Bachelor thesis
Computer Science

Radboud University

The Evolvability of iTasks using Normalized Systems

Author:
Justin Mol
s4386094

First supervisor/assessor:
prof.dr.ir. M.J. (Rinus) Plasmeijer
rinus@cs.ru.nl

Second assessor:
dr. Peter Achten
P.Achten@cs.ru.nl

August 17, 2016

Abstract

The problem of evolvability of information systems is playing a bigger and bigger role in the world of IT. Projects have great overhead expenses when a change in a large system is necessary. In the struggle to define evolvability, we experience difficulties finding stable solutions. Normalized Systems theory is a novel approach that makes a promising step towards finding this solution. In this thesis, we follow its path and put the method in a functional programming context in order to analyze the evolvability of the iTask framework. We have found that the two domains share similarities and the iTask approach shows promising qualities.

Contents

1 Introduction **2**

2 Normalized Systems **4**
2.1 Lehman's Laws of Software Evolution 4
2.2 Systems Theoretic Stability 5
2.3 The Theorems 6
2.4 Realizing Normalized Systems 10

3 Introduction to iTask **12**
3.1 Tasks 13
3.2 Shared data 13
3.3 Generic Interaction 14
3.4 Task Composition 14

4 Applying Normalized Systems theory to Clean **15**
4.1 Primitives in Clean 15
4.2 Implications for the NS Theorems 17
4.3 NS Elements in Clean 21

5 The Evolvability of iTask **25**
5.1 Data Representation 25
5.2 Stateful Workflow 27
5.3 User Interaction and Communication 28
5.4 Summary 29

6 Related Work **31**

7 Conclusions & Future Work **32**

References **33**

A Appendix **36**

Chapter 1

Introduction

The problem of evolvability of information systems is playing a bigger and bigger role in the world of IT. Projects have great overhead expenses when a change in a large system is necessary. As environments change, so do business requirements and therefore also the desired systems realizing these requirements. Alike all information systems, this requires iTask systems to be as evolvable as possible. In the struggle to define evolvability, we experience difficulties finding stable solutions.

Normalized Systems (NS) theory is a novel approach that makes a promising step towards finding this solution. The theory originates in the University of Antwerp and takes a distinctive approach to develop agile and evolvable software. In this thesis we use the theory to analyze the iTask system methodology in an attempt to determine and increase its measure of evolvability. This resulted in the following *contributions*:

- We explain how Normalized Systems theory relates to functional programming by exploring NS terminology in a functional context. This resulted in a wide discussion that gives new insight in how to create evolvable software in functional languages (Section 4.1 and 4.2). Hereto, we also discuss the feasibility of the automatic generation of normalized systems in a functional language (Section 4.3).
- We analyze the evolvability of iTask applications and identify parts of the system that are (dis)advantageous to their evolvability (Chapter 5). We do so by discussing the iTask framework and its practice in detail and comparing it to Normalized Systems theory.
- A third contribution can be mentioned, as this thesis finds yet another way to apply the Normalized Systems theory in an already growing field.

The rest of this thesis is structured as follows. We first go into the theoretical background on Normalized Systems and iTask in Chapters 2 and 3. In Chapter 4, we discuss the theory of Normalized Systems in the context of the functional language *Clean*. Following that, we analyze the iTask methodology and contrast it to Normalized Systems theory in Chapter 5. Finally, we will go into the related work in Chapter 6 and draw our final conclusions in Chapter 7.

Chapter 2

Normalized Systems

Normalized Systems is a fairly new theory developed at the University of Antwerp. It uses the system theoretic notion of stability (Mannaert, Verelst, & Ven, 2008) and Manny Lehman's laws of Software Evolution (Lehman, Ramil, Wernick, Perry, & Turski, 1997) to offer a method on building information systems that are resistant to change. We first go into the background of Normalized Systems (sections 2.1 and 2.2) and then continue towards the theorems of Normalized Systems and how to realize these in practice (sections 2.3 and 2.4).

2.1 Lehman's Laws of Software Evolution

The study that Lehman did in 1969 has led to a research field in computer science that has received an increasing amount of attention: *software evolution*. Software evolvability can be described as the ability for software to deal with change (Oorts, Huysmans, et al., 2014). This means that when a software system is evolvable, it is *easy* to modify. In other words, adapting the system does not result in extra work other than the desired change; the overhead is small.

Lehman's goal was to formulate a scientific theory of software evolution. He analyzed data acquired during a study of the IBM programming process (Lehman, 1996). In the mid seventies he formulated the first three laws of software evolution. These laws have been revisited various times, over a period of 20 years, in order to deal with the rapidly changing development practices in the 80s and 90s (Herraiz, Rodriguez, Robles, & Gonzalez-Barahona, 2013). The last change was made in 1996 and the following can be considered the current laws:

Lehman's Laws of Software Evolution

I	*Continuing Change*
1974	An E-type system must be continually adapted, or else it becomes progressively less satisfactory in use.
II	*Increasing Complexity*
1974	As an E-type system is changed, its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
III	*Self Regulation*
1974	Global E-type system evolution is feedback regulated.
IV	*Conservation of Organizational Stability*
1980	The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.
V	*Conservation of Familiarity*
1980	In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity.
VI	*Continuing Growth*
1980	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
VII	*Declining Quality*
1996	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.
VIII	*Feedback System*
1996	E-type evolution processes are multilevel, multiloop, multiagent feedback systems.

