Reverse Engineering of Event Handlers of RAD-Based Applications

Óscar Sánchez Ramón
University of Murcia
Murcia, Spain
Email: osanchez@um.es

Jesús Sánchez Cuadrado
University of Murcia
Murcia, Spain
Email: jesusc@um.es

Jesús García Molina
University of Murcia
Murcia, Spain
Email: jmolina@um.es

***Abstract*—Businesses are more and more modernising the legacy systems they developed with Rapid Application Development (RAD) environments, so that they can benefit from the new platforms and technologies. When facing the modernisation of applications developed with RAD environments, developers must deal with event handling code that typically mixes concerns such as GUI and business logic. In this paper we propose a model-based approach to tackle the reverse engineering of event handlers in RAD-based applications. Event handling code is transformed into an intermediate representation that captures the high-level behaviour of the code. From this representation, some modernisation tasks can be automated, and we propose the migration to a different architecture as an example. In particular, from PL/SQL code, a new Ajax application will be generated, where business logic, user interface and control code have been separated.**

I. Introduction

The RAD methodology arose in the early 90's as a response to the non-agile development processes widely used in those years [1]. Along with the methodology, a number of integrated environments supporting third and fourth generation languages appeared. Oracle Forms, Visual Basic or Borland Delphi are well known examples of them. They shorten development time by facilitating graphical user interface (GUI) design and coupling data access to GUI components. These tools have been used to develop a great number of desktop applications as part of information systems, but at present companies are facing the migration to modern technologies and platforms that better meet their needs for extensibility and maintenance. Another reason for the migration is that some vendors are increasingly ceasing support in favour of other platforms.

The migration of a legacy system is a common scenario of software reengineering which involves dealing with different aspects of an application such as the GUI or the business logic. In RAD environments, the GUI and the database are linked basically in two ways. Firstly, by setting properties in data source components that connect regular widgets with database tables or columns. As these links are explicit, they can be easily reverse engineered. However, developers often implement event handlers which are attached to widgets that access the database and at the same time manipulate the GUI. This makes migration difficult, in particular to web platforms, since database code cannot be executed in the client side.

As pointed out in [2], coping with the migration of these systems requires disentangling GUI, control code and business logic, so that the new system has a better separation of concerns. Besides, migration would be facilitated by tools that help to discover architectural concerns that are only implicit (and mixed together) in the source code, such as database access, navigation flow, validation or exception handling.

On the other hand, *Model Driven Engineering* (*MDE*) has arisen as a new software development paradigm in which models, which are abstract representations of systems, drive the whole development process. In this setting, model transformations are used to convert models between different levels of abstraction. MDE techniques are not only applicable to the creation of new software systems, but can also be used to evolve existing systems, by automating evolution activities through model transformations. For instance, modeling can be used at the reverse engineering level to define high-level representations of the code, and model transformations can automate the generation of the code of the new system.

At present we are building a reengineering framework based on MDE for applications written with RAD environments, where the reverse engineering, restructuring and forward engineering activities of the reengineering process [3] are applied in a systematic way. In a previous work we dealt with the discovery of the implicit layout of GUIs [4]. In this paper, however, we will focus on reverse engineering event handling code in order to explicitly represent the behaviour of the application and be able to separate the concerns that are entangled. Both are complementary as they deal with different aspects of RAD applications. As far as we know, there are no works dealing with automated migration of event handlers of RAD applications.

We have defined a RAD environment-independent metamodel to represent the original code in a more abstract form, which is based on a set of primitive operations to describe common behaviour of RAD-based code. We have identified a set of recurrent programming idioms for a specific RAD environment (Oracle Forms) and we have mapped these idioms to the RAD representation by means of a model transformation. As an example of the usefulness of the RAD representation, we have also defined a metamodel and a model transformation to separate the source code into three concerns, namely GUI, business logic and control. We have built a prototype implementation to test this approach. It has been validated with a case study based on a real application written

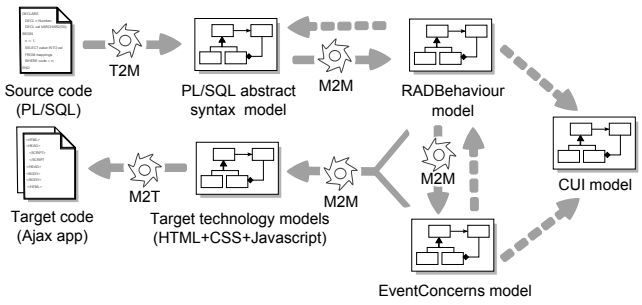

Fig. 1. Model-based architecture for reengineering RAD-based applications. Solid lines mean transformations and dashed lines are model dependencies.

in Oracle Forms, which has been migrated to a client-server web application.

The next section presents our model-based reengineering architecture. Section III introduces a running example based on our case study. In Section IV our platform-independent representation for the source code is introduced. Then, Section V explains how to use the representation to separate code concerns and automatically migrate the original application. An evaluation of the case study is shown in Section VII. Related work is commented in Section VIII, and conclusions and future work are given in the last section.

II. Architecture

Our model-based architecture is shown in Figure 1, exemplified for Oracle Forms as source RAD technology and HTML/Javascript as client-side target platform. Anyway, the approach could be applied to a different RAD technology or target platform likewise. As can be seen it is based on the Horseshoe model [5], which provides a conceptual framework for the different stages of reengineering, focused on architecture recovery and the different transformation steps involved.

First of all, the application of an MDE-based approach in tasks of software modernisation requires a first step aimed at obtaining models from the source code. In order to accomplish this extraction, we use Gra2MoL [6], a domain specific language specially tailored for extracting models from source code that conforms to a grammar. Writing a text-to-model (T2M, also known as code-to-model) transformation in Gra2MoL requires the existence of the grammar of the programming language (in our case PL/SQL) and the corresponding abstract syntax metamodel. As a result of applying Gra2MoL, an abstract syntax model representing the code of the application triggers (event handlers) is obtained. If we wanted to tackle a different RAD environment (e.g. Borland Delphi 5), we would use Gra2MoL with the corresponding grammar (e.g. the Delphi grammar).

