Event control action rules for Ampersand

Yuriy Toporovskyy (toporoy)
Yash Sapra (sapray)
Jaeden Guo (guoy34)

Supervised by: Dr. Wolfram Kahl

Department of Computing and Software
McMaster University
Ontario, Canada

April 18, 2016

Abstract

Ampersand Tarski is a tool used to produce functional software documents based on business process requirements. At times, logical discrepancies arise when system changes occur which violate the restrictions set forth by the user. When a system violation occurs, one of two things can happen: the change that is meant to take place is adjusted so it no longer violates the restrictions or the changes are discarded. The purpose of Event condition action rules for Ampersand (EFA) was to replace the exec-engine that is currently used to deal with violations; unlike the exec-engine, EFA is automated and provides proof of correctness embedded in the code, it able to type SQL statements and assure no "dead-ends" occur when queries are executed.

Contents

1 Introduction

This document is a meant as a guide for EFA that includes the motivations taken from a business perspective, the mathematical and software foundations that resulted in the logical flow of EFA's design, and the testing that took place to assure EFA's functionality and correctness.

Currently, Ampersand is readily accessible to the public through Github and it is equipped with the ability to assess logical discrepancies on sets of data based on user-specified restrictions. Logical discrepancies arise when system changes occur which violate the restrictions set forth by the user. When a system violation occurs, one of two things can happen: the change that is meant to take place is adjusted so it no longer violates the restrictions or the changes are discarded. Ampersand is used to manipulate data and generate prototypes, although there is a debugger, certain errors still slip through. When the system rules are changed by the user, all data which are inconsistent with the new system must be eliminated or rehabilitated so it can be returned back into the system. Data inconsistencies are persistent bugs that can distort the product that Ampersand seeks to provide.

These data inconsistencies are corrected through ECA rules which use process algebra (PA) to correct or discard data using violations. EFA is used to translate these ECA rules, execute SQL queries to correct violations and safeguards the database from illegal transactions.

Ampersand

Objectives

Document Guide

2 The Purpose of the Project

A large part of designing software systems is requirements engineering. One of the greatest challenges of requirements engineering is translating from business requirements to a functional specification. Business requirements are informal, with the intention of being easily understood by humans; however, functional specifications are written in formal language to unambiguously capture attributes of the information system. Typically, this translation of business requirements to a formal specification is done by a requirements engineer, which can be prone to human error.

Ampersand is a tool which aims to address this problem in a different way; by translating business requirements written in natural language into a formal specification by means of a "compilation process" (Joo07). Even though the business requirements and formal specification are written in entirely different languages, the "compiler guarantees compliance between the two" (Joo07, 2).

Ampersand also provides engineers with a variety of aids which helps them to design products that fulfill all of the needs of their clients and the end-users; including data models, service catalogs and their specifications. Ampersand has proven reliable in practical situations, and there have been efforts to teach this approach to business analysts.

A large portion of the Ampersand system is already in place; the primary focus of this project was to augment Ampersand with increased capabilities for automation. The module "Automatically Fix Violations" in Figure 1 represents the EFA project and where it fits in the current version of Ampersand.

Ampersand, before the EFA project included an algorithm to generate the post-conditions (i.e. formal specifications) of each function. It also contained the ECA rules that are generated by the Ampersand compiler. However, these ECA rules were not being called in order to automatically restore violations. With the EFA extension for Ampersand, it allows us to automatically rehabilitate existing system data while maintaining the information system according to user specifications. It also allows us to automatically restore system invariants and create a program that allows Ampersand to make the most efficient choice regarding how it wishes proceed based on each individual case.