The laws all relate to *E-type systems*. Lehman describes E-type systems as software systems that solve a problem or implement a computer application in the *real world* (Lehman, 1996).

2.2 Systems Theoretic Stability

In the field of Systems Theory, *stability* is one of the most fundamental properties of a system. The notion of stability is related to the input and output of a function: "a bounded input function results in bounded output values, even as $t \to \infty$" (Mannaert et al., 2008). In the context of software evolvability, we get that a bounded amount of changes to a software system leads to a bounded amount of impacts to the system.

In Software Engineering, the structure of an information system is commonly seen as a composition of various *design patterns*. This is a widely studied subfield of Software Engineering. Design patterns are often described as reusable solutions to commonly occuring problems. In Normalized Systems theory, design patterns are analyzed from the perspective of change. Mannaert et al. (2008) argue that design patterns need to be stable with respect to some anticipated changes in order to deliver evolvable software systems. Here, design patterns refer to aggregations of entities that document good design practices and represent a way to document software designs. Their study on stability and software design has led to a number of theorems described in the next section.

Information systems that exhibit stability are referred to as *normalized systems* (Mannaert, Verelst, & Ven, 2012, p. 94). Normalized systems are stable with respect to some pre-defined set of anticipated changes. Mannaert et al. defined the following changes as a lower bound of evolvability:

- An additional data attribute or field.
- An additional data entity.
- An additional action entity, which may imply:
  - an an additional action entity having a specific data entity as input, or producing a specific data entity as output and
  - an additional action entity calling a specific action entity.
- An additional version of a task, which may imply
  - an additional version using another external technology,
  - an additional version representing a mandatory upgrade, and
  - an additional version containing an additional error state.

2.3 The Theorems

The need of information systems to be stable brings us to the requirement of avoiding any *combinatorial effects*. Mannaert et al. (2012, p. 93) define combinatorial effects in the following:

> We consider a combinatorial effect to be the consequence of dependencies between multiple modules in information systems that cause a change to a specific module to have an impact on other modules that are [...] unrelated to the original change.

In other words, combinatorial effects exist whenever one module x of an information system depends on a different module y in such a way that when a change is introduced in y, another unrelated change is also necessary in x.

Avoiding these combinatorial effects leads to an information system where changes remain constant over time, i.e. changes are independent of the size of the system. Achieving this can be difficult. Therefore, four design theorems were formulated that "act as constraints on the modular structure of information systems in order to achieve stability". In these theorems and the rest of this thesis, we will use the following definitions.

Firstly, the term *software entity* refers to the more fundamental concept of a module. They are instantiations of the constructs provided by a technology environment. All software entities of an application together form the modular structure of the software architecture. For example, in Java we have the `Class` construct. An instantiation of this construct is considered to be a software entity.

Next, the term *data entity* refers to: "a software entity that contains various attributes or fields, including links to other data entities". Data entities only contain data and have no interface. For example, the `study` variable in the below JavaScript code is one data entity, but so is `student`:

```
var study = "Bachelor Computer Science";
var student = {
  first_name: "Justin",
  middle_name: "",
  last_name: "Mol",
  study: study
};
```

Listing 2.1: A JavaScript example of a data entity

The term *action entity* refers to: "a software entity that represents an operation at a given modular level in a hierarchy". Action entities consume and/or produce data entities and have an interface. An action entity contains one or more *tasks*. A task, in the widest sense, is a set of instructions that performs a certain functionality. It can be a unit of processing that can change independently, or an invocation of another processing action (Mannaert, Verelst, & Ven, 2011).

It is up to the designer of the information system on which level the tasks are considered. When the modular structure is more detailed, action entities will be more fine-grained and any remaining details become submodular tasks. Moreover, Mannaert et al. write that the identification of these taks should be based on *change drivers* (or simply *concerns*).

Mannaert et al. (2012) distinguish between two types of tasks:

- Functional tasks: tasks that perform a specific functional operation in an information system
- Supporting tasks: generic tasks that perform a cross-cutting concern in an information system (e.g. persistency, logging, remote access)

The `module` object in the following JavaScript code is an example of an action entity with two functional tasks and one supporting task:

```
var students = [];
var module = {
  // A functional task:
  newStudent: function (first_name, last_name, study) {
    var student = {
      first_name: first_name,
      middle_name: "",
      last_name: last_name,
      study: study
    };
    students.push(student);
    return student;
  },
  // Another functional task:
  changeStudy: function (student, study) {
    students[students.indexOf(student)].study = study;
  },
  // A supporting task:
  logStudents: function () {
    for(var i = 0; i < students.length; i++)
      console.log(students[i]);
  }
};
```

Listing 2.2: A JavaSript example of an action entity

We will refer to data entities, action entities and tasks as *primitives*.