The reverse engineering step starts by transforming the abstract syntax model of the event handlers into an intermediate model, named *RADBehaviour*. This model captures the behaviour of the source code in terms of simple primitives which are common in RAD environments, such as read data from a database or write some data in the GUI controls. It is worth noting that the *RADBehaviour* representation is a RAD environment-independent abstraction of the source code. A different model transformation is needed for each RAD technology (e.g. Oracle Forms or Borland Delphi) in order to generate the *RADBehaviour*.

From this model, further reverse engineering can be performed to extract implicit information from the source system. *EventConcerns* is a model derived from *RADBehaviour*, which represents the source code with a kind of control flow graph made up of code fragments. A *code fragment* is a sequence of primitives related to the same category (UI, control, business logic). This is useful to achieve the separation of concerns in the target application.

It is important to note that the intermediate models (*RADBehaviour* and *EventConcerns*) contain cross-references to the model from which they have been derived, in order to trace back the original code when performing forward engineering. In addition, they keep some cross-references to a *Concrete User Interface (CUI) model*, which is a platform-independent representation of a GUI. It includes the visual properties about the GUI elements (widgets, panels, windows and so forth) and also contains information about how the GUI elements are structured in the window. The metamodels and transformations to obtain CUI models are explained in [4].

Based on the presented models, restructuring and forward engineering of the original system to a different architecture are possible. In our case, we have experimented regenerating the original application into an AJAX-based web architecture, with HTML/Javascript in the client-side and Java/JPA in the server-side, but other target platforms are possible.

All the model-to-model (M2M) transformations in the architecture have been implemented with the RubyTL transformation language [7], and code generation has been performed with the facilities of the AGE environment [8].

The next section introduce the example that will be used through the paper to support the explanations.

III. Running example

The running example is based on a real-world application for managing the lifecycle of public grants in a Spanish university, that we have used as a case study. It was implemented in Oracle Forms 6. For illustrative purposes we have simplified and translated into English one of the windows (Figure 2) from the original application.

In the upper part of the window there are several widgets to display general information about the grant, and a tabbed panel is shown below, where can be seen some information about the activities for which the grant is conceived, in addition to the periods when the grant must take place.

We will focus the example on a simplified event handler associated to the only checkbox in the window (named *ACT_MODALITIES*). This checkbox is used to indicate if an activity can have several modalities. When the checkbox is not checked, the *Modalities* tab must be disabled, but this can only be done if there are not periods for that activity. The behaviour of the checkbox is defined in an event handler implemented as a PL/SQL trigger and can be seen in Figure 3.

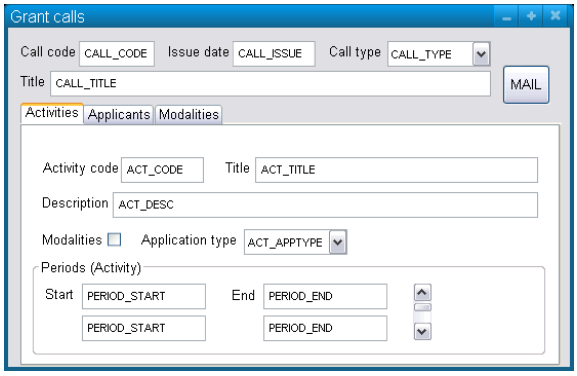

Fig. 2. Grants example

```
IF :ACT_MODALITIES = 'Y' THEN
  SELECT COUNT(*) INTO periods FROM CallPeriods
    WHERE activity = :ACT_CODE;
  IF periods != 0 THEN
    Show_alert('ModalitiesAlert');
    :ACT_MODALITIES := 'N';
  ELSE
    SET_TAB_PAGE_PROPERTY('TABS.MODALITIES',
      ENABLED, PROPERTY_TRUE);
  END IF;
END IF;
```

Fig. 3. PL/SQL trigger for the checkbox change event

The trigger works as follows. The code that is nested in the *IF* statement is executed if the checkbox *ACT_MODALITIES* is checked. It is worth noting that the *'Y'* value is not a predefined value but is specified in the checkbox properties. To check if there are periods for the current activity, a SQL query is used to store the number of periods in the *periods* variable. If there are periods, a pop-up with a message is displayed. Otherwise the tab page is enabled.

IV. Representing event handling code

In this section we will first describe the *RADBehaviour* representation for event handling code. Next, we will comment on the structure of the metamodel, and we will explain how to obtain *RADBehaviour* models from the PL/SQL abstract syntax models.

A. Motivation

In our experience using RAD environments and facing their reengineering, we have found that event handling code of applications in information systems normally has the following characteristics.

- As explained, code managing the GUI is mixed with business logic and database access. There is no clear separation among the different concerns of the application.
- It does not perform complex algorithms or calculations. Event handlers are hardly ever complex, which is caused by the fact that complex functionality is typically implemented in separate functions or stored procedures that are called by the handlers.
- Loops are only used to iterate over database tables. This is a consequence of the previous point, since algorithms used to solve problems are programmed in procedures. Loops are only found when using collections or PL/SQL explicit cursors to iterate over database rows.
- Several levels of nested conditional statements are common, where conditions check values from the GUI or the database. Actions such as updating the GUI or modifying the database are normally performed in the most inner blocks.
- Applications usually repeat a series of idioms. Some of them are specific of each environment, while others are conventions dependant on the company.
- Although each RAD environment has its own programming language to write event handling code, most of them provide similar constructs.