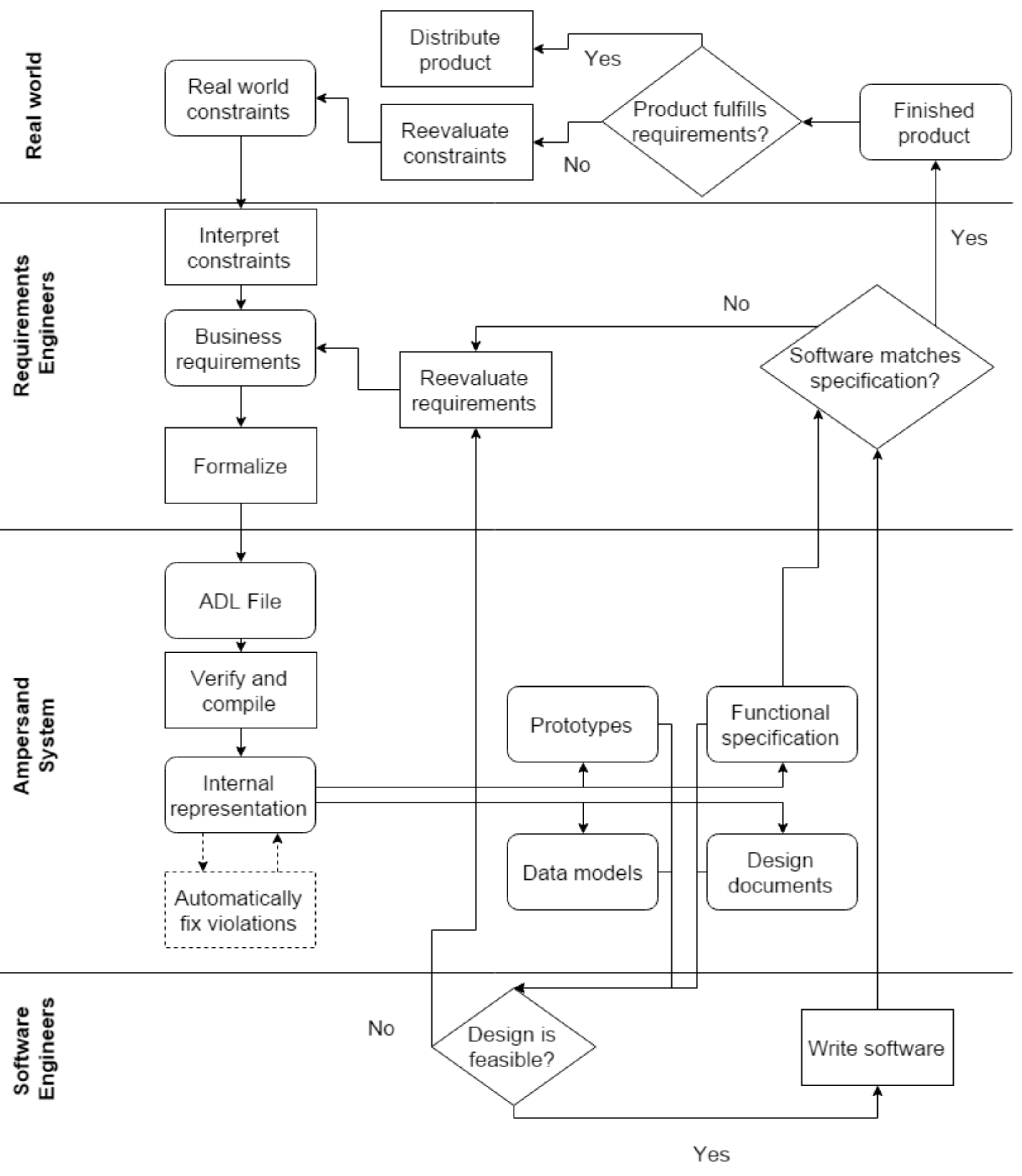


Figure 1: Business process diagram representing EFA project represented as a dashed box

3 System Architecture and Module Hierarchy

3.1 System Architecture

This section provides an overview of system architecture and module hierarchy. The initial section introduces term and tools used in the making of each EFA module. The module design is detailed with UML-like class diagrams. However, UML class diagrams are typically used to describe the module systems of object-oriented programs, as opposed to functional programs. Many of the components of the traditional UML class diagram are inapplicable to functional programs; therefore, we detail our modifications to the UML class diagram syntax in section

Furthermore, the syntax used to describe types and data declarations is not actual Haskell syntax. The syntax shares many similarities, but several changes to the syntax are made in this document in order to present the module hierarchy in a clear manner. These changes are also detailed, in section

