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	03 / 11 / 2015	Version 0
Yuriy Toporovskyy	27 / 10 / 2015	Reorganized document
Yuriy Toporovskyy	27 / 10 / 2015	Initial version - template
Yash Sapra	31/10/2015	Testing tools
Jaeden Guo	31/10/2015	System Test Descriptions

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 is centered around our contribution to Ampersand and ignores other elements of the Ampersand process such as the artifacts that are generated.

1.2 Objectives

Preparation for testing

The primary objective of this test plan is to gather all relevant information that could aid in creating effective tests for EFA. EFA in this regard is modularized and tested as an individual component before it is tested within the Ampersand system.

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 based on the constraints and requirements presented in the Software Requirements Specification (i.e. SRS). [Just say "Software Requirements

Specification (SRS)" —DS] This document focuses on how the functional and non-functional requirements provided in the SRS will be tested for this project.

Environment

This test plan 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 the Ampersand system.

1.3 Definitions

SRS

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

EBNF

Extended BackusNaur Form. A notation for specifying the syntax of languages, see [**?**
].

RA

Relation algebra. The mathematical language used in ADL files to specify business rules.

PA

Process algebra. The mathematical language used by ECA rules to describe the action to be taken to fix violations. A "PA clause" (also written as "PAclause"), or process algebra clause, is an imperative-style language which represents the *mathematical* process which Ampersand uses. The syntax of PA clauses, in EBNF notation, is as follows:

```
PAclause ::= 'One' '(' PAclause { ',' PAclause } ')' ;
           | 'Choice' '(' GPAclause { ',' GPAclause } ')' ;
           | 'All' '(' PAclause { ',' PAclause } ')' ;
           | ('Ins' | 'Del') '(' RExpr ',' RAtom ')' ;
           | 'Nop'
           | 'Blk'
GPAclause ::= RExpr '->' PAclause ;
```

where "RExpr" represents RA expressions, and "RAtom" (RA atom) represents *atomic* RA expressions (i.e. terms with no operators).

Table 1.1: Semantics of PAclause terminals

One($p_0 \dots p_n$)	Execute exactly one of $p_0 \dots p_n$.
Choice($g_0 \rightarrow p_0 \dots g_n \rightarrow p_n$)	Execute exactly one of p_i, such that g_i is a non-empty RA term.
All($p_0 \dots p_n$)	Execute all of $p_0 \dots p_n$.
<Ins/Del> (e, r)	Insert or delete the expression e from the relation r.
Nop	Do nothing.
Blk	The null command, which blocks forever.

The semantics of process algebra says that the "choice" operators (e.g. One and Choice) may execute any one of their subclauses; if *any* of the subclauses can be completed, the PA clause has restored the violation. One choice may be considered better in some ways, for example, different alternatives could have vastly different execution costs. For the purpose of this document, however, we will make the simplest "choice" possible, which generally means an arbitrary choice.

ECA Rule

Event-Condition-Action Rule. A rule which describes how to handle a constraint violation in a database. The syntax of ECA rules is as follows:

```
ECArule ::= 'On' ('Ins' | 'Del')
            '(' RExpr ',' RAtom ')'
            'Do' PAclause
```

HUnit

A Haskell library for unit testing. See section 3.2.

QuickCheck

A Haskell library for running automated, randomized tests. See section 3.2.

Sentinel

A test server accessible through the Ampersand webcite [“website” —DS] ([**?**]) which executes a set of randomly generated tests on Ampersand on a daily basis, see [**?**] for details.

Chapter 2

Plan

2.1 Software Description

Ampersand is a software tool that converts the formal specifications of business entities and rules, compiles it into various design artifacts (e.g. latex documents), and produces a prototype web application.

The business prototype implements business logic according to specifications provided by the user, it uses the entities and relationships that the user provides to form a relational database with a simple web application front-end.

ECA for Ampersand focuses on automatically restoring a particular class of database violations using an algorithm called AMMBR [**?**]. This class of violations is realized within Ampersand as ECA rules – our contribution to Ampersand will add support for ECA rules, that will affect Ampersand's back-end as well as the generated prototype it provides.