In this setting, we have defined a representation aimed at expressing event handlers in a more abstract form than just the abstract syntax model of the program. It acts as technology-independent pivot model, which allows transformations to ignore technology-specific details in the following steps of the reengineering process. It consists of primitive operations (referred as *primitives* from now on) that intend to represent a wide range of code written in event handlers. This representation brings three main benefits, namely:

- **Simplicity**. Several statements in the original code could be replaced by a few primitive actions that summarise what the statements do. For example, opening a database cursor is a recurrent pattern in PL/SQL, which typically requires around five statements. In our case, from the reverse engineering and restructuring point of view it is enough to know that these statements are just accessing the database. Raising the abstraction level in this way facilitates later processing.
- **Code categorisation**. It facilitates the automated categorisation of pieces of code. In [2] this is regarded as an important activity to disentangle spaguetti code. In this way, it should be possible to differentiate between statements related to the GUI, the control or to business logic. Also, variables uses are explicitly defined (i.e., which primitives are using a certain variable) and they are assigned a category according to their use.
- **Technology independence**. Since there are many RAD environments, most of them providing similar concepts in different flavours, our representation allows us to have the source code expressed in a technology-independent manner. Therefore, reverse engineering algorithms within our framework could be reused across several environments, so promoting extensibility and reusability. Whenever we want to deal with legacy applications developed with different RAD environments, ”only” a transformation to our intermediate representation will be required. Even though this transformation may not be straightforward, reverse engineering and restructuring transformations will be readily applicable, which will certainly pay off.

TABLE I
RAD PRIMITIVES

Primitive	Meaning
ReadFromUI	Reads a value from a widget
WriteToUI	Modifies a widget value
WriteToVar	Writes a value in a global or local variable
ReadFromDB	Reads a value from a database
ModifyUI	Modifies a widget graphical attribute
ManipulateData	Performs an operation on a primitive datatype
SelectionFlow	Selects an execution flow based on some conditions
ExecuteBL	Executes a (user-defined or stored) procedure
OpenView	Opens a specified window
ShowMessage	Opens a modal window (Pop-up)
Leave	Aborts event handler execution

B. Metamodel description

The *RADBehaviour* metamodel is presented in Figure 4. Its main concept is *EventCode*, which is an abstract representation for the code of an event handler. *EventCode*s include information about the type of event and a reference to the widget that originated the event (it is also possible that an event occurs before the window is displayed and therefore has no widget associated). *EventCode*s are grouped into *EventGroup*s which represent the event handlers that are related to the same application window.

The behaviour of every *EventCode* is expressed in terms of a sequence of *RADPrimitive*s. A *RADPrimitive* attempts to replace a statement or a set of statements of the original code, defining what they were intended for. Some of the primitives are listed in Table I. The *input* of a *RADPrimitive* can be another *RADPrimitive* or a variable, so primitives can be composed. The optional *output* must be a variable.

There are several types of variables. *UIVar* is a variable that represents the value contained in a widget. *LocalVar* is a user-defined temporary variable that is just visible in the *EventCode* scope. *GlobalVar* is a user-defined variable that is visible in all the event handlers throughout the application execution. *PredefinedVar* represents any technology-dependant variable that keeps the application status.

Some *RADPrimitive*s, such as *SelectionFlow*, need to specify conditions on their application. These conditions are expressed in terms of a simple expression language, whose base class is *RADExpression*. There are two types of expressions, typical expressions such as *Or*, *And*, *Equals*, and more complex expressions such as *HasData* that checks if a *RADVariable* has a value.

C. Deriving a RADBehaviour model

A *RADBehaviour model* is obtained through a model-to-model transformation that takes an abstract syntax model as input. The transformation matches code patterns and generates a *RADBehaviour model* that summarises the behaviour of the original code. As explained above, event handling code is usually repetitive, and there are some idioms that frequently appear. Conceptually, we have identified three types of idioms in RAD applications.

- **Programming language idioms**. They are facilities provided by the underlying RAD programming language to perform recurrent and/or specialized tasks. For instance, PL/SQL allows special versions of SQL DML statements (e.g., SELECT) to be used within regular imperative code, while Delphi uses data sources configured with queries.
- **Community idioms**. They refer to sequences of statements that are widely accepted by the corresponding community as a good way of doing a particular task. For instance, obtaining and traversing a database cursor. These idioms are typically found in technical documentation.
- **Business-dependant idioms**. These are idioms that are originated from the company conventions and practices. Available knowledge about the way of work in business can be expressed with patterns that could highly improve the reverse engineering process.

It is possible that several idioms match the same code snippet, so a priority criterion is followed to decide which of them must be selected. Business-dependant idioms have the highest priority, followed by the community idioms, while programming language idioms have the lowest priority. Note that some of the idioms can be composed of other idioms, and in this case the same priority criterion is followed.

The transformation to derive a *RADBehaviour model* must be implemented specifically for each different RAD environment. At present we have a model-to-model transformation that supports Oracle Forms PL/SQL. We have separated transformation modules to deal with each type of idiom independently, so that they can be extended or replaced seamlessly. This is particularly useful in the case of business-dependant idioms, that may need to be adapted for a specific company.

Figure 5 shows some mappings between PL/SQL idioms and *RADBehaviour* primitives (expressed with a textual notation only for illustrative purposes). We have followed this nomenclature: *x* and *y* are variables (*LocalVar* or *UIVar*), *s1* and *s2* are strings, *p1* and *p2* are predefined properties, *v1* and *v2* are specific values for these properties, *c1* and *c2* are table columns of a table *t*, *c* is a PL/SQL explicit cursor, and *d1* and *d2* are Forms datablocks (a logical group of widgets linked to the database). The meaning of each construct can be easily deduced from the notation.

Mappings *M1* to *M4* are *programming language idioms*, PL/SQL idioms in this case. If a variable name starts with ":" (mapping *M1*), then it refers to a widget value and it must be mapped to a *WriteToUI* primitive, otherwise it would be mapped to a *WriteToVar*. Mappings *M2* and *M3* are library functions to show a message box and to change tab page respectively. It is worth noting that the translation of *M3* consists of creating a ModifyUI action that refers to the corresponding CUI model, which was obtained in a previous transformation.

On the other hand, mappings *M5* and *M6* are *community idioms* as they are recommendations typically followed by

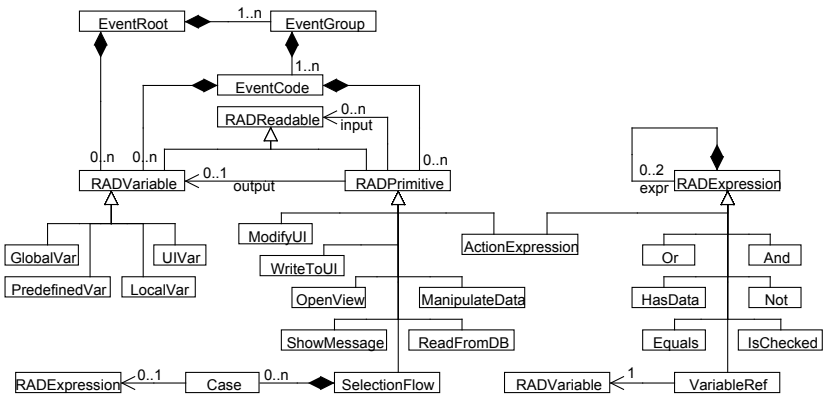

Fig. 4. Excerpt of the RADBehaviour metamodel.

PL/SQL idioms	RAD Behaviour mappings
M1 :x := <function>;	**WriteToUI**(*output*=x, *input*=<function>)
M2 Show_alert(s1);	ShowMessage(*message*=s2) (*s2* is a message associated with the *s1* alert)
M3 Set_tab_page_property(x,p1,v1);	**ModifyUI**(input=x, property=p2, value=v2); (*p1*, *v1* are mapped to *p2*, *v2*)
M4 **SELECT** c1 **INTO** x **FROM** t **WHERE** c2 = y;	**WriteToVar**(*output*=x, *input*=**ReadFromDB**(*input*=y, *table*=t, *col*=c1, *cond*=c2=y)
M5 **IF** <cond> **THEN** x := Find_relation(s1); Query_master_details(x, d2); **END IF**; (The trigger is in datablock *d1*, which is different from datablock *d2*)	{ **WriteToUI**(*output*=y, *input*=**ReadFromDB**(<dbdata>)) }1..n (*y* is a widget of the *d2* datablock, <dbdata> is obtained from datablock *d1* and the properties of the relation *s1*)
M6 **OPEN** c; **FETCH** c **INTO** x; **IF** (c%FOUND) **THEN** <statements> **END IF**; **CLOSE** c;	**SelectionFlow** **Case**(*condition*=**HasValue**(**ReadFromDB**(<dbdata>))) <primitives> (<dbdata> is obtained from the cursor declaration)
M7 x := Name_in (:SYSTEM.TRIGGER_ITEM || '_Value')	**WriteToVar**(*output*=x, *input*=**ReadFromUI**(*input*=y)) (*y* is a widget whose name is the name of the widget associated to the trigger plus '_Value')

Fig. 5. PL/SQL to RADBehaviour mappings

PL/SQL programmers, in this case to fill a master/detail relationship and to manipulate a database cursor respectively. These idioms have a coarser-grained granularity than the previous ones.

Mapping *M7* is a *business-dependant idiom* to write generic event handlers, which is based on Oracle Forms reflective facilities to manipulate the GUI. The *:SYSTEM.TRIGGER_ITEM* special variable contains the name of the widget that is the source of the event that has lead the event handler (trigger in PL/SQL terminology) to be executed. The *Name_in* function takes a widget name as a parameter and returns its value. Therefore, M7 is mapped to a WriteToVar whose input is the value of a widget whose name is the same as the one triggering the event plus "_Value". In this way, using naming conventions(for example having a widget *X* and a related widget *X_Value*) it is possible to write generic event handlers that can be executed for different widgets. Translating this idiom requires embedding the convention into a specific transformation. Besides, the outcome of the transformation is not a single reference to a widget, but it computes every possible widget that could be read (looking for widgets in the CUI model that match the pattern). This uncovers widget relationships that were implicitly specified in the source code.

Finally, some idioms and statements cannot be translated without additional information, because Oracle Forms allows the developer to declaratively specify some behaviour by means of property sheets, i.e., without writing code. For example, there is a function named *execute_query()* that executes a database query defined for the current data block, and fills in all the widgets that are related to this data block and are linked to database columns. Therefore, our model transformation also takes as input this information (gathered from a *Oracle Forms model*, not shown in the architecture diagram due to space reasons) in order to deal with this kind of functionality.

D. Example

The fragment shown in Figure 6 is the *RADBehaviour* representation of the PL/SQL code for the checkbox change event that was introduced in Figure 3. The same textual notation as in Figure 5 is used.

The outmost *IF* statement, whose condition is that the checkbox is checked, has been replaced for a *SelectionFlow*. We must remark that the *RADBehaviour model* captures explicitly the *IsChecked* condition, while in the original code this condition is not clearly expressed since the checked value (*'Y'* in our case) is not predefined, but defined by the programmer in the checkbox property sheet.

The *Select* statement that counts the number of periods has been replaced with a *WriteToVar*, composed of a read from the database (*ReadFromDB*). The inner *IF* statement becomes a *SelectionFlow* which has two cases. The first case shows a message to the user if there are some periods left

```
SelectionFlow
  Case(condition=IsChecked(UIVar(name=ACT_MODALITIES)))
    WriteToVar(output=LocalVar(name=periods),
               input=ReadFromDB(input=UIVar(name=ACT_CODE)
                                table=CallPeriods,
                                isCount=true))
    SelectionFlow
      Case(condition=HasData(LocalVar(name=periods))
        ShowMessage(msg="...")
        WriteToUI(output=UIVar(name=ACT_MODALITIES),
                  input=Literal(value=true))
      Case
        ModifyUI(input=TABS.MODALITIES,
                 property=enabled, value=true)
```

Fig. 6. RADBehaviour example for the checkbox event

(*ShowMessage* is enclosed in a *WriteToVar* since a pop-up could allow the user to perform some actions) and sets the checkbox as checked. The second case modifies a predefined property (*enabled*) from the widget *TABS.MODALITIES*.

V. Separating concerns

As explained in Section I dealing with the migration of applications written with a RAD environment requires disentangling GUI, control and business logic. Therefore, our aim is to automatically categorise fragments of code where statements of each fragment are related to the same concern.

In order to achieve this goal, we have defined a metamodel (named *EventConcerns*) that represents fragments and their categories. It is obtained from a *RADBehaviour* model through a model transformation. This transformation is facilitated by the fact that we are not dealing directly with source code, for two main reasons: i) as the source code is represented with a few primitives we just need to check the type of the primitive and sometimes the variables that it uses, so limiting the number of cases that must be handled, ii) given that every primitive represents its input and output explicitly, establishing variable dependencies between primitives can be easily done.

Next we will introduce the *EventConcerns* representation.

A. EventConcerns representation

In this representation each event handler is represented as a kind of control flow graph, where the nodes are basic blocks [9] and the edges are execution flows. Interestingly, basic blocks are composed of fragments, where a fragment is defined as a sequence of primitives classified in the same category. So far, we have considered three categories: user interface, control and business logic, and we plan to further refine this categorisation in the future.

Figure 7 shows a graphical rendering of the *EventConcerns model* derived from the *RADBehaviour* model shown in Figure 6. In the example there are four basic blocks represented as rounded boxes with two compartments: a upper compartment that shows the descriptive name given to the block and a lower compartment that includes the sequence of fragments for that block. Fragments are represented with roundes boxes that indicate the type of fragment and a descriptive name.

The metamodel for this representation is shown in Figure 8. The types of nodes in the graph are: *BasicBlocks*, *EntryNode*

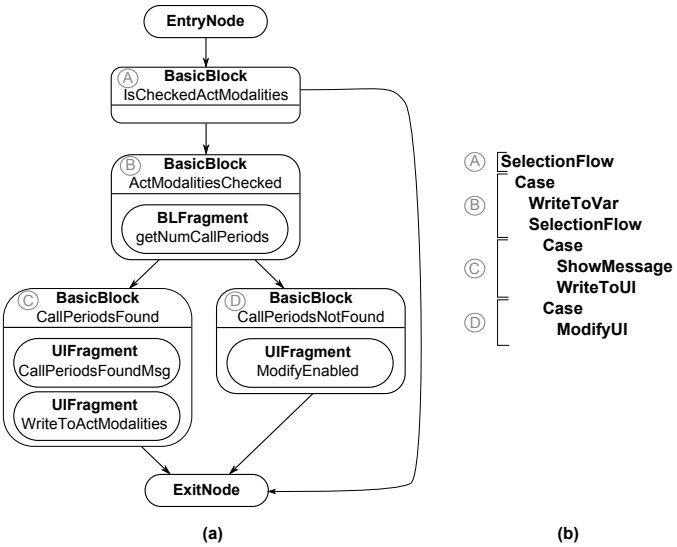

Fig. 7. EventConcerns model derived from the model in Figure 6. Labels *A, B, C, D* are used to show the primitives that originate the basic blocks.

that is a unique node that refers to the first basic block and *ExitNode* that is a unique node that represents the end of the execution flow. A basic block is composed of code fragments that can be of three types:

- *UIFragment*s contain primitives that read some data from the interface, or perform a change in the GUI (e.g. show a pop-up, change the value of a text field or change the background colour of the widget that has got the focus).
- *BLFragment*s represent code that performs some kind of calculation or information processing (which is commonly done by calling a function that implements the required functionality), or is code related to data persistence.
- *ControlFragments* are used to represent those primitives that are neither user interface nor business logic related and affect the status of the application. For example, set a user identifier in a global variable that is used throughout the user session.

It is worth noting that *Fragment*s keep references to *RADBehaviour* primitives (i.e., instances of the *RADPrimitive* metaclass). Also, for each basic block we keep the set of input variables (*inputVars*) and output variables (*outputVars*), which are obtained by joining the input and output (respectively) of the single primitives. This will be useful to identify variable dependencies among the fragments.

The nodes of the graph are linked by means of edges (*FlowEdge*s) that allow us to navigate through the graph. When there are alternative paths from a node, each edge has a *condition* associated. We have another relationship for code fragments named *dependencies*, which is based on the idea that a fragment can depend on previous fragments. Particularly, when a fragment *f1* assigns a value to a variable that is read in another fragment *f2* that can be reached from *f1*, then *f2* depends on *f1*.

Each code fragment is given a significant name that is

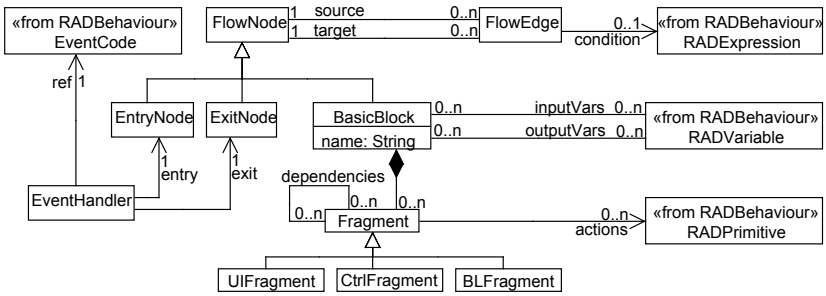

Fig. 8. Excerpt of the EventConcerns metamodel

inferred from the statements of the block due to it can be useful later, for example to generate methods from the fragments.

B. Fragment identification

In this section we will explain how can we use the *RAD-Behaviour* representation to obtain *EventConcerns models*. We have split the transformation in several phases that are described next.

1) Create a control flow graph of fragments: The following algorithm is based on the basic block partitioning algorithm that can be found in [9]. According to this algorithm, a **basic block** is a sequence of instructions which are executed from the first one to the last one without performing jumps. The first instruction of a basic block is called **leader**.

1) Create the *EntryNode* and the *ExitNode*. Create a *BasicBlock* that refers to the *EntryNode*.
2) Iterate over the primitives from the first one.
   2.1) If the primitive is different from a *SelectionFlow* and it is a leader, then we create a *BasicBlock* based on the primitives of the basic block where the primitive is leader. For each primitive *p* do:
   a) If there are no fragments, create a fragment that contains *p*.
   b) If there are fragments, check if *p* fits any of them (this means that *p* has the same category than the primitives in the fragment and at least one of the input variables of *p* is the output of another primitive that is executed before). If *p* fits one or more fragments, add *p* to the fragment *f* that is closer to the primitive (i.e. the fragment that contains a primitive which is closest to *p*). Add the output variable of *p* to the output of *f*.
   c) If *p* does not fit any fragment, create a new fragment *f* and add *p* to it. The type of the fragment (*UIFragment*, *BLFragment*, *CtrlFragment*) depends on the type of the primitive. Add the output variable of *p* to the output of *f*. Add *f* to the current *BasicBlock* (they are ordered following the creation order).

   2.2) If the primitive is a *SelectionFlow* primitive, create a branch in the graph for each *Case*. This is, create a new *BasicBlock* and an *EdgeFlow* from the current BasicBlock to the new one. The *condition* of the *EdgeFlow* is referred to the *condition* of the *Case*.
3) For all the *BasicBlock* without outgoing links, create a *EdgeFlow* that refers to the *ExitNode*.

The algorithm uses a function that determines the category of the primitive based on its type and input variables. For example, a *WriteToVar* whose input is a *UIVar* will belong to the UI, but if we had *WriteToVar* whose input is a *ReadFromDB*, then the primitive will be tagged as a BL concern.

We can see for example in Figure 7 that the basic block *A* has no fragments since it has just been derived from a *SelectionFlow*, and that block *C* includes two *UIFragments* due to there are no variable dependencies between the *ShowMessage* and the *WriteToUI*.

It is not an optimum algorithm in the sense that it can generate several fragments for UI primitives that refer to the same widget. Anyway, it is not a real problem since contiguous fragments of the same type can be treated as if they belonged to the same fragment when generating code.

2) Give a descriptive name to the fragments: This is an important step which will be useful to generate code, particularly the name of the *BLFragments* can be used to generate the name of the business logic methods. Moreover, giving a meaningful name to the fragments allows capturing the semantics of a fragment code, which is useful as documentation of the original system. However, it is not always possible to infer a useful description for the fragment.

We assign names to the fragments based on heuristics. In many cases, *BLFragments* perform some operations on the database after reading the value of some widgets. In these cases, we give a name to the fragment by looking at the database access primitives and ignoring the rest of them. For example, the *ReadFromDB* primitive that appears in Figure 6 comes from the *SELECT* statement in Figure 3, and the name generated from this primitive is *getNumCallPeriods*, as can be seen in the *BLFragment* of block *B* in Figure 7 (note the infix *Num* that indicates that the operation returns a number). When we have that a *BLFragment* invokes a function or procedure, we take the first invocation as a name.

A *UIFragment* often refers to just one primitive, so in that cases we obtain the name based on the primitive. For example, a fragment with a *WriteToUI* primitive is named with *WriteToX* where *X* is the widget that is being written, for example the second *UIFragment* in block *C* is called *WriteToActModalities*.

The name of a *BasicBlock* that starts with a *SelectionFlow* is the name of the condition of the *Case*, taking into account previous primitives that are referred by this condition. For example, in block *A* the condition of the *SelectionFlow* is an *IsChecked* expression that does not depend on previous primitives (actually the *SelectionFlow* is the first primitive), so the inferred name for the block is *IsCheckedActModalities*.

The name of a *BasicBlock* which is the first block in a branch uses the branch condition and previous primitives that are referred by this condition. For example, the name of the block *C* is *CallPeriodsFound*, which is derived from the condition and the *WriteToVar* that precedes the *SelectionFlow*.

3) Setting dependencies among fragments: Code fragments often depend on some values that where calculated or retrieved in other fragments which were executed before, so it is interesting to explicitly capture these relationships. To know the dependencies, we must identify the set of input and output variables for each fragment, which is easily done by using *input* and *output* attributes of the primitives. Then we set the dependencies according to this criterion: A fragment *f1* depends on another fragment *f2* if the input set for *f1* includes some variables from output set of the fragment *f2*.

VI. Generating code

In this section we will outline the last part of the architecture proposed in Figure 1, that is, how a *EventConcerns model* can be used to generate a part of the new system.

Separating the different concerns of the legacy system allows us to migrate the application to a new platform and technology, especially to some web technologies where the separation between UI and business logic is imposed.

We have built a chain of model-to-model and model-to-text transformations that migrates PL/SQL event handlers to a heavy-client, two tier architecture, where the GUI is defined with HTML/Javascript/jQuery which invokes a REST service made up of business logic fragments. We have defined several metamodels to represent the target architecture, which comprise the several technologies involved: HTML and jQuery (Javascript) for the client side and Java for the server side.

The model-to-model transformation takes the abstract syntax model of the PL/SQL code as input and outputs one or more models representing the target artefacts. The transformation between snippets of PL/SQL to either Javascript or Java is relatively straightforward. The main challenge is how to decide which parts of the PL/SQL must be transformed to Javascript (UI) or to Java (business logic). To this end, the transformation also takes the *EventConcerns model* as input. The *EventConcerns model* actually drives the transformation in the sense that it is used by the transformation rules to disentangle the original code by changing and relocating the content of a fragment according to its category. The cross-references between a fragment and the *RADBehaviour model* are essential to achieve this effect.

Listings 1 and 2 show the translation of the original code of the running example (Figure 3). We do not go into details of the transformation and the code for space reasons, but there are three issues which are worth noting, namely:

- First of all, it is possible to some extent generate idiomatic code because the *RADBehaviour* model contains certain semantic information. For instance, line 2 checks whether a checkbox is checked or not in idiomatic jQuery.
- Secondly, the generated UI code in Javascript has the same shape as the original PL/SQL code, except business logic fragments, which are translated to a remote AJAX call. In Javascript a callback is executed when the result is available, so every fragment (UI or BL) which depends on the transformed logic fragment is put within such a callback (lines 7-14). Currently, we only support synchronous calls, but we intend to develop another transformation which will be able to perform asynchronous calls based on the dependencies among fragments.
- Finally, each business logic fragment is mapped to a Java method which connects to the database and performs the required logic. The input parameters of this method are the UI variables that the fragment depends on, and the returning value is a JSON object made up of the variables (UI or local) used by other fragments that depend on this fragment.

```
1  var periods;
2  if ( $('#act_modalities').is(':checked') ) {
3    $.ajax({
4      url: "getNumCallPeriods/" + $('#act_code').val(),
5      dataType: "json",
6      async   : false,
7      success : function(result) {
8        periods = result.periods
9        if ( periods !== 0 ) {
10         alert('No periods');
11         $('#act_modalities').attr('checked', true);
12       } else {
13         $('#act_modalities').tabs('enable', 1);
14       }
15     }
16   });
17 }
```

Listing 1. Event handling code rewritten in Javascript

```
1  @Produces("application/json")
2  public class Service extends BaseResource {
3    private static EntityManager em = ...;
4
5    @GET @Path("grants/getNumCallPeriods/{act_code}")
6    public Representation getNumCallPeriods(
7          @PathParam("act_code") String code) {
8      Query q = em.createQuery("SELECT COUNT(*) FROM
           CallPeriods WHERE activity = :ACT_CODE");
9      q.setParameter("ACT_CODE", code);
10     return new JSONObject().put("periods", q.getSingleResult());
11   }
12 }
```

Listing 2. Entangled business logic moved to REST service

VII. Evaluation of the approach

In order to assess the utility of our approach we have performed a case study. As explained, it is a management application targeted at being used to manage research projects and grants that are assigned to research groups of Spanish universities. It is a real-world application which was developed in Oracle Forms 6 by an industrial partner and is composed of

TABLE II
RADBEHAVIOUR EVALUATION

LOC of idioms matched / total LOC	95.65%
LOC mapped OK / total LOC	83.04%
LOC of matched programming idioms / total LOC	36.67%
LOC of matched community idioms / total LOC	56.45%
LOC of matched business idioms / total LOC	6.88%

TABLE III
EVENTCONCERNS EVALUATION

LOC classified OK	86.10%
LOC classified as BL	15.87%
LOC classified as Ctrl	4.80%
LOC classified as UI	79.33%

107 windows. Around 11000 LOC were evaluated (comments are not counted), what indicates a medium-high complexity.

We have executed the complete reverse engineering process for the application and we have manually inspected the models in order to count the lines of code (LOC[1]) correctly matched and classified. For the *RADBehaviour* model, we count the LOC that have been successfully matched, comparing the idioms matched with the expected ones. For the *EventConcerns model* we count the LOCs that have been classified in each category, in order to assess the amount of code that our approach is able to relocate.

A. RADBehaviour evaluation

The results are shown in Table II. *LOC of idioms matched* is the percentage of LOCs out of the total that have been matched with some idiom. However, not all LOCs that are matched are mapped properly, so *LOC mapped OK* is a measure of the amount of code whose behaviour has been captured right.

As can be seen, almost all LOCs match some idiom. This is a consequence of having fine-grained idioms (programming language idioms) that match almost everything that coarse-grained idioms (community and business idioms) cannot. This avoids the need for writing idioms for every built-in function, and it offers a migration option when some coarse-grained idioms have not been identified. For example, there is a built-in function that copies a value to a given variable if the current value of the variable is NULL. Since we do not have a specific mapping for this function, it is automatically transformed into a *ExecuteBL*, which is a wrong mapping. When there are statements that are not matched, they are notified to the user.

As can be seen, almost 17% of LOC are mismatched. In our case, the majority of the fails are due to the fact that we do not deal with PL/SQL exceptions, and because of some specific Forms functions that are not mapped properly. We can conclude that the set of primitives identified in *RADBehaviour* is enough to capture the basic behaviour of the application.

The second part of Table II shows the percentage of each type of idiom that has been matched out of total of correct matches. This reinforces the idea that RAD applications are programmed based on a set of more or less fixed idioms that are used throughout the code given that approximately 63% of the code are coarse-grained idioms (i.e., community and business idioms).

[1]Tokens like *begin* or *end*, and variable declarations are not counted.

B. EventConcerns evaluation

Table III shows the amount of LOCs that has been classified in each category (user interface, control and business logic). *LOC classified OK* shows the number of LOCs out of the total that have been categorised properly. Interestingly, the success percentage in this case (86.10)% is slightly higher than the percentage of code well mapped when obtaining the *RADBehaviour* (83.04%, *LOC mapped OK* in Table II). This is due to the fact that some original statements are mapped to wrong primitives, but by chance they belong to the right category, so they are classified correctly. However, this may lead to generate a wrong piece of target code (i.e., around 3% of the generated code is wrong). We are looking into ways of detecting this corner case.

It can be seen that a certain amount of the code (20.67%) should be relocated to achieve separation of concerns, what shows that RAD applications are tightly coupled, and that our approach facilitates identifying fragments related to each concern and automatically relocating them.

With regard to code classified as UI (79.33%), it is translated in a straightforward manner to the new application. However, we have estimated that around 18% out of UI code is in charge of performing interactions among widgets or performing a change in the navigation flow of the application, and could be moved to a different module if we intended to decouple the interactions among widgets. Based on the *RADBehaviour* representation it is possible to identify those widgets interacting with other GUI elements, so enabling further separation of concerns.

VIII. RELATED WORK

Reverse engineering of legacy systems has been tackled using static and dynamic techniques. For instance, in [10] static analysis based on SSA is used to obtain a graph that represents GUI window relationships. In [11], a wrapper-based approach is proposed to make legacy systems available as web services. In [12] user interactions are dynamically traced to provide a web access to the original application.

In [13] a commercial tool for migrating Oracle Forms applications to the .NET platform is presented. The source platform events for which there exists a semantically similar mapping in the target platform are directly migrated, while the rest must be supported by the developer. With respect to our approach, they use neither models nor another intermediate representation and the source and target of the migration are fixed. In [2] the evolution of legacy systems to three-tier SOA systems is tackled. In this proposal the code fragments are manually tagged with the aspect they are related (interface,

business logic or data), whereas our proposal automatically inferes these aspects from the *RADBehaviour*.

Knowledge Discovery Metamodel (*KDM*) [14] is a common intermediate representation proposed by the OMG for existing software systems and their operating environments, which defines common metadata required for deep semantic integration of Application Lifecycle Management tools and which was defined for software modernization. This metamodel includes a code package for representing platform-independent programming language code, so there are some similarities between this metamodel and our *RADBehaviour*. KDM can be used to represent any programming language so in cases it can be too general, requiring extensions that break interoperability for particular cases. In contrast, our *RADBehaviour* has been devised to be used in the domain of RAD applications, and it expresses the behaviour of the applications more consisely. In [15], KDM models are used to reverse engineer business processes using model transformations. However, the translation is straightforward and the obtained processes are too low-level thus requiring manual adaptation.

In [16], user interface reverse engineering is proposed with the goal of performing interface tests. In this work an *Event Flow Model* is presented, which represents all the possible event execution paths that can occur in application windows. The reengineering process is focused on deducing the dependencies between windows that are caused by events. Our representation can be used to construct such models.

There are some works that deal with the reverse engineering of code to generate models. In [17] the authors propose an algorithm for the automatic extraction of UML interaction diagrams from C++ code based on static analysis. They create an Object Flow Graph and perform a flow propagation inside the graph. In [18] it is introduced a number of heuristic mapping rules for reverse engineering UML class models from C++ source code, which are based on domain knowledge of the C++ language. These approaches are focused on generating models from the analysis of objects, types and method invocations whereas our approach is focused on understanding fragments of event handlers.

IX. Conclusion and future work

In this work a model-based architecture to reverse engineer event handlers of applications developed with RAD environments has been presented. A platform-independent representation for event handler code has been proposed, showing how it can be used to perform additional reverse engineering and restructuring tasks. In particular, a control flow graph of fragments has been obtained, which categorise the source code and enable further transformations to achieve a better separation of concerns in the new system. To assess the approach we have tackled a study case in which event handlers of a Oracle Forms application have been migrated to an Ajax web application. Besides, our tooling is available at http://modelum.es/guizmo for testing purposes.

As future work, we will refine our code categorisation with categories such as navigation or exception handling. In this sense, our aim is to take advantage of our intermediate representations to make more information explicit, such as widget dependencies and navigation flows. Finally, we plan to define a language to specify idioms and their transformation, so they are not hard-coded in model transformation anymore.

Acknowledgment

This work has been supported by Spanish Ministry of Science and Innovation (grant TIN2009-11555) and Regional Government of Murcia (grant 15389/PI/10).

References

[1] J. Martin, *Rapid application development*. Macmillan Publishing Co., Inc., 1991.

[2] R. Heckel, R. Correia, C. M. P. Matos, M. El-Ramly, G. Koutsoukos, and L. F. Andrade, "Architectural transformations: From legacy to three-tier and services," in *Software Evolution*, 2008, pp. 139–170.

[3] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, pp. 13–17, 1990.

[4] O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina, "Model-driven reverse engineering of legacy graphical user interfaces," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10, 2010, pp. 147–150.

[5] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[6] J. L. Cánovas Izquierdo and J. G. Molina, "A domain specific language for extracting models in software modernization," in *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, 2009, pp. 82–97.

[7] J. Sánchez Cuadrado and J. García Molina, "Modularization of model transformations through a phasing mechanism," *Software and System Modeling*, vol. 8, no. 3, pp. 325–345, 2009.

[8] J. Sánchez Cuadrado and J. García Molina, "Building domain-specific languages for model-driven development," *IEEE Software*, vol. 24, no. 5, pp. 48–55, 2007.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[10] S. Staiger, "Reverse engineering of graphical user interfaces using static analyses," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 189–198.

[11] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, "A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures," *J. Syst. Softw.*, vol. 81, pp. 463–480, April 2008.

[12] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuk, "Reverse engineering legacy interfaces: An interaction-driven approach," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, ser. WCRE '99, 1999, pp. 292–.

[13] L. F. Andrade, J. Gouveia, M. Antunes, M. El-Ramly, and G. Koutsoukos, "Forms2net - migrating oracle forms to microsoft .net," in *GTTSE*, 2006, pp. 261–277.

[14] OMG, *Knowledge Discovery Meta-Model (KDM) v1.0*, http://www.omg.org/spec/KDM/1.0/, 2008.

[15] R. Perez-Castillo, I. Garcia-Rodriguez de Guzman, M. Piattini, and A. S. Places, "A case study on business process recovery using an e-government system," *Software: Practice and Experience*, vol. Online Preview, 2011.

[16] A. M. Memon, "An event-flow model of gui-based applications for testing: Research articles," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[17] P. Tonella and A. Potrich, "Reverse engineering of the interaction diagrams from c++ code," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03, 2003, pp. 159–.

[18] A. Sutton and J. Maletic, "Mappings for accurately reverse engineering uml class models from c++," in *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005, pp. 175–184.