose and furthermore it complies with

common web services technologies.The messages are defined as mapping from a session ID to the requested information.

The message-based approach could be exploited to exchange both data and state information between servers. Regarding the fact that server to server communications are mostly asynchronous, the message-based communication is an appropriate candidate.

Some operations of the secondary servers and the travel agency system which involve communication are presented in Figure 4. In this model, *"reqflight_buf"* is used to transmit requests from the travel agency to flight agencies. Flight agencies use *"respflight_buf"* message buffer to send responses to the travel agency.

```
INVARIANT
   /* Server's New Variables */
   reqflight_buf ∈ FLIGHT_AGENCY ⇸ (SESSION ⇸ FLIGHT_REQUEST) ∧
   /* Flight Agency Variables */
   respflight_buf ∈ SESSION ⇸ (FLIGHT_AGENCY ⇸ ℙ(FLIGHT_DETAIL))
OPERATIONS
Request_Flight ≙       /* Server Operation */
 ANY sid,fr WHERE sid ∈ SESSION ∧ fr ∈ FLIGHT_REQUEST THEN
    reqflight_buf:= λ fa . (fa∈ FLIGHT_AGENCY | reqflight_buf(fa) ∪{sid↦fr})
 END;
Resp_FlightReqs ≙       /* Flight Agency Server Operation */
  ANY sid,fa,fr WHERE sid ∈ session ∧  fa ∈FLIGHT_AGENCY ∧
                                  fr ∈ FLIGHT_REQUEST   THEN
    ANY xx WHERE xx∈ ℙ(FLIGHT_DETAIL) ∧ xx ⊆
                               Matchflight(fr ↦ flight_db1(fa)) THEN
       respflight_buf(sid):= respflight_buf(sid) ∪ {fa ↦ xx}
    END   ||
    reqflight_buf(fa):= reqflight_buf(fa)- {sid ↦ fr}
  END;
```

Fig. 4. Some operations of the secondary servers.

5 Abstraction and Refinement of Complex Data-types

In many Web applications frequently we need to represent some complex data types in different abstraction levels. For example this data could be a record with many fields containing all necessary information for a booking request. Refining abstract data types in a single step, especially when we do not need all details in this step, is not a good approach to refinement; because it swiftly turns our simple abstract model into an over-complicated refined model. Therefore we need to find a mechanism for stepwise refinement of the abstract data types.

Challenge: What is a proper abstraction for data structures like records and how we can refine an abstract representation of a record in a step-wise manner?

Guideline: We found that most details could be abstracted away by defining some simple data types in the form of set definitions in the specification level.

In refinement stage to overcome the problem of unnecessary detail we found that, instead of direct refinement of abstract data types, some constant mapping could be used. A mapping defines a relation from an abstract data type to the required additional detail. By employing this method we introduce fields into refined model when it is necessary.

The abstract data types make operations very simple and understandable at specification level and help us to have a clearer picture of overall functionality of system. But we need to introduce the necessary details into these abstract data types in the refinement level. Using constant mappings to introduce new fields of a previously defined abstract type could help to avoid unnecessary complication in the early stage of refinement and postpone the detailed refinement of abstract data types to after decomposition.

Using constant mapping to refine an abstract record may present some ambiguity to the reader. So we will try to make some clarification here. Let assume that we have an abstract record, *"REC"* in the specification level. We want to refine this abstract record by introducing two new fields of it, namely *"afield"* and *"bfield"*. We can define these two fields as a constant mapping from *"REC"* to two arbitrary types *"SETA"* and *"SETB"* respectively. Now we can assert that for any *"aa"* and *"bb"* that *"aa"* belong to *"SETA"* and *"bb"* belong to *"SETB"* we can define a record that belongs to *"REC"*. Performing record refinement with a constant mapping rather than a variable mapping simply means that this information is global to all subsystems. Using constant mapping has not any restrictive impact on records manipulation. To clarify this issue we have presented an example operation in Figure 5 that adds a record to a database.