2.2 Test Team and Test Schedule

2.2.1 Test team

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

- Yuriy Toporovskyy
- Yash Sapra
- Jaeden Guo

2.2.2 Test Schedule

Dates	Tests to be performed
Before Revision 0 demonstration	Unit testing, black box testing, white box testing with heavy use of QuickCheck
Before Revision 0 demonstration	System testing EFA with Ampersand
Before Final Demonstration	Testing for abnormality and performance.

—

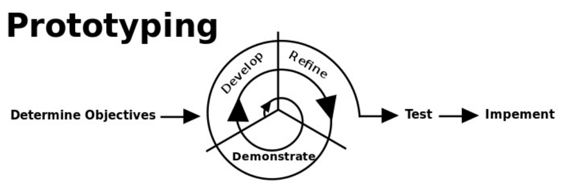
Note: More details for specific test will be provided in the future.

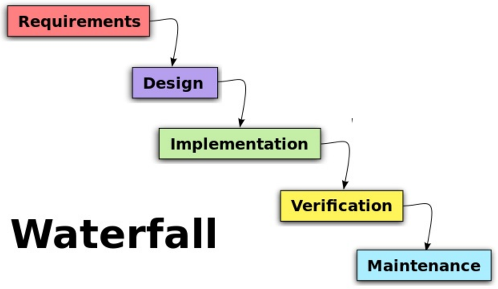
Chapter 3

Methods and constraints

3.1 Methodology

A Waterfall Development methodology is used in combination with Software Prototyping and will likely result in something similar to the spiral model. The Waterfall method help translate the requirements outlines in the SRS into functional and non-functional requirements that the design must include. Furthermore, as Software Prototyping is not a standalone method, but rather an approach to development activities these two compliment each other well.





3.2 Test tools

Various tools are used by the existing Ampersand developers to test Ampersand. In order to integrate best with Ampersand, our code will adopt their 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 RA 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. QuickCheck will be used for individualized module testing and well as [“as well as” —DS] provide a fair array of random tests for the combination of all modules [**?**].

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 results. 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

Requirements are the main motivator for tests and test methodoloy – testing should help ensure that requirements are met. To this end, our requirements are briefly summarized and labelled below.

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 ECArule format.

F3 produce an output compatible with the existing pipeline.

F4 annotated [“annotate” —DS] generated code with proofs of correctness or derivations, where appropriate.

F5 automatically fix database violations in the mock database of the prototype.

F6 not introduce appreciable performance degradation.

F7 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 be a pure function; it should not have side effects.

3.4 Data recording

Our test data is stored in the following format

- QuickCheck tests - Stored in source code.
- ADL files - ADL files which act as an input to the Ampersand system will be stored on github under the "Ampersand-models" repository.
- ASQL - Annotated SQL generated for the EFA project will be stored in the source code as comments.

3.5 Constraints

- It is difficult to obtain ADL (input) files that will generate a broad range of rules for our test.
- Testing with QuickCheck involves careful crafting of invariants (rules).
- ECA rule pipeline depends on the things above it. Any bug found will have to be traced back and communicated to the developers.

Chapter 4

System Test Descriptions

Many test cases use domain specific language to indicate inputs and outputs, for both clarity and brevity. This includes the syntax and semantics of ECA rules and Abstract SQL. For the full syntax and semantics of these, as well as related definitions, see section 1.3.

T1 ECA rule executing "All" subclauses

Test type	Dynamic, black box, automated
Schedule	Term 2
Requirements	F1

Input

The input is an ECA rule of the form:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{On Ins}(\Delta, r_0)\ \ \textbf{Do All}(\textbf{Ins}(e_1, r_1), \textbf{Ins}(e_2, r_2))} \\ \text{where} & r_0, r_1, r_2 \ := \text{Atomic Relation} \\ & e_1, e_2, \Delta \ := \text{Expression} \end{array}$$

Output

The output is an abstract SQL function of the form:

```
f (delta, r_0):
  INSERT INTO <r_1_table> VALUES <e_1_query>;
  INSERT INTO <r_2_table> VALUES <e_2_query>;
```

Description

ECA rules of the input format are generated using QuickCheck, converted to abstract SQL, then compared against the expected output format using HUnit.