We summarize the Normalized Systems theorems in the following table:

1.	*Separation of concerns* An action entity can only contain a single task in normalized systems.
2.	*Data version transparency* Data entities that are received as input or produced as output by action entities, need to exhibit version transparency in normalized systems.
3.	*Action version transparency* Action entities need to exhibit version transparency in normalized systems.
4.	*Separation of states* The calling of an action entity by another action entity needs to exhibit state keeping in normalized systems.

Table 2.1: **Normalized Systems Theorems**

Theorem 1 shows that it is not possible to create an evolvable information system with action entities that combine two tasks. After all, when N action entities all contain the same task y and each a different (version of) task x_i, then changing task y would require a coding change to each of these N entities.

Theorem 2 states that adding a new field to a data entity should not affect any action elements that do not use the new field. This is to ensure that you can have multiple versions of data entities, that are used in or are the result of action entities, without changing other aspects of the system.

Since we have only one task per action entity and tasks can have multiple versions, action entities must be able to have multiple versions too. Thus, theorem 3 states that an action entity can have multiple versions without affecting the actions that call the action entity. In languages like Java, this is usually achieved using polymorphism or by using wrapper functions.

Theorem 4 implies the need for every action entity to keep state for every call to the action entity. The state contains information such as error or event information, making it closely related to asynchronous processing. Rather than having an error passed down synchronously, the error is stored in state and another action should react on it. Another practical manifestation is the need for a separate *stateful workflow*.

2.4 Realizing Normalized Systems

We continue by looking into how to design a normalized system. We realize that applying the theorems in practice can be very difficult for the average programmer. To this extend, Mannaert et al. (2012) introduce the concept of *elements* (Table 2.2). Elements are higher level modular structures that encapsulate software entities in such a way that they comply to the four theorems. They will be the building blocks to designing a stable information system.

As with all theories on Normalized Systems, elements are independent of a specific technology environment or programming language. The internal implementation of the elements that Normalized Systems theory provides is described as *design patterns*. As de Bruyn et al. (2012) point out, every NS element expands to around ten classes, giving a very fine-grained modular structure. Therefore, it is infeasible to create a normalized system by hand. Luckily, the work Mannaert et al. (2012) did has led to design patterns suitable to work with automatic code generation for these elements. This is a process that they call *pattern expansion*. For example, a data element is expanded using a basic name, context information (such as component and package name) and more detailed information about the data fields. A data element is then expanded to various classes, including a JavaBean class and classes to realize supporting tasks.

Earlier, we spoke of normalized systems being stable with respect to some anticipated changes. Mannaert et al. (2012) define the following anticipated changes for elements: an additional: data field, data element, action element (which may have a data element as in- or output), version of a task within an action element, action in a workflow element, workflow element, connector element or trigger element. They consider deletions a matter of garbage collection and modification a combination of addition(s) and deletion(s).

Table 2.2: **Normalized Systems Elements**

Element	Rationale
Data	• Represents an encapsulated data construct providing data version transparency. • Cross-cutting concerns (like persistency and access control) are considered to be a part of the data element.
Action	• Represents a single encapsulated functional task. • Arguments and parameters of action entities must be encapsulated data entities. • Workflows need to be separated from action entities and will be encapsulated in workflow elements. • Tasks need to be encapsulated in such a way that a separate action entity wraps the action entities representing task versions.
Workflow	• Contains the sequence of action elements that should be executed to fulfill a workflow. • Workflow elements cannot contain other functional tasks. • Workflow elements must be stateful. The state is required for every instance of use of the action element and must be part of, or linked to the instance of the data element that serves as argument.
Connector	• Connector elements must ensure that external systems can interact with data elements, but they cannot call an action element in a stateless way.
Trigger	• Trigger elements need to control the separated states and check whether an action element has to be triggered.

Chapter 3

Introduction to iTask

iTask is an embedded *Task-Oriented Programming* (TOP) language built on the functional programming language *Clean*. It aims to offer a way for programmers to create interactive, distributed, multi-user applications that are commonly manifested as web-services. These applications aid users in achieving some common goal. It is described as a *combinator language* (Achten, Koopman, & Plasmeijer, 2015). This means that it uses *combinators*, i.e. named programming patterns that in a very precise way state how a new piece of program is assembled from smaller pieces of programs.

TOP makes it possible to program complex multi-user applications by defining the tasks that need to be accomplished (Plasmeijer, Lijnse, Michels, Achten, & Koopman, 2012). iTask tries to take away the burden of developing and managing complex web applications over various environments and helps the programmer to focus on *what* the processors (human users and computers) have to do and what information structure is necessary to coordinate this in an efficient manner. It leaves the programmer wondering only what tasks are necessary to achieve the goal. iTask deals with the *how* of the story.

TOP applications use a client-server architecture with the client side implementing the front-end components of the application in various environments such as web browsers, smart phones and tablets. The server side implements the back-end of the application, taking care of any coordination and synchronization jobs done for the front-end components.

TOP makes use of four core concepts described and formalized by Plasmeijer et al. (2012). These four core concepts are implemented in the iTask framework and represent the components that build an iTask application.