Here is An example from the case study is provided in Figure 6. We have two abstract data types; the first one is an abstraction for a record which contains all the necessary information for a flight request and the second one is the abstraction of a record which contains all details about an offered flight by a flight agency. We have used two abstract set definitions *"FLIGHT_REQUEST"* and *"FLIGHT_DETAIL"* for these two abstract records respectively. In the refinement stage we need to access the flight agency that has provided a flight. We assume that the flight agency identifier is a part of the *"FLIGHT_DETAIL"* record. Instead of direct refinement of the abstract data type, we have defined a constant mapping from *"FLIGHT_DETAIL"* to *"FLIGHT_AGENCY"* which could satisfy our requirement in this stage. The definition of this constant mapping is presented in Figure 5. The use of a constant function provides a way of modelling a record's field in B. By using similar techniques we are able to introduce any extra detail which might be necessary in successive refinement steps. Obviously at the implementation stage we have to replace these constant mapping with an actual data field; but the fact that we could postpone this step until after decomposition is helpful.

```
MACHINE Database
SETS
   REC; SETA; SETB
CONSTANTS
  afield, bfield
PROPERTIES
  afield ∈ REC⟶SETA  ∧ bfield ∈ REC⟶SETB ∧
  ∀ (aa, bb•((aa ∈ SETA ∧ bb ∈ SETB)⇒
  ∃ rr. (rr ∈ REC ∧ afield(rr) = aa ∧ bfield(rr) = bb))
VARIABLES
  db
INVARIANT
  db ∈ ℙ(REC)
INITIALISATION
   db:=∅
OPERATIONS
 Add_Database ≙
  ANY af,bf,rn WHERE af ∈ SETA ∧ bf ∈ SETB ∧ rn ∈ REC ∧
                     afield(rn)=af ∧ bfield(rn)= bf
   THEN
     db:= db ∪ {rn}
   END
END
```

Fig. 5. An example of constant mapping.

```
CONSTANTS
  flightagency
PROPERTIES
  flightagency ∈ FLIGHT_ DETAIL → FLIGHT_AGENCY
SETS
 FLIGHT_REQUEST; FLIGHT_DETAIL; FLIGHT_AGENCY;
```

Fig. 6. An example of constant mapping from the case study.

6 Abstraction and Refinement of Distributed Databases

Data that is shared between Web components and persistent between invocations of a Web application is usually maintained by one or more databases. These databases generally are distributed over different servers. Developing a formal abstract model and refinement for them is another challenge that we examine in this section. This issue has a close relation with process distribution; therefore we consider the process distribution and distributed databases together. We can assume different functionality for a database system. For example the simplest case is a database which allows its contents to be viewed by different parts of the Web application. On the other hand a complex database could support different type of queries and permits updating current information or removing

some records from it. As the system is distributed it means that when a server makes some changes in its database which could affects another part of the Web application, it takes some time for the other part to know about it.

Challenge: How we can represent a proper abstraction and refinement of certain distributed database operations?

Guideline: In a distributed setting involving multiple clients, the high level specification of a transaction such as confirming a flight booking needs to include the possibility of failure. Also query operations involving multiple databases should be specified very loosely at the abstract level.

To understanding the complicate relation of process and Database refinement we need some examples. In the travel agency system as depicted in Figure 1 we have a set of secondary servers which store some information about their available services. Based on web clients' requests the travel agency server occasionally initiates and sends a distributed query to these secondary servers for information lookup. Later it should collect and send available services to related Web clients. Obviously in the specification level we need an abstract formal representation of these distributed processes and databases.