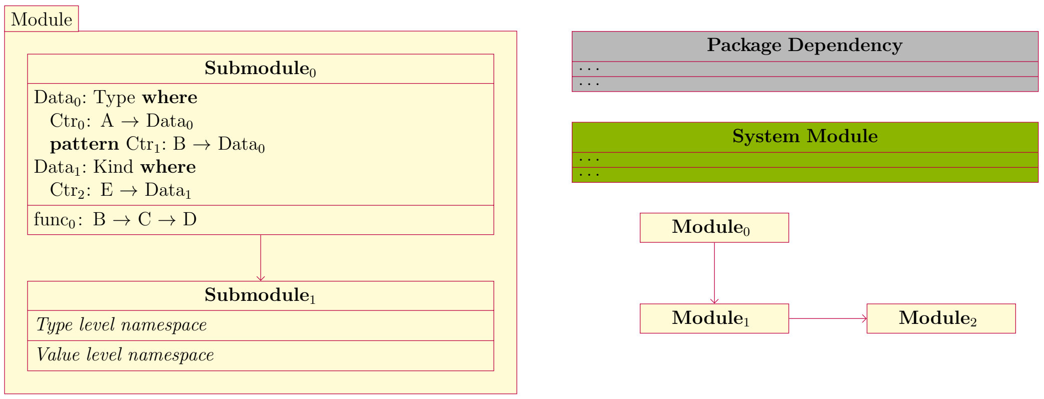


Figure 2: Example of module diagram syntax

3.2 A Description of Haskell-Similar Syntax

This section details the syntax used to describe the module system of Ampersand. This syntax largely borrows from actual Haskell syntax, and from the Agda programming languageAgda is a dependently typed functional language, and since a large part of our work deals with "faking" dependent types, the syntax of Agda is conducive to easy communication of our module system. The principle of faking dependent types in Haskell is detailed in Hasochism (**?**) (a portmanteau of Haskell and masochism, because purportedly wanting to fake dependent types in Haskell is masochism). While the implementation has since been refined many times over, the general approach is still the same, and will not be detailed here. While the changes made to the Haskell syntax are reasonably complex, the ensuing module description becomes vastly simplified. This section is meant to be used as a reference - in many cases, the meaning of a type is self-evident.

3.3 Description of Types and Kinds

In the way that a type classifies a set of values, a kind classifies a set of types. Haskell permits one to define algebraic data types, which are then "promoted" to the kind level (**?**). This permits the type constructor of the datatype to be used as a kind constructor, and for the value constructors to be used as type constructors. In every case in our system, when we define a datatype and use the promoted version, we never use the *unpromoted* version. That is, we define types which are never used as types, only as kinds, and constructors which are never used as value constructors,only type constructors. We write $\mathrm{X} : \mathrm{A} \to \mathrm{B} \to \ldots \to \mathrm{Type}$ to denote a regular data type, and $\mathrm{Y} : \mathrm{A} \to \mathrm{B} \to \ldots \to \mathrm{Kind}$ to denote a datatype which is used exclusively as a kind.

3.4 Description of Dependant types

The syntax used to denote a "fake" dependent type in our model is the same as used to denote a real dependent type in Agda. $(\mathrm{x} : \mathrm{A}) \to \mathrm{B}$ is the function from x to some value of type B, where B can mention x. This nearly looks like a real Haskell type - in Haskell, the syntax would be `forall (x :: A) . B`. However, the semantics of these two types are vastly different - the former can pattern match on the value of x, while the latter cannot.

In certain cases, it may be elucidating to see the *real* Haskell type of an entity (function, datatype, etc.). To differentiate the two, they are typeset differently, as in this example.

The real type of a function whose type is given as $(\mathrm{x} : \mathrm{A}) \to \mathrm{B}$ in our model is `forall (x :: A) . SingT x -> B`. `SingT :: A -> Type` denotes the singleton type for the kind A, which is inhabitted by precisely one value for each type which inhabits A. The role and use of singleton types is detailed further on, in section **??**.

The syntax $\forall\ (\mathrm{x} : \mathrm{A}) \to \mathrm{B}$ is used to denote the regular Haskell type `forall (x :: A) . B`. As is customary in Haskell, the quantification may be dropped when the kind A is clear from the context: $\forall\ (\mathrm{x} : \mathrm{A}) \to \mathrm{P\ x}$ and $\forall\ \mathrm{x} \to \mathrm{P\ x}$ denote the type `forall (x :: A) . P x`.

3.5 Constraints

The Haskell syntax `A -> B` denotes a function from A to B. However, we use the arrow to additionally denote constraints. For example, the function `Show a => a -> String` would be written simply as Show a $\rightarrow$ a $\rightarrow$ String. In certain casees, a constraint is intended to be used only in an implicit fashion (i.e. as an actual constraint), in which case the constraint is written with the typical $\Rightarrow$ syntax.

3.5.1 Existential quantification

The type $\exists$ (x : A) (P x) indicates that there exists some x of kind A which satisfies the predicate P. Unfortunately, Haskell does not have first class existential quantification. It must be encoded in one of two ways:

3.5.2 Types, kinds, and type synonyms

Type synonyms are written in the model as Ty : K = X, where Ty is the name of the type synonym, K is its kind, and X its implementation. This is to differentiate from type families, which are written as Ty : K **where** Ty ... = ...