3.1 Tasks

Firstly, **tasks** are abstract descriptions of interactive persistent units of work that have a typed value. Tasks may be interactive and can observe the current value of other tasks in three carefully controlled ways:

1. The task has no value observable for others
2. The task has an unstable value
3. The task has a stable value

Such a *task value* is of type `Value a`:

```
:: Value a   = NoVal    | Val a Stability
:: Stability = Unstable | Stable
```

Tasks can handle events. Events have a time stamp that is incremented using a counter. A *task result* can be a new task value, coupled with the time stamp of the event that created the value, or an *exception value*. A task can raise an exception value in case it is known that it can no longer produce a meaningful value.

Semantically, a task is a state transforming function that reacts to an event. It rewrites itself to a *reduct* and accumulates responses to users. A reduct contains the latest task result and a task that represents the remaining work. The `Responses` collect all responses of all subtasks the task is composed of. Three events are known: a `RefreshEvent` for refreshing the webpage, an `EditEvent` for editing values within a task and an `ActionEvent` for telling the step combinator what to do next. Similar to events, tasks are provided with a fresh identification value and a time stamp. The identification numbers are generated by using the most recent value in the `State`.

3.2 Shared data

Secondly, **shared data** allows for the simultaneous execution of tasks. As tasks are usually not concerned with *how* or *where* data is stored, Michels and Plasmeijer (2012) abstract away from these details using typed abstract interfaces called *Uniform Data Sources* (UDS) or *Shared Data Sources* (SDS). SDSs can be read, written and updated automatically. Moreover, when one task modifies shared data, other tasks can observe this change.

```
:: RWShared r w = { get :: *State -> *(r,*State)
                  , set :: w -> *State -> *State }
```

Internally, SDSs (de)serialize values of arbitrary type by using the `Dynamic` type. An SDS has type `RWShared r w`, defined as the record structure above. A function `createShared` produces a fresh SDS by returning corresponding `get` and `set` functions. The `@>` combinator is used on a task to write the task value to an SDS.

3.3 Generic Interaction

Thirdly, **Generic Interaction** makes it possible to generate user interfaces for any type of data used by tasks. Interactions can be described with arbitrary detail, although this is not necessary to get a fully functional application. There is no need for the programmer to worry about designing the user interface or the required event handling.

User-interactions are defined as interactive tasks (called *editors*) in TOP. They allow a user to enter and modify a visualized value of some (first order) type. Editors follow a model-view pattern where the model is the value of the task and the view is the visualization. The TOP framework handles the events in the view and updates the model accordingly and vice versa.

3.4 Task Composition

Finally, **task composition** is used to construct tasks from core combinator functions, resulting in the *combinator language* mentioned before. Task composition knows three modes:

1. Sequential composition
2. Parallel composition
3. Value transformation

A task stays alive until it is no longer needed. The task *step* operator (`>>*`) is used as sequential composition. It inspects the task value and decides whether or not to step to a next task. The operator takes a list of possible steps of any length. Steps can happen based on a user action (`OnAction`), the current task value (`OnValue`) or an exception (`OnException`). Uncaught exceptions are propagated by the step operator. Plasmeijer et al. (2012) refer to task steps that can continue without interference of the user as *triggers*. These are the abovementioned `OnValue` and `OnException` steps. Triggers have priority over user actions.

To compose parallel (sub)tasks, one can use the `parallel` combinator, of which there are two modes of use: `Detached` and `Embedded`. Subtasks of the former sort get distributed to different users, whereas `Embedded` subtasks are executed by the current user. Both sorts of parallel subtasks can inspect each other's progress.

Chapter 4

Applying Normalized Systems theory to Clean

In order to apply the theory of Normalized Systems, we must first clear up its respective concepts. Therefore, we give an overview of meaning and terminology within the context of functional languages such as Clean (Sections 4.1 and 4.2). Besides that, we explore the feasibility of implementing pattern expanders for the NS elements in Clean (Section 4.3).

4.1 Primitives in Clean

4.1.1 Software entities

Software entities refer to the more fundamental concept of a module and are defined as the instantiations of the constructs provided by the technology environment. In Clean, we have definition modules and implementation modules (in `dcl` and `icl` files). These are *not*, however, considered software entities. The main constructs provided by Clean are type definitions and function definitions. Instantiations of these are therefore considered to be the software entities.

This complies to the idea that Clean does not handle *private* and *public* modules or inheritance. A software entity is known to the application as long as the module it was defined in is imported. Therefore, the modular structure is not formed by the (definition and implementation) modules, but rather by the definitions inside these modules.

4.1.2 Data entities

A data entity is defined as a software entity with various data attributes or fields, including links to other data entities and that has no interface. Let us

first reflect upon the difference between the way data is represented in functional and object oriented languages. Whereas in object oriented languages we'd have a class containing (private) attributes (with getters and setters) of certain types, in functional languages this is defined as a data structure.