T2 ECA rule executing “Choice” subclauses

Test type	Dynamic, black box, automated
Schedule	Term 2
Requirements	F1

Input

The input is an ECA rule of the form:

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{On Ins}(\Delta, r_0) \;\; \textbf{Do Choice}(p_0, p_1)} \\ \text{where} & r_0 & := \text{Atomic Relation} \\ & \Delta & := \text{Expression} \\ & p_0, p_1 & := \text{PA Clause} \end{array}$$

Output

The output is an abstract SQL function of the form:

```
f (delta, r_0):
  <p_i_statement>
```

where $i \in \{0, 1\}$.

Description

ECA rules of the input format are generated using QuickCheck, converted to abstract SQL, then compared against the expected output format using HUnit.

T3 ECA rule with empty PA clause

Test type	Dynamic, black box, automated
Schedule	Term 2
Requirements	F1

Input

The input is an ECA rule of the form:

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{On } \{\textbf{Ins/Del}\}(\Delta, r_0) \ \ \textbf{Do Nop}} \\ \text{where} & C_0, C_1 & := \text{Concept} \\ & r_0 & := \text{Atomic Relation} \end{array}$$

Output

The output is the empty abstract SQL statement; that is, a statement of the form:

```
f (delta, r_0): {} \\ Do nothing
```

Input

ECA rules of the input format are generated using QuickCheck, converted to abstract SQL, then compared against the expected output format using HUnit.

T4 ECA rule inserting into Identity relation

Test type	Dynamic, black box, automated
Schedule	Term 2
Requirements	F1

Input

The input is an ECA rule of the form:

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{On } \{\textbf{Ins/Del}\}(\Delta, r_0) \ \ \textbf{Do } \{\textbf{Del/Ins}\}(e_0, \mathbb{I}_{C_0})} \\ \text{where} & C_0 & := \text{Concept} \\ & r_0 & := \text{Atomic Relation} \\ & e_0 & := \text{Expression} \end{array}$$

Output

The output is an abstract SQL statement of the form:

```
f (delta, r_0):
  {INSERT INTO/DELETE FROM} <C_0_Population> VALUES <e_0_query>;
```

where `C_0_Population` is the table corresponding to $\mathbb{I}_{C_0}$.

Description

ECA rules of the input format are generated using QuickCheck, converted to abstract SQL, then compared against the expected output format using HUnit.

T5 ASQL is valid

Test type	Dynamic, black box, manual
Schedule	Term 2
Requirements	F3, F1

Input

The input is an ADL file which contains various entities, among them business process rules which produce ECA rules.

Output

The prototype generated by Ampersand, which should be syntactically and semantically valid, as determined by the software which runs the prototype.

Description

Ampersand generates a prototype from the input ADL file, which a developer will open on a web server running on their machine. If the file compiles (or more likely, is interpreted) successfully, this is an indication that the generated ASQL inside the prototype is correct, in the context of the entire prototype.

T6 EFA system compatibility

Test type	Functional, black box, manual
Schedule	Dec 2015
Requirements	F3

Input

An ADL file containing business process rules which gives rise to ECA rules. The ADL file comes from the Ampersand test database.

Output

The successful compilation of the input script.

Description

The same ADL script is compiled with two different version of Ampersand. The first is without ECA rules the second is with ECA rules that this project adds. Both of these scripts should pass through the Ampersand generator without causing errors or violations. If the second script which contains ECA rules successfully passes through the Ampersand compiler, then the new code additions due to EFA are compatible with the Ampersand system.

T7 EFA is a pure function

Test type	Static, white box, manual
Schedule	Proof of Concept demonstration
Requirements	F4

Description

Two conditions must hold for a function to be considered a pure function. Firstly, The function always evaluates to the same result given the same argument values, and secondly, the evaluation of the result does not cause any semantically observable side effects (e.g. mutation). Since all functions in Haskell are pure, and our code must be implemented in Haskell, this is guaranteed if the Haskell type checker accepts our program. The only caveat is that “unsafe” functions (i.e. functions which explicitly violate the semantics of Haskell) must not be used, or this guarantee may be broken. [What are considered “unsafe” functions? Do you have a reference or rule of thumb? —DS]

T8 EFA gives appropriate feedback

Test type	Functional
Schedule	Proof of Concept demonstration
Requirements	F6,N1

Input

The input shall be ECA rules specifying invariants that must be maintained throughout the program. These ECA rules should be those derived from a user-declared rule, and the original rule should have appropriate documentation. For example, for a system composed of actors, plays, and acting rules, there could be a rule like:

```
1   RULE "who's cast in roles"
2     : cast;instantiates |- qualifies;comprises~
3   MEANING "an Actor may appear in a Performance of
4             the Play only if the Actor is skilled
5             for a Role that the Play comprises"
```

These two declarations (which are actual ADL syntax) declare a rule called "who's cast in roles" (line 1), gives the definition of that rule in RA on line 2(`cast`, etc, are previously declared relations), and gives an english language explanation of the rule's meaning on lines 3-5.

Output

When the prototype is run, and a violation occurs, any feedback regarding the ECA rule which fixes that violation should include the original "MEANING" annotation which corresponds to the business rule from whic the ECA rule originated. The actual ECA rule is omitted here:

```
======= Violation log entry <...>
=== ECA rule fired: <...>
=== Delta: <...>
=== Original rule: cast;instantiates |- qualifies;comprises~
Violation occured because rule "who's cast in roles" was not
 satisfied. This is because "an Actor may appear in a
 Performance of the Play only if the Actor is skilled for a
 Role that the Play comprises"
```

T9 EFA code walk-through

Test type	Static, white box, manual
Schedule	January 2016
Requirements	N2

Brief Explanation

Input and output are not available for this test, as it requires each member of the design team to walk through the code line by line to check if it is easy to understand by another programmer and well documented. If it is easy to read and understand but not only the individual who wrote it but those around them, then it should be easy to maintain. Part of this standard will be to conform to the style of existing Ampersand code.

[In this case you could say that the source code is an "input" for the test. There is no output as it is a measure of readability/maintainability. —DS]

T10 Degradation Test

Test type	Dynamic, black box, partially automated
Schedule	First Week of February 2016
Requirements	F6

Input

The Ampersand test suite.

Output

The time taken to run the test suite, both with EFA enabled and EFA disabled.

Description

The amount of time Ampersand takes to compile a prototype will measure performance degradation; if Ampersand takes substantially longer to compile after the addition of EFA then it is an appreciable difference. The comparison will be made between `time ampersand-test-suite` and `ampersand-test-suite --noEFA` for enabled and disabled, respectively. These are expected to differ by no more than 5%.

[Is "time" a keyword? If so should it be on both commands? —DS]

T11 EFA generates annotated code

Test type	Dynamic, white box, manual
Schedule	January 2016
Requirements	F7, N1

Input

An ADL file containing business process rules which gives rise to ECA rules.

Output

The generated prototype, containing annotated code.

Description

The engineer will compile the input ADL file with ampersand, passing the appropriate command line options [Do you know what these options are or have they not been designed yet? —DS] to produce AQSL annotated with the proof of their derivation, and to produce code which logs to the console whenever an ECA rule is used to fix a violation.

The console log should include all information relevant to the ECA rule and violation which fired. [This sentence sounds a bit awkward. —DS] Parts of this information may be internal information, but it is mainly composed of the proof of derivation of that ECA rule, and the origin of the ECA (which business rule it came from). The annotation indicating these things must be clear to the typical user of Ampersand.

T12 EFA domain

Test type	Static, white box, manual
Schedule	Term 2
Requirements	F2, F3

Description

Our contribution will take as an input the internal representation of the entire ADL file, which is essentially a large record, called `FSpec`. For the most part, the majority of components will not be used – we will mainly focus on one or two components of dozens. However, potentially the entire structure may be useful, so we say that our domain is this structure.

Therefore, the entry point to our code should be a function with type `FSpec -> a -> ASQL` for some `a` (representing optional or auxillary arguments), where `ASQL` is the type representing abstract SQL. If this code type checks, then the compiler has verified that our domain of input is indeed exactly `FSpec`.

T13 Sentinel test

Test type	Dynamic, black box, automated, integration test
Schedule	Term 2
Requirements	N3, F1

Description

The Ampersand system includes an online automated test framework, called Sentinel (see 1.3), which runs at regular intervals and emails developers about failing tests. Sentinel uses the same environment used by the developers of Ampersand. Our code should compile and our test cases should all succeed when run by Sentinel. Additionally, there should be no more failures among the existing test cases of Ampersand (those not written by us) when run with our contibution as compared to when run without our contributon.

[You make no mention of any tests related to requirement F5. Can you test for exceptions? —DS]