3.5.3 Overloading

Haskell supports overloaded function names through type classes. When we use a type class to simply overload a function name, we simply write the function name multiple times with different types. The motivation for this is that often the real type will be exceeding complex, because it must be so to get good type inference.

3.5.4 Omitted implementations

When the implementation of a type synonym, or any other entity, is omitted, it is replaced by "...". This is to differentiate from a declaration of the form Ty : Type, which is an abstract type whose constructors cannot be accessed. Furthermore, types may have pattern-match-only constructors; that is, constructors which can only be used in the context of a pattern match, and not to construct a value of that type. This is denoted by the syntax "**pattern** Ctr : Ty". Furthermore, it is not a simple matter of convention - the use of this constructor in expressions will be strictly forbidden by Haskell.

- With a function (by DeMorgan's law):
  `(forall (x :: A) . P x -> r) -> r`

- With a datatype:
  `data Exists p where Exists :: p x -> Exists p`

Choice of form is based on the circumstances in which the function will most likely be used, since whether one form is more convenient than the other depends largely on the intended use. However, these two forms are completely interchangeable (albeit with some syntactic noise) so the syntax presented here does not distinguish between the two.

4 Module Hierarchy

4.1 A Description of Module Diagram Syntax

The module hierarchy is broken down into multiple levels to better describe the system. A coarse module hierarchy is given, and each module is further broken into submodules. A dependency between two modules A and B indicates that each submodule in A depends on all of B. There is no necessity to break down modules into submodule, if they do not have any interesting submodule structure. Arrows between modules and submodules denote a dependency.

External dependencies, which are modules which come from an external pacakge, are indicated in grey. System modules, which are modules part of Ampersand, but not written specifically for EFA (or, on which EFA depends, but few or no changes have been made from the original module before the existance of EFA), are indicated in green. The module heirarchy of these modules is not described here; they are included simply to indicate which symbols are imported from these modules. An example of the syntax is found in figure

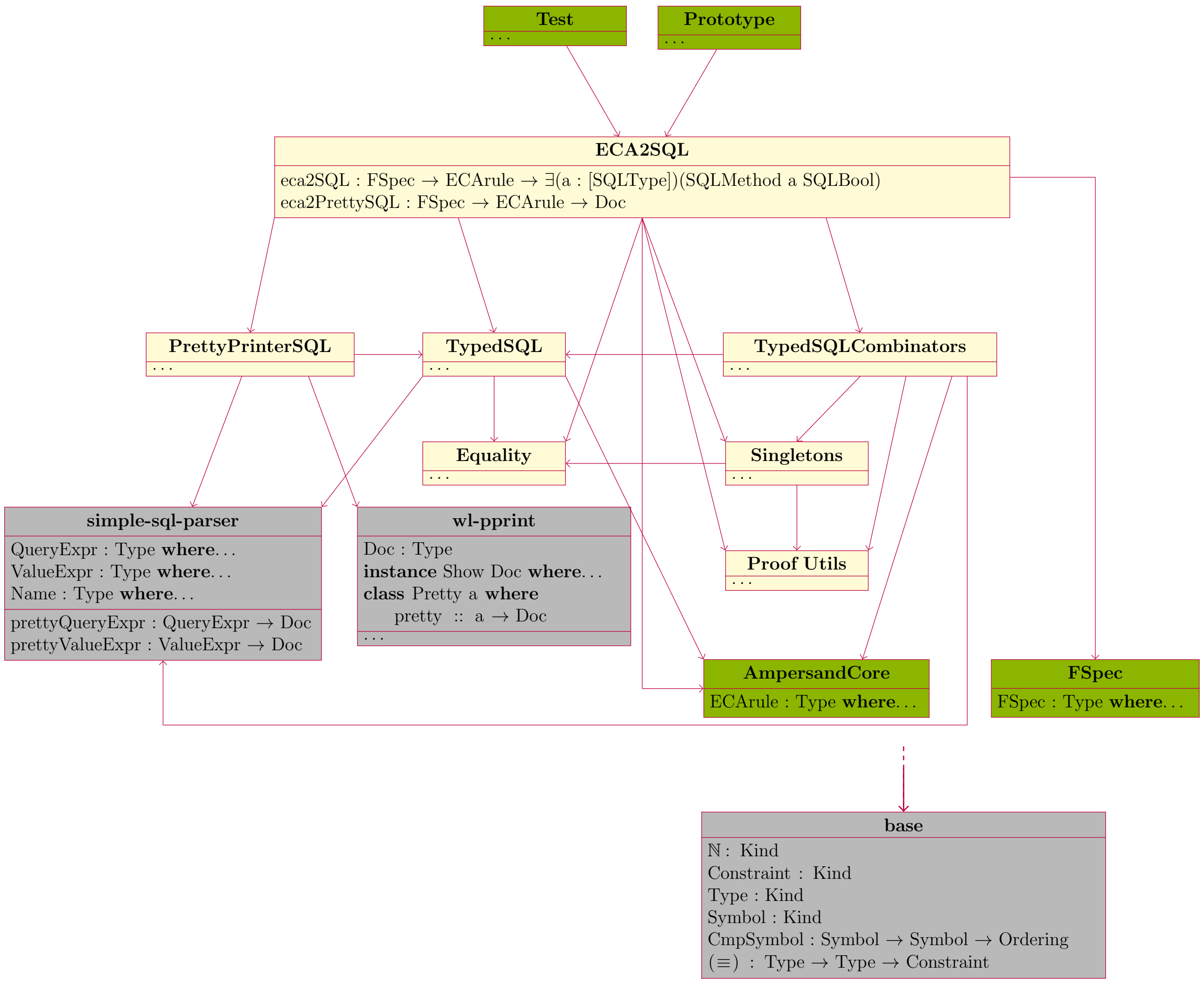
This section contains a hierarchal breakdown of each module, as well as a brief explanation of each modules' elements.