The first abstract model is presented in Figure 7. In this specification *"Matchflight"* is a constant function type definition. It takes "*FLIGHT_REQUEST"* as an abstraction for user request and an abstract database which contains some "*FLIGHT_DETAIL"* records and returns a set of *"FLIGHT_DETAIL"* records which match the user request. In this abstract model we have defined the virtual

```
CONSTANTS
  Matchflight
PROPERTIES
 Matchflight∈ FLIGHT_REQUEST × ℙ(FLIGHT_DETAIL)→ℙ(FLIGHT_DETAIL)
INVARIANT
  flight_db ∈ ℙ(FLIGHT_DETAIL)
Retrieve_FlightOptions≘        /* Server Operation */
 ANY sid, fr WHERE sid ∈ session ∧ fr ∈ FLIGHT_REQUEST  THEN
   ANY xx WHERE xx ∈ ℙ(FLIGHT_DETAIL) ∧ xx ⊆ Matchflight(fr ↦flight_db)
   THEN
    flight_options(sid) := xx
   END
END;
```

Fig. 7. An abstract model of the database operation.

database, "*flight_db"*, as an abstract representation for a set of distributed databases which reside on secondary servers. As we mentioned earlier the content of these distributed databases could change independently from the travel agency system. Based on the above assumption we have defined the operation *"Retrieve_FlightOptions"* which is an abstraction for collecting secondary servers' responses to a distributed query for a service. Obviously we have not introduced

secondary servers and their related databases in the abstract model to avoid making the model over-complicated.

A refinement of the abstract model is presented in Figure 8. In this refinement based on the superposition technique we have introduced some new operations. The *"Request_Flight"* operation models the travel agency side event that initiates a query broadcast to a set of secondary servers. Equally when a secondary server receives a query for a service, it responds if it has any available option(s). This is demonstrated in "*Resp_FlightReqs"* operation. The virtual database definition has been replaced by actual databases which are distributed among secondary servers and we have defined these by a mapping form *"FLIGHT_AGENCY"* to power set of *"FLIGHT_DETAIL"*. The *"Retrieve_FlightOptions"* has been refined in response to the introduction of the new operations and now clearly reflects the fact that it should collect the secondary servers' responses to reply the initial service query. Our intention is that the abstract database is an ab-

```
CONSTANTS
  Matchflight
PROPERTIES
 Matchflight ∈ FLIGHT_REQUEST × ℙ(FLIGHT_DETAIL) → (FLIGHT_DETAIL)
INVARIANT
   flight_db ∈ FLIGHT_AGENCY ⇸ ℙ(FLIGHT_DETAIL)
OPERATIONS
 Request_Flight ≘          /* Server Operation */
  ANY sid,fr WHERE sid ∈ SESSION ∧ fr ∈ FLIGHT_REQUEST THEN
   reqflight_buf:= λfa . (fa ∈ FLIGHT_AGENCY | reqflight_buf(fa) ∪ {sid↦fr})
  END;
 Resp_FlightReqs ≘         /* Flight Agency Server Operation */
 ANY sid,fa,fr WHERE sid ∈ session ∧  fa ∈FLIGHT_AGENCY ∧
                                     fr ∈ FLIGHT_REQUEST
   THEN
    ANY xx WHERE xx∈ ℙ(FLIGHT_DETAIL) ∧ xx ⊆
                                    Matchflight(fr ↦ flight_db1(fa))
     THEN
      respflight_buf(sid):= respflight_buf(sid) ∪ {fa ↦ xx}
     END
   END;
 Retrieve_FlightOptions ≘          /* Server Operation */
 ANY sid WHERE sid ∈ session THEN
   flight_options(sid):= ⋃ fa.(fa ∈ FLIGHT_AGENCY ∧
                    fa∈ dom(respflight_buf(sid)) | respflight_buf(sid)(fa))
 END;
```

Fig. 8. A refinement of the database operations.

straction of the union of all of the distributed databases. The response to a client request is formed from the union of the responses from each of the agencies so this may seem like a reasonable abstraction. However, we faced some difficulties

when we tried to prove that the model in Figure 8 is a valid refinement of the abstract model in Figure 7. The problem is that the abstract specification of *"Retrieve_FlightOptions"* is based on the value of (the abstraction of) all the flight agency databases at the point at which the results are collated by the travel agency. But the results collated in the refined version will have been generated by the individual flight agencies at earlier points in time. If the flight agency databases did not change in between the point at which they respond to a flight request and the point at which those responses are collated by the travel agency, then our refinement would be valid. However, this is clearly an unrealistic restriction. The fact that the user gets information about an available flight is no guarantee that that flight will still be available when they try to book it. In principle the value of a flight agency database at the point of generating a response might be completely different to its value at the point at which that response is collated with other responses.

One possible abstract specification for this kind of distributed database query is presented in Figure 9. Although it appears to be a very loose specification but it is the strongest specification that we could introduce in the abstract level. In this specification we do not use definitions like *"Matchflight"* and virtual database *"flight_db"*. As we mentioned earlier data and process distribution have

```
Retrieve_FlightOptions ≙       /* Server Operation */
 ANY sid WHERE sid ∈ session  THEN
  VAR xx IN
    xx : ( xx ∈ ℙ(FLIGHT_DETAIL))  ||
    flight_options(sid):= xx
  END
 END;
```

Fig. 9. A valid abstraction of the database operation.

a reciprocal effect on each other. We present another scenario from the travel agency case study to clarify this issue further. During the booking process when a web client receives some available options from the travel agency system, it can select one of them and send back its selected service to the travel agency system. Now the travel agency system will know which secondary server has offered this service and then send a booking request to this specific secondary server. In the meantime this service could have been offered to another Web client and is no longer available. Therefore in general the travel agency system could expect either a successful or a failed response for a requested service booking. If the travel agency system receives a confirmation for service booking it will add an appropriate record to it local database for booked services. In either case of success or fail, it should reply to the related Web client with a suitable response.

Developing an abstract formal specification for this case is not a straightforward task. In the abstract level we have not introduced secondary servers, just to avoid complication, but we have to find a mechanism to model the system behaviour. Using nondeterministic "choice" could be an acceptable approach to model this case in the abstract level. This solution is depicted in Figure 10. It should be emphasised that in the actual system the booking process is a two stage process. If the requested service is still available on a specific secondary server, then the first stage takes place on that secondary server. In the second stage, when the travel agency system receives a message from this specific secondary server denoting successful booking in the first stage, then the travel agency system will add this booking to its database. Therefore the booking database on each secondary server just stores booked services which have been offered by this specific server. On the other hand the booking database on the travel agency system stores all booked services of its users. The *"Flight_Booking"* operation in Figure 10 demonstrates the booking process in the travel agency system. This operation is defined as the nondeterministic choice of two outcomes. In cases, the request is processed. In the first case it results in a successful booking, while in the second (failed) case, no booking is made. In the refined model when we

```
Flight_Booking ≙
 ANY sid,fd WHERE sid ∈session ∧ fd ∈ FLIGHT_DETAIL ∧
        session_state(sid)= valid ∧ sid ↦ fd ∈ selctflight_buf  THEN
   CHOICE
     flight_booking := flight_booking ∪ {session_user(sid) ↦ fd} ||
     selctflight_buf := {sid}⩤ selctflight_buf ||
     session_state(sid):= fresh || session_request(sid):= none
   OR
     selctflight_buf:= {sid}⩤ selctflight_buf ||
     session_state(sid) := fresh || session_request(sid):= none
   END
 END;
```

Fig. 10. Modelling the possibility of failure.

introduced databases in the secondary servers, now the booking process in the Travel agency system is no longer nondeterministic and it depend on the state of these databases. Therefore the refined operation could be modelled as we presented in Figure 11. Here the *"Agency_flight_booking"* shows the first stage of booking in the secondary server and the *"Flight_Booking"* has been refined accordingly.