We quickly find that type definitions in Clean can be considered data entities. Let us elaborate on this using a code example:

```
1  // Type definitions
2  :: Point    :== (Int, Int)
3  instance toString Point where
4    toString (x,y) = "(" +++ (toString x) +++","+++ (toString y) +++
         ")"
5
6  :: Triangle   :== (Point, Point, Point)
7  instance toString Triangle where
8    toString(p1, p2, p3) = toString p1 +++ ", " +++ toString p2 +++
         ", " +++ toString p3
9
10 // Instances of points and triangle
11 p1 :: Point
12 p1 = (1,1)
13
14 p2 :: Point
15 p2 = (2,1)
16
17 p3 :: Point
18 p3 = (1,3)
19
20 t :: Triangle
21 t = (p1, p2, p3)
```

Listing 4.1: Data entities in Clean

This example illustrates that (an instance of) a type definition can be considered a data entity in Clean. `p1`, `p2` and `p3` are instantiations of the type `Point` and `t` of the type `Triangle`. We see that, under the definition of a data entity above, both the points `p1`, `p2`, `p3` and the triangle `t` can be regarded data entities, where `t` holds references to other data entities.

In functional languages we do not require concepts as *private* and *public* attributes, since all data is passed as arguments of functions to progress through the workflow of the application. Moreover, Clean works using a graph rewriting system. Therefore, defining a `Point p3 = p1` would not technically result in `p3` referring to `p1`, but rather in assigning `p3` to `p1` by value.

4.1.3 Action entities and tasks

An action entity is defined as a software entity that consumes and produces data entities and represents an operation at a given modular level in a hierarchy. Using the definition of software entities, it shouldn't come as a surprise that a function represents an action entity in Clean. Consider, for example, a function `addToPoint :: Point Int Int -> Point` that adds two integers to the x- and y-coordinates of a point. This function can be considered an action entity since it consumes and produces data entities and represents an operation at the modular level.

This leaves us to discuss the role of tasks within the context of Clean. The identification of a task is somewhat arbitrary. It is up to the designer of the system to choose what will be considered a task. The more fine-grained the modular level is, the more separate tasks can be identified. However, any change driver, such as using an external technology, should be a separate task. Although not pointed out by Mannaert et al., this closely relates to the Single Responsibility Principle (Martin, 2003) in object oriented programming. SRP essentially expresses that a module should have only one *reason to change* (one responsibility). Speaking in terms of normalized systems, we have that an entity should have only one responsibility, or one task, as mentioned earlier. Mannaert et al. (2012) refer to *reasons to change* as *change drivers*.

4.2 Implications for the NS Theorems

In this section we discuss the roles of the Normalized Systems theorems in the context of Clean and with the previously discussed definitions of the software entities. We repeat each of the theorems before going into them. As with all sections in this chapter, our goal is to analyze the concepts of Normalized Systems in the context of functional programming languages.

We consider deletion a matter of garbage collection, as done by Mannaert et al. (2012). Deletion of entities or fields usually means these entities or fields are not necessary anymore. In this case, one could simply stop using them. In a stable system, these unused parts do not affect the rest of the system.

We consider modifications as a combination of deletion(s) and addition(s). A change of type of a field, for example, is actually an addition of a new field with the new type. In stable systems, the addition of this field does not affect the rest of the system. When the old field is not used anymore, it can be deleted as a garbage collection process.

4.2.1 Separation of Concerns

> *An action entity can only contain a single task in normalized systems.*

Continuing on the idea that a task refers to a *responsibility* or *reason to change*, we get that a Clean function should have only one responsibility. This means that a function can be separated into two function definitions whenever it has two or more responsibilities.

As an example, consider an application that has the functionality to print out a complex pdf-document for the tax authority. The document is computed from given tax data and has a specific format. Assume that this is achieved in one action entity (function). Then apparently this software entity is not only aware of computing the required data, but also how to format the data on the document. Changing either the format or the computation then leads to unrelated changes within the software entity. Thus, the software entity should be split into separate action entities.

This also means that cross-cutting concerns such as logging should always be implemented in a separate action entity. Therefore, a function should not be able to both perform its functional task, such as a computation, and perform a supporting task, such as logging, at the same time. Supporting tasks are generally seen as a separate change driver or responsibility.

4.2.2 Data Version Transparency

> *Data entities that are received as input or produced as output by action entities, need to exhibit version transparency in normalized systems.*

In order to adhere to the theorem of Data Version Transparency, we will consider record structures in Clean. A nice property about record structures is that adding a field to the structure does not require changing the functions that consume the type if the function uses selectors, like so:

```
1 isMotherOf :: Person Person -> Bool
2 isMotherOf mom child = child.mother == mom
```

It also does not require changing functions that produce this type as long as they use `&`-notation (called the *update* function) such as in the following:

```
1 setMidName :: Person String -> Person
2 setMidName p m = {p & mid = m}
```

Neither of the functions above are aware of the fields in the `Person` type. They are only concerned with the field(s) they explicitly use. This already offers a large deal of Data Version Transparency. When adding a new field

to `Person`, both functions will stay correct.

This approach is closely related to information hiding in object oriented programming. We could see the selectors (`.`) as getter-operations in the record structure and the update function (`&`) as a setter. However, complete Data Version Transparency is not yet achieved by using just these two operations. Consider, for example, the `Person` record to be:

```
:: Person = { first  :: String
            , last   :: String
            , mother :: Person }
```

Adding a `mid` field to the record will not change action entities that consume a `Person` (as long as they use selectors), but action entities that produce a new `Person` will fail at first, as the `mid` field will not be specified. This expresses the need for every record structure to have a `zero` instance to be defined. Action entities that produce a new `Person` will then have to use this instance in combination with the update function to achieve Data Version Transparency:

```
instance zero Person where
    zero = {frst = "", mid = "", lst = "", mother = zero}

createPerson :: String String String Person -> Person
createPerson first mid last mom =
    {zero & first = first, mid = mid, last = last, mother = mom}
```

Since addition of a field in a record structure does not affect action entities consuming or producing the record structure when using these three functions (`.`, `&` and `zero`), we achieve Data Version Transparency at compile time with respect to the anticipated changes defined earlier.

In general, and as pointed out by Ven, Van Nuffel, Bellens, and Huysmans (2010, p. 40), the use of primitive types (such as `String`, `Int`, `Real`, etc.) violates both Data Version Transparency and Action Version Transparency. Consider an action entity that contains a primitive type in its interface. A new version of the action entity may imply sending more information to the entity. The primitive type is not data version transparent however, as it represents a single value. Therefore, the action entity does not exhibit Action Version Transparency, as it requires changing the interface of the action entity.

As for supporting tasks, these should be implemented at the level of the data entity. For logging, this is easily achieved making use of the `toString` instance. A class `Log a | toString a` can be defined with instances such as `logln` that logs type `a` with a newline character at the end:

```
  logln :: *File a -> *File
  logln io x = io <<< (toString x) <<< "\n"
```

This way, we preserve Data Version Transparency, since a new version of a data entity only implies changing the `toString` instance of the entity.

Algebraic Data Structures

In additon to record structures, Clean also offers Algebraic Data Types (ADT). ADTs can be made polymorphic by adding type variables to the type constructors. Several data constructors (*variants*) can be used in one ADT, each with zero or more type arguments. Finally, ADTs can be recursive.

```
1 :: Tree a = Leaf
2           | Node a (Tree a) (Tree a)
```

The type definition above is a classic example of a recursive polymorphic ADT, representing a tree structure taking values of type `a`. If we look at ADTs from an evolutionary perspective, we find a few implications. We immediately see that ADTs have an interface, as both the type constructor (the left-hand side) and the data constructors (the right-hand side) take type arguments. This requires functions consuming ADTs as input to *pattern match* on the ADT. Functions producing the ADT need to deliver any of the variants, using its data constructor. Consequently, functions producing ADTs need to be aware of the structure of the variants. Therefore, a change in any of the data constructors results in combinatorial effects.

However, these combinatorial effects can be avoided by defining getter, setter and creation functions for each ADT. Using ADTs this way is similar to using record structures with the selector, update and `zero` functions. Since all record structures come with the selector and update functions, it may be more convenient to use record structures as main data construct.

4.2.3 Action Version Transparency

> *Action entities need to exhibit version transparency in normalized systems.*

Action Version Transparency expresses the need to wrap tasks in wrapper functions. If the implementation of a task changes, one would have to change all other entities that call the task directly. If, on the other hand, the task is called indirectly using a wrapper function, a new version of the task only implies a change in the function wrapping the task. This implies that every task must be wrapped in a function in order to deal with additional versions of the task.

In Clean, polymorphism is achieved by using function overloading. Different versions of a task can be defined as instances of a function class in

Clean. For example, a function that computes subsidy data can have multiple versions for cars and for houses. The same function can exist for both versions by using an overloaded function. Therefore, function overloading allows for additional versions of tasks that require different types of data.

As discussed earlier, in order to achieve Action Version Transparency, action entities may never contain primitive types in their interface. This would violate both the theorem of Data Version Transparency and of Action Version Transparency.

4.2.4 Separation of States

> *The calling of an action entity by another action entity needs to exhibit state keeping in normalized systems.*

Separation of States expresses the need for the definition of action states. Every call to an action entity needs to exhibit state, giving an asynchronous way of calling components. How Separation of States must be realized in a functional language is not immediately clear. Ven et al. (2010) describe a violation in the context of Java:

> The third violation occurs when a method throws a custom exception (i.e., an exception that is not part of the default Java environment). This constitutes a violation to the separation of states theorem and therefore results in combinatorial effects.

The concept of exceptions is not a standard construct in Clean, however. Error handling is usually achieved by using data structures such as `Maybe`. If one wishes to also receive an error response if an exception occurs, one could define a structure such as:

```
1 Exception :: Just x | Error String
```

However, the introduction of such a new error state in a new version of an action entity indeed violates the theorem of Separation of States, as illustrated in Listing A.1 and A.2 in the Appendix. Moreover, this only goes as far as *synchronous* exception handling in the sense that an exception is raised directly by the program itself. However, a handful of papers have shown that asynchronous exceptions are possible in both Haskell (Marlow, Jones, Moran, & Reppy, 2001) and Clean (Achten & Plasmeijer, 1994; Van Weelden & Plasmeijer, 2002), but that will not be our focus.

4.3 NS Elements in Clean

In this section, we propose an implementation strategy for the five elements as described by Mannaert et al. (2012); Oorts, Ahmadpour, Mannaert, Verelst, and Oost (2014). In order to reduce the complexity of this

analysis, we have chosen to regard smaller systems that, for example, do not make use of external technologies. This leads to incompleteness of these descriptions in the sense that not all elements have trivial implementations. However, more indications are found in the analysis of iTask in Chapter 5.

4.3.1 Data elements

Data elements need to encapsulate data entities in a data version transparent way. We disregard the use of supporting tasks on data elements for now. As discussed earlier, using record structures already provides a large deal of data version transparency. We show that a record structure with its defined `zero` instance is stable with respect to the anticipated changes:

- Addition of a field, as discussed earlier, does not affect the rest of the system and can be done according to theorem 2 as long as action entities use selectors when consuming the record or use the `zero` and `&`-function when producing one. Adding a field only requires an addition to the record structure and to its respective `zero` instance.
- Addition of a data element implies defining a new record structure, which in turn requires the addition of its respective `zero` instance.
- Additional action elements consuming or producing data elements need to use selectors, the update function and their respective `zero` instances, as discussed in the following subsection.

Therefore, record structures are suitable for the Clean implementation of data entities, as they achieve Data Version Transparency. Data entities are wrapped in their respective elements, also using record structures.

To exhibit state according to theorem 4, data elements need to somehow be linked to a state. There are multiple ways to achieve the stable addition of a state to a data element. An obvious option is to include the state as a field in the record structure encapsulating the data entity. This way, the state is stored in a data version transparent way.

The supporting task of persistency would involve automatically serializing the data element and would have to be done on every change of the data element. This allows the system to be accessible even after failure, using the serialized data elements and their state to continue working. This expresses the need for a generic way of serialization. In Chapter 5 we will see that this can be achieved using the Shared Data Sources described in Section 3.2.

4.3.2 Action elements

As an action element represents an atomic action without any internal workflow, we define action elements to be functions with a single task, complying

to theorem 1. Examples include: the encryption of a file, the computation of taxes, the generation of a form, etc. To achieve Action Version Transparency (theorem 3), a function must always be wrapped in order to deal with additional versions of a task. To further deal with additional versions of tasks, one should use function overloading for action elements. This way, a new version of a task using a different type than its original, will not affect any existing calls to the existing action element. Action elements consuming data elements as input must always use selectors. Action elements producing new instances of data elements must use their respective `zero` function and/or the `&`-operation as discussed earlier.

4.3.3 Workflow elements

Workflow elements must contain the sequence of action elements to execute a workflow. They can not have any other functional tasks beside that. In order to exhibit state and comply to theorem 4, the calling of each action element must be stateful. Moreover, the state must be linked to the instance of the data element that served as argument. Nuffel, Huysmans, Bellens, and Ven (2010) and Krouwel and Op 't Land (2011) have demonstrated that decision rules should be separated from the workflow to comply with the theorem of Separation of Concerns. Therefore, a workflow element does not evaluate the states returned by action entities. State checking and control is explicitly done by trigger elements. We will discuss these in the next section.

Implementations of workflow elements in Clean are represented by functions taking the target data element as input and applying the appropriate actions based on its current state and transition. The way this is done must be abstract and uniform, *i.e.* at the level of a workflow element, defining a transition should be consistent with each action element that returns a state. If this is not the case, we clearly do not comply with the theorem of Action Version Transparency, as each version of an action element would require a different way of handling.

Supporting tasks can be added to a workflow as necessary. Logging, for example, can be achieved by adding the `*World` type to the interface of the workflow. The `*World` type can provide the function with `IO`, making logging possible. Since workflow elements are functions themselves, this needs to happen in a stable manner. This again emphasizes the need for action version transparent workflow elements. If we encapsulate the `*World` in a data element, we can easily achieve this. Additionally, function overloading may help to achieve polymorphism on workflow elements, thus eliminating combinatorial effects arising from differently typed workflow elements.

4.3.4 Trigger elements

Trigger elements are responsible for controlling states and checking whether an action or workflow element needs to be called. It is not immediately apparent what the implementation of trigger elements in Clean should be. The use of a trigger element becomes more clear in a distributed system with human actors. Such a system often requires some sort of event handling. We disregard the details of how these events are structured and how they are sent and received.

As events come in, a trigger element has to react on it based on the current state. An event may hold information about actions in the system, or indicate some exception. The goal of the trigger element is to update the state and indicate what action element needs to be invoked next. This implies that a trigger element is a function that has access to the current state and can interface with existing action elements. The instantiation of a new trigger element then involves defining what action element should be triggered and some rule that decides whether the action element should be invoked, based on the state and/or the incoming event.

To achieve Action Version Transparency and to add supporting tasks, the same rules apply as for the workflow element.

4.3.5 Connector elements

Connector elements need to ensure that data elements are exposed to (possibly external) actors. They ensure external systems are able to interact with the system without calling elements in a stateless way. In addition, Mannaert et al. (2012) describe user connector elements. These elements are responsible for the user interaction of the application and exposing data elements through a user interface. This clearly implies separation of application logic and tasks involving user interaction. Protocol connector elements, on the other hand, have the responsibility of sending and receiving data messages over a certain protocol. Both connector elements need to exhibit state by making use of the data element.

Chapter 5

The Evolvability of iTask

This chapter is aimed at the core subject of this thesis. We will apply the theory of Normalized Systems to the iTask framework. To this extent, we explore the concepts of iTask introduced in Chapter 3 and contrast them to the Normalized Systems theory introduced in Chapter 2 and discussed further in Chapter 4. Our goal is to compare the TOP paradigm to the NS theory in an attempt to explore the evolvability of iTask applications.

We start this chapter by discussing three themes that cover all iTask concepts (Section 5.1 to 5.3). The sections discussing these themes will start by describing the related concepts in iTask and then listing related NS concepts. The relation between both domains are then discussed in a separate subsection. In Section 5.4 we give a summary of this chapter. We will refer to tasks in the context of TOP to *iTasks*, in order to distinguish them from tasks in the context of NS.

5.1 Data Representation

Before all else, we focus on the data representation in iTask systems. The first form of data is identified as the *task result* of an iTask. The task result is a structure `TaskResult a` that contains either a *task value*, denoted `Value a`, or an *exception*. The iTask framework introduces functions to:

- check whether the `Value` structure contains a value (`hasValue`),
- retreive the value from the `Value` structure (`getValue`) and
- change the task value, resulting in a new task (`@?` and `@`).

In other words, the framework provides functionality similar to a getter and setter, as well as a check for empty values, for the main data construct.

Shared Data Sources (Section 3.2) are the second notion in iTask involving data representation. SDSs makes automatic sharing of task values pos-

sible. Since the serialization performed by SDSs makes use of dynamics, task values of any type can obtain an SDS. Writing a value to an SDS is achieved by using the `@>` combinator on a task. This results in the reactive behaviour of iTasks automatically (de)serializing their values. Under water this is achieved by defining a `get` and `set` method for each SDS.

Related NS concepts identified for the following discussion involve: the Data Version Transparency theorem, the Separation of States theorem, data elements, data entities, supporting tasks.

5.1.1 Discussion

It quickly becomes clear that instantiations of the task value type (`Value a`) can be considered data entities[1]. The `TaskResult` type encapsulates this entity and gives it a time stamp. A `TaskResult` can also be an exception:

```
1 :: TaskResult a =      ValRes TimeStamp (Value a)
2                  | E.e: ExcRes e & iTask e
```

An exception can simply be considered a form of error state. Normalized Systems theory points out that states must be linked to or part of data elements, but does not tell exactly how. In this case, the exception is a variant in the data structure defining `TaskResult`. It can therefore be considered as linked to the data element. This tells us that instantiations of `TaskResult` can be considered data elements.

The encapsulation of values in `TaskResults` achieves Data Version Transparency for the most part. However, this is greatly due to the generic nature of the evaluation of `Tasks`. This is a rather complex process which we abstract from at the specification of iTask applications. For instance, consider a task of non-transparent type, *e.g.* a `String`. Changing the task type from `String` to `Int` does not necessarily break the task, because it will be handled automatically by the framework. Yet it can be troublesome when using the step operator (`>>*`; Section 3.4), since the `TaskStep` expects something of the same type as the task it is used on. If we assume each task only has a limed amount of steps, this effect does not grow with the system. Therefore, we conclude that Data Version Transparency is kept.

The Shared Data Sources achieve the supporting tasks of persistency and remote access. Persistency is done automatically, as each time the value of an iTask changes, it gets serialized. This is implemented at the level of the data element. On the level of an iTask definition, we are not concerned with how serialization works internally and seems evolvable in that sense.

[1]There is a slight subtlety here. Since `Value a` also captures `Stability`, which can be seen as a state, it is actually a data entity that has already been linked to a state.

5.2 Stateful Workflow

In this section, we focus on how iTask systems carry out workflows and how states are kept. This includes the subject of exception handling. The main concept that glues tasks together in iTask systems is, of course, task composition. An iTask is recursively rewritten until it has a stable value or an uncaught exception is raised. When an iTask has an unstable value, it means the iTask is not finished yet, and the continuation task will be rewritten instead.

Sequential tasks are built up using the task step operator `>>*`. The operator takes a task and a list of possible steps (Section 3.4). Steps contain a predicate to test the task value and a function that passes the task value to the new task.

Parallel tasks can be constructed using the `parallel` operator. The `-&&-` (*and*) and `-||-` (*or*) combinators make use of the operator and can be used to start two tasks in parallel. Parallel tasks constructed with `-&&-` give a pair of task values. Parallel tasks constructed with `-||-` have a stable value if either of the individual tasks has a stable value. Otherwise it will either be unstable or non-observable.

Related NS concepts identified for the following discussion involve: the Separation of Concerns theorem, the Separation of States theorem, workflow elements, trigger elements, action elements.

5.2.1 Discussion

The concept of task composition in iTask systems is an obvious form of workflow. We consider iTasks that are executed in parallel as one, because the parallel iTask as a whