Ampersand
Event-Condition-Action Rules

Test Plan

Yuriy Toporovskyy, Yash Sapra, Jaeden Guo

CS 4ZP6
October 30th, 2015
Fall 2015 / Winter 2016

Table 1: Revision History

Author	**Date**	**Comment**
Yuriy Toporovskyy	27 / 10 / 2015	Reorganized document
Yuriy Toporovskyy	27 / 10 / 2015	Initial version - template

Contents

Chapter 1

General Information

1.1 Purpose

This document outlines the test plan for ECA for Ampersand, including our general approach to testing, system test cases, and a specification of methodology and constraints. This test plan specifically targets our contribution to Ampersand, namely ECA – elements of Ampersand, such as design artifact generation, will not be tested.

1.2 Objectives

Preparation for testing

The primary objective of this test plan is to collect all relevant information in preperation of the actual testing process, in order to facilitate this process.

Communication

This test plan intends to clearly communicate to all developers of ECA for Ampersand their intended role in the testing process.

Motivation

The testing approach is motivated by constraints and requirements outlined in the Software Requirements Specification. This document seeks to clearly demonstrate this motivation.

Environment

This test plans outlines the resources, tools, and software required for the testing process. This includes any resources needed to perform automated testing.

Scope

This test plan intends to better describe the scope of our contribution, ECA, within Ampersand.

1.3 Acronyms, Abbreviations, and Symbols

SRS Software Requirements Specification. Document regarding requirements, constraints, and project objectives.

ECA Rule Event-Condition-Action Rule. A rule which describes how to handle a constraint violation in a database. See SRS for details.

HUnit A Haskell library for unit testing. See TODO: ref test tools

QuickCheck A Haskell library for running automated, randomized tests. See TODO: ref test tools

Chapter 2

Plan

2.1 Software Description

Ampersand is a software tool which converts a formal specification of business entities and rules, and compiles it into different design artifacts, as well as a prototype web application.

This prototype implements the business logic in the original specification, in the form of a relational database with a simple web-app front-end.

A particular class of relational database violations can be automatically restored; the algorithm for computing the code to fix these violations is called AMMBR [Joo07]. This class of violations is realized within Ampersand as ECA rules – our contribution to Ampersand will add support for ECA rules, in both the Ampersand back-end and the generated prototype.

2.2 Test Team

The test team which will execute the strategy outline in this document is comprised of

- Yuriy Toporovskyy
- Yash Sapra
- Jaeden Guo

2.3 Test Schedule

Chapter 3

Methods and constraints

3.1 Methodology

3.2 Test tools

3.2.1 Static Typing

Programming languages can be classified by many criteria, one of which is their type systems. One such classification is static versus dynamic typing. Our implementation language, Haskell, has a static type system. Types will be checked at compile-time, allow us to catch errors even before the code is run, reducing the errors that need to be found and fixed using testing techniques.

3.2.2 Formal verification

A part of our project deals with generating source code annotated with the proof of derivation of that source code, which will act as a correctness proof for the system. In particular, when we generate code to restore a database violation using ECA rules, then the generated code will have a proof associated with it, which details how that code was derived from the original specification given by the user.

3.2.3 Random Testing

Random testing allows us to easily run a very large number of tests without writing them by hand, and also has the advantage of not producing biased test cases, like a programmer is likely to do.

We will be using QuickCheck for random testing. The existing Ampersand code base using QuickCheck for testing, therefore, using QuickCheck has the added benefit of easier integration with the existing Ampersand code base.

QuickCheck allows the programmers to provide a specification of the program, in the form of properties. A property is essentially a boolean valued Haskell function of any number of arguments. QuickCheck can test that these properties hold in a large number of randomly generated cases. QuickCheck also takes great care to produce a large variety of test cases, and generally produces good code coverage.

3.2.4 Unit Testing

Unit testing is comprised of feeding some data to the functions being tested and compare the actual results returned to the expected resultd. We will be using HUnit for unit testing of the new source code in Ampersand. HUnit is a library providing unit testing capabilities in Haskell. It is an adaption of JUnit to Haskell that allows you to easily create, name, group tests, and execute them.

3.3 Requirements

3.3.1 Functional requirements

The functional requirements for ECA for Ampersand are detailed in the SRS; they are also briefly summarized here. Our implementation must

F1 provably implement the desired algorithm.

F2 accept its input in the existing ADL file format.

F3 produce an output compatible with the existing pipeline.

F4 be a pure function; it should not have side effects.

F5 not introduce appreciable performance degradation.

F6 provide diagnostic information about the algorithm to the user, if the user asks for such information.

3.3.2 Non-Functional requirements

The functional requirements for ECA for Ampersand are detailed in the SRS; they are also briefly summarized here. Our implementation must

N1 produce output which will be easily understood by the typical user, such as a requirements engineer, and will not be misleading or confusing.

N2 be composed of easily maintainable, well documented code.

N3 compile and run in the environment currently used to develop Ampersand.

N4 annotated generated code with proofs of correctness or derivations, where appropriate.

N5 automatically fix database violations in the mock database of the prototype.

3.4 Data recording

3.5 Constraints

3.6 Evaluation

Chapter 4

System Test Descriptions

T1 EFA Black box test

Test type	Black box/Functional, using dynamic analysis
Schedule	January 2016
Requirements	Functional

Input

The input shall consist of PAClauses (i.e. a data type) that is composed of ECA rules. These ECA rules consist of a condition (i.e. ecaTriggr) that initiates a set of actions to be taken (i.e. ecaAction) based on the violation (i.e. ecaDelta). Please see example below for further elaboration.

Output

The functional output shall be a SQL command template generated through a Haskell script, for each type of ECA violations

Procedure

QuickCheck is used to test the functionality of individual functions; it is a package that provides a library for testing program properties. The programmer is able to provide properties they want tested in their program specification, and QuickCheck generates numerous random cases to test that the property holds[hac].

Example 1.

Input ECA Rule: No action needs to be taken when a data relation is deleted because an order placed by a client has been canceled.

```
ECA { ecaTriggr = On Del rel_orderedBy_Order_Client
, ecaDelta  = vio_Delta_Order_Client
, ecaAction = Nop []
, ecaNum    = 2
}
```

Brief Explanation of Example 1:

The ecaTriggr specifies an event that takes place such as the deletion of a Client's order; the action to be taken is "Nop[]", which means no (mathematical) operations is to be performed on the data set. The ecaNum = 2, indicates the ECA rule that has been violated which is number 2. The deletion of data must be consistent and take place across all regions which it will affect, for example of c = (a,b) where a is an element in set a ($a \in A$) and b is an element in set B ($b \in B$), c may no longer exist. For this example, we shall call c, cost which is the tuple (a,b) where a is an item and b is the cost per weight unit of the item. Both a and b are needed to determine the cost because cost is based on the item and its weight per unit. If the weight no longer exists (i.e. it was deleted), the data table cost (c) no longer shows proper cost for the item in question. If the cost table was entirely devoted to tuples of set A and B, then the entire table needs to be adjusted for a new set of weights or the table must be deleted. EFA will output SQL commands to delete the necessary components.

Example 1. Possible Recursive Deletion SQL Output: Please note that this is a rolled out generic example of what EFA may produce, the actual code EFA provides may be very different.

```
CREATE OR REPLACE PROCEDURE delete_cascade(
table_owner          VARCHAR2,
parent_table         VARCHAR2,
where_clause         VARCHAR2
) IS
-- ex:  execute delete_cascade('MY_SCHEMA', 'MY_MASTER', 'where ID=1'); */

child_cons     VARCHAR2(30);
parent_cons    VARCHAR2(30);
child_table    VARCHAR2(30);
child_cols     VARCHAR(500);
parent_cols    VARCHAR(500);
delete_command VARCHAR(10000);
new_where_clause VARCHAR2(10000);

-- gets the foreign key constraints on other tables which depend on columns in
```

```
parent_table --
CURSOR cons_cursor IS
SELECT owner, constraint_name, r_constraint_name, table_name, delete_rule
FROM all_constraints
WHERE constraint_type = 'R'
AND delete_rule = 'NO ACTION'
AND r_constraint_name IN (SELECT constraint_name
FROM all_constraints
WHERE constraint_type IN ('P', 'U')
AND table_name = parent_table
AND owner = table_owner)
AND NOT table_name = parent_table; -- ignore self-referencing constraints


-- for the current constraint, gets the child columns and corresponding parent
columns
CURSOR columns_cursor IS
SELECT cc1.column_name AS child_col, cc2.column_name AS parent_col
FROM all_cons_columns cc1, all_cons_columns cc2
WHERE cc1.constraint_name = child_cons
AND cc1.table_name = child_table
AND cc2.constraint_name = parent_cons
AND cc1.position = cc2.position
ORDER BY cc1.position;
BEGIN
-- loops through all the constraints that refer to parent table
FOR cons IN cons_cursor LOOP
child_cons   := cons.constraint_name;
parent_cons  := cons.r_constraint_name;
child_table  := cons.table_name;
child_cols   := '';
parent_cols  := '';

-- loops through the child/parent column pairs; builds column list of del
statements
FOR cols IN columns_cursor LOOP
IF child_cols IS NULL THEN
child_cols  := cols.child_col;
ELSE
child_cols  := child_cols || ', ' || cols.child_col;
END IF;

IF parent_cols IS NULL THEN
```

```
parent_cols  := cols.parent_col;
ELSE
parent_cols  := parent_cols || ', ' || cols.parent_col;
END IF;
END LOOP;

/* construct the WHERE clause of the delete statement, including a subquery to
get the related parent rows */
new_where_clause  :=
'where (' || child_cols || ') in (select ' || parent_cols || ' from ' ||
table_owner || '.' || parent_table ||
' ' || where_clause || ')';

delete_cascade(cons.owner, child_table, new_where_clause);
END LOOP;

--makes delete statement for the current table
delete_command  := 'delete from ' || table_owner || '.' || parent_table || ' '
|| where_clause;

-- print delete command
DBMS_OUTPUT.put_line(delete_command || ';');

EXECUTE IMMEDIATE delete_command;

-- issue commit here
END;
```

Example 2 Input ECA Rule: An ECA rule that specifies that all clauses must be executed based on conditions. Insertion of deletion must take place to restore invariants.

```
ECA { ecaTriggr = On Ins rel_orderedAt_Order_Vendor,
ecaDelta  = vio_Delta_Order_Vendor,
ecaAction = ALL [ Do Ins rel_I_Order (EDif
(EIsc (ECps (EDcD vio_Delta_Order_Vendor,EFlp(EDcD
vio_Delta_Order_Vendor)),
EDcI cpt_Order),EDcI cpt_Order
))[], Do Ins rel_I_Vendor (EDif (EIsc (ECps (EFlp
(EDcD
vio_Delta_Order_Vendor),EDcD
vio_Delta_Order_Vendor),EDcI cpt_Vendor),EDcI
cpt_Vendor))[]][], ecaNum    = 3}
```

Brief Explanation of Example 2:

T2 EFA White box test

Test type	White Box/Non-functional, using static analysis
Schedule	January 2016
Requirements	Non-functional using static analysis

Input

N/A

Output

The desired output depends on the specific targeted non-functional specification

Procedure

The test shall be performed by an third party (Dr.Kahl), where Dr. Kahl shall perform a code walk through where he will be able to confirmed the validity of the code that is written, the correctness and the maintainability of the code. Structural testing uncovers errors during implementation of the program, and focuses on how the process occurs, and evaluates structure of the program. The white box test is also implemented by teach individual of our team and shall focus on abnormal or extreme behaviour. The white box test focuses on static program analysis which can be performed without actually executing the program. The main non-functional requirement for EFA is code maintainability and correctness, which requires heavy documentation alongside the source code. ¿¿¿¿¿¿¿ 6faa150b68ea6f514aa3f0761a6a27ef9a69ca7f

Bibliography

[hac] QuickCheck package automatic testing of haskell programs. `https://hackage.haskell.org/package/QuickCheck`. Accessed: 2015-10-29.

[Joo07] Stef Joosten. AMMBR: A Method to Maintain Business Rules. 2007.