7 Developing Formal Models for Communication Links

Communication links are the medium for interaction between different parts of distributed systems. In Web-based systems communication links connect a client

```
CONSTANTS
  flght_agency
PROPERTIES
  flght_agency ∈ FLIGHT_DETAIL → FLIGHT_AGENCY
Agency_flight_booking ≙        /* Flight_agency Server Operation*/
  ANY fa,sid,fd WHERE fa ∈ FLIGHT_AGENCY ∧
                               sid ∈ SESSION ∧ fd ∈ FLIGHT_DETAIL
    THEN
    SELECT fd ∈ flight_db1(fa) THEN
      ANY fdb WHERE fdb ∈ ℙ(FLIGHT_DETAIL) ∧ fdb ⊆ flight_db1(fa) THEN
        /* Updating original Database that maybe affected by booking */
        flight_db1(fa):= fdb
      END ||
       fa_booking(fa):= fa_booking(fa) ∪ {(fd ↦ session_user(sid))} ||
       flightbookingresp(sid) := success
      WHEN fd ∉ flight_db1(fa) THEN
        flightbookingresp(sid) := failed
     END
  END;
Flight_Booking ≙        /* Server Operation */
 ANY sid,fa,fd WHERE sid ∈session ∧ fd ∈ FLIGHT_DETAIL ∧
        fa∈ FLIGHT_AGENCY
    THEN
      SELECT sid↦success∈ flightbookingresp THEN
        taf_booking:= taf_booking ∪ {(session_user(sid) ↦fd ↦fa)} ||
        suc_session:=suc_session ∪ {sid}
      WHEN sid↦failed∈ flightbookingresp THEN
       selectflight_buf(fa):= selectflight_buf(fa) - {sid ↦fd} ||
       unsuc_session:=unsuc_session ∪ {sid}
      END
 END;
```

Fig. 11. Refined model after introduction of secondary servers.

to a Web server or a Web server to another Web server or a data server. Although communication in different levels could be based on different protocols and standards, but in general a message-based approach is a widely accepted method in Web based application. This approach is flexible and general enough to be implemented in the context of available standards like XML based technologies and tools. In Event-B developments introducing communication is an important stage before decomposition of a single model to several sub-models. In the following sections we discuss the process of developing a formal model for communication links.

7.1 Formal models of Synchronised Communication Links

Synchronised communication is a common pattern of communication between Web clients and Web servers. In other words generally the communication between clients and the Web sever follows the send-process-receive pattern.

Challenge: What is an appropriate abstract model and refinement for communication links between clients and the Web Server?

Guideline: At the abstract level it is convenient to model communication link a one-place buffer. But this causes problems with model decomposition. So we present a pattern for refining a communication link involving a one-place buffer by an unbounded buffer.

To exemplify this issue we have presented some operation of the travel case study in Figure 12. We have used a function definition to present a single place buffer for data communication between each client and the travel agency server. In this model the *"reqsevice_buf"* and *"resp_buf"* define a single-place buffer from

```
INVARIANT
  reqsevice_buf ∈ SESSION ⇸ REQUEST ∧
  resp_buf∈ SESSION ⇸ RESPOSE
OPERATIONS
 PicService ≘        /* Client Operation */
 ANY sid, req WHERE sid ∈ fresh_session ∧ sid∉dom(reqsevice_buf) ∧
                                          req∈ REQUEST ∧ req≠ none THEN
   reqsevice_buf(sid):= req ||
   fresh_session:= fresh_session - {sid}
 END;
 SelectService ≘        /* Server Operation */
  ANY sid,req WHERE sid ∈ session ∧ req ∈ REQUEST ∧ resp ∈ RESPONSE ∧
                                                sid∈dom(reqsevice_buf) THEN
   session_request(sid):= req ||
   reqsevice_buf:= {sid} ⩤ reqsevice_buf ||
   resp_buf(sid):= resp
 END;
Submit_Servic_Dtail ≘        /* Client Operation */
 ANY sid,resp WHERE sid ∈dom(resp_buf) ∧ resp ∈ RESPONSE ∧
                                            resp_buf(sid):= resp THEN
   resp_buf(sid):=  {sid} ⩤ resp_buf(sid)
 END;
```

Fig. 12. Abstract model with one-place buffers.

"session" to "*REQUEST*" and *"RESPOSE"* respectively. The *"PicService"* is a client operation which puts a request in the *"reqsevice_buf"*. The client then waits for the server response, i.e., the first client operation is no longer enabled for this session and the second client operation is enabled when a response appears in the response buffer. On the server side the *"SelectService"* operation takes the request from the buffer and then produces a response for the client by placing a response in the *"resp_buf"*. Later the client's operation *"Submit_Servic_Dtail"* can take this response from buffer when received it.

Before the decomposition step we have to refine each buffer by splitting it to three buffers and distribute them between the client, the communication and the

server machines. But when we replace a single-place buffer with three single-place buffers we face difficulty. We should be able to demonstrate that all buffers are empty when for example a web client's operation produces a new message to put a in the buffer. Clearly this is not a practical solution since, for example, a client cannot see whether or not a buffer on the server side is empty. To overcome this difficulty we consider the refinement of the one-place buffers with unbounded buffers based on using sequence definition in B-method.

This intermediate refinement would help to split the buffers between different machines and without too much restriction discharges prove obligations associated with this distribution. Using unbounded buffers resolves the need for condition that distributed buffers should be empty when we add a new message.

The intermediate refinement for the above model is presented in Figure 13. Here the single-place buffers of Figure 12 have been replaced by unbounded buffers. Part of the necessary gluing invariant are illustrated as well. The gluing invariant was constructed using an iterative approach in combination with the B prover as described in [15]. We first considered the case of a single implicit session. This simplification means that the invariant has no universal quantifiers and the proof is much more automatic. We start with a trivial invariant contains type information. We then generate and attempt to prove the refinement proof obligations. Those that cannot be proved lead to a clause in the invariant. The additional invariant clauses result in further proof obligations which may in turn lead to further invariant clauses. In this case a sufficient invariant was constructed in three iterations and the proof was completely automatic (for the case without universal quantification). The invariant is then generalised to multiple sessions and the proof goes through, though not completely automatically. The above refinement indicates that single-place buffers could be refined by multi-places buffers.

The refinement works because the request-response protocol that the client and server follow. Multi-place buffers allow having more than one message at the same time in different buffers. Although in this model message duplication is impossible but due to error and delay in communication links, message duplication is very likely and it could be taken in to account in later refinements. The next step refinement involves splitting each unbounded buffer into three unbounded buffers and introducing new operations for communications between these. These three buffers will be distributed between client, communication and server respectively. This decomposition process is a straightforward task with a simple gluing invariant which states that the order concatenation of the sub-buffers should be equal to the original buffer. Due to space restriction we have not presented this refinement here.

Using sequences to represent communication buffers imposes the order of messages. In other words it assumes that the communication link should guarantee message delivery in the order which they been sent out by sender. This implication could be considered as a restriction and in some cases it might be necessary to use a more general model to represent communication buffers. Therefore a different model based on using unordered multi-place buffers which is

```
INVARIANT
  sreq_buf∈ SESSION ⇸ seq(REQUEST) ∧
  sresp_buf∈ SESSION ⇸ seq(RESPOSE) ∧
  /* Gluing Invariant */
  ∀sid.(sid ∈ fresh_session ⇒ reqsevice_buf(sid) =∅ ) ∧
  ∀sid.(sid ∈dom(sreq_buf) ∧ sreq_buf(sid)≠[] ⇒
      first(sreq_buf(sid))∈ reqsevice_buf(sid) ) ∧
OPERATIONS
 PicService ≙        /* Client Operation */
  ANY sid, req WHERE sid ∈ fresh_session ∧ sid ∉ dom(sreq_buf) ∧
                     req ∈ REQUEST ∧ req≠ none THEN
    sreq_buf (sid):= sreq_buf ⌒ [req] ||
    fresh_session:= fresh_session - {sid}
  END;
 SelectService ≙       /* Server Operation */
  ANY sid,req WHERE sid ∈ session ∧ req ∈ REQUEST ∧ resp ∈ RESPONSE ∧
        sid ∈dom(sreq_buf) ∧ sreq_buf(sid)≠[] ∧ first(sreq_buf(sid)) = req THEN
   session_request(sid):= req ||
   sreq_buf:= tail(sreq_buf) ||
   sresp_buf(sid):= sresp_buf(sid) ⌒ [resp]
  END;
Submit_Servic_Dtail ≙        /* Client Operation */
 ANY sid,resp WHERE sid∈dom(resp_buf) ∧ resp ∈ RESPONSE ∧
                     sresp_buf(sid)≠ [] First(sresp_buf(sid)):= resp THEN
  sresp_buf(sid):=  tail(sresp_buf(sid))
 END;
```

Fig. 13. Refined model with unbounded buffers.

defined in [16] can be used. This unordered buffer had been named as a "*Bag*" which is a collection of elements that may have a multiple occurrences of any element. Due to space restriction we have not presented this solution here.

8 Summary of Results, Conclusions and Further Work

We have identified some key issues in formal modelling of Web-based systems like state representation in server and client side, distributed database system abstraction and refinement, handling complex data types and formal model for communication links. We have proposed some solutions for these aspects which have been exemplified with event-b models of a Travel agency case study.

In formal modelling we have considered only the safety properties and we have not tackled the liveness issue. Although our work has been influenced by mainstream work in Web-based system modelling and implementation, our models require further refinement to implementation level.

Furthermore Web-based systems are constructed from distributed subsystems which could operate concurrently. The fact that complicated nature of such systems could not be completely enclosed by a single B machine, reveals the

importance of decomposition as a next step in formal development process. Decomposition is also an essential strategy for tackling the rapid growth of system complexity. Decomposition strategy could be based on CSP style value passing channels which has been developed in [16] and applied to other types of distributed systems [17].

References

1. Murugesan, S., Deshpande, Y., eds.: Web Engineering, Software Engineering and Web Application Development. Lecture Notes in Computer Science 2016, Springer (2001)
2. S. Murugesan, et al.: Web Engineering: A New Discipline for Development of Web-based Systems. In: Proceedings of the First ICSE Workshop on Web Engineering. LNCS 1189, Los Angeles (1999)
3. Y. Deshpande, et al.: Web Engineering: Beyond CS, IS and SE. In: Proceedings of the First ICSE Workshop on Web Engineering, Los Angeles (1999) 171–176
4. J. R. Abrial: The B book - Assigning Programs to Meanings. Cambridge University Press (1996)
5. E. Sekerinski and K. Sere (eds.): Program Development by Refinement Case Studies Using the B Method. SpringerVerlag (1998)
6. P. Luigia et al.: A Methodology for Integrating of Formal Methods in a Healthcare Case Study. Technical Report 436, TUCS (2001)
7. M. Butler and M. Waldén: Distributed system development in B. In: Proceedings of the 1st Conference on the B Method, Nantes, France (1996) 155–168
8. J.-R. Abrial and D. Cansell: Click'n'Prove- Interactive Proofs Within Set Theory, Version 23 (2003) http://www.loria.fr/ cansell/cnp.html.
9. : (Atelier B Web Page) http://www.atelierb.societe.com/.
10. : (B4free Web Page) http://www.b4free.com/.
11. J.-R. Abrial: Extending B without changing it (for developing Distributed Systems). In Abrias, H., ed.: Proceedings of the 1st Conference on the B Method. (1996) 169–191
12. J.-R. Abrial and L. Mussat: Introducing Dynamic Constraints in B. In: B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method. (1998) 83–128
13. M. Waldén and K. Sere: Reasoning About Action Systems Using the B-Method. Formal Methods in Systems Design **13** (1998) 5–35
14. R. Back and K. Sere: Superposition Refinement of Reactive Systems. Formal Aspects of Computing **8** (1996) 324–346
15. C. Ferreira and M. Butler: Using B Refinement to Analyse Compensating Business Processes. In: ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users. LNCS 2651, Turku, Finland, Springer (2003)
16. M. J. Butler: Stepwise Refinement of Communicating Systems. Science of Computer Programming **27** (1996) 139–173
17. A. Rezazadeh and Michael Butler: Event-Based Modelling and Refinement of Distributed Monitoring and Control Systems. In: Refinement of Critical Systems (RCS'03), Turku (2003)