The module heiarchy of EFA as a whole is given in figure 3. Note that every module which is part of EFA depends on the Haskell `base` package (which is the core libraries of Haskell). Also note that for the `base` package, we only include primitive definitions (i.e. those not defined in real Haskell) which may be difficult to track down in the documentation. The kinds ℕ and Symbol correspond to type level natural number and string literals, respectively. The kind Constraint is the kind of class and equality constraints, for example, things like Show x and Int $\sim$ Bool. Note that `Show` itself does *not* have kind Constraint – its kind is Type $\rightarrow$ Constraint. The detailed semantics of these primitive entities can be found in the GHC user guide(**?**

). While many modern features of GHC are used in the actual implementation, they are not mentioned in, nor required to understand, the module description.

The primary interface to EFA is the function eca2PrettySQL, which takes an FSpec (the abstract syntax of Ampersand) and an ECA rule, and returns the pretty printed SQL code for that rule. Also note that while the dependencies within EFA modules is relatively complex, they depend on the rest of the Ampersand system in a simple manner. The modules Test and Prototype implement the testing framework and the prototype generation, respectively; these modules depend directly on only one module from EFA, namely ECA2SQL. Similarly, the majority of EFA itself does not depend directly on Ampersand modules outside of EFA. This makes EFA very resilient to changes in the core Ampersand system; in order to update EFA to work with a modification to Ampersand, only one EFA module – ECA2SQL – will generally need to be modified.

All functions named in the module hierarchy are total - they do not throw exceptions, or produce unhandled errors or infinite loops. Therefore, no additional information past the type of the function is required to deduce the inputs and ouputs of the function – they are precisely the inputs and outputs of the type.



Test
. . .
Prototype
. . .
ECA2SQL
eca2SQL : FSpec → ECArule → ∃(a : [SQLType])(SQLMethod a SQLBool)
eca2PrettySQL : FSpec → ECArule → Doc
PrettyPrinterSQL
. . .
TypedSQL
. . .
TypedSQLCombinators
. . .
Equality
. . .
Singletons
. . .
simple-sql-parser
QueryExpr : Type where. . .
ValueExpr : Type where. . .
Name : Type where. . .
prettyQueryExpr : QueryExpr → Doc
prettyValueExpr : ValueExpr → Doc
wl-pprint
Doc : Type
instance Show Doc where. . .
class Pretty a where
pretty :: a → Doc
. . .
Proof Utils
. . .
AmpersandCore
ECArule : Type where. . .
FSpec
FSpec : Type where. . .
base
ℕ : Kind
Constraint : Kind
Type : Kind
Symbol : Kind
CmpSymbol : Symbol → Symbol → Ordering
(≡) : Type → Type → Constraint

Figure 3: Module diagram for EFA as a whole

References

[Joo07] Stef Joosten. Deriving Functional Specifications from Business Requirements with Ampersand. *CiteSeer*, 2007.

[LM13] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. *SIGPLAN Not.*, 48(12):81–92, September 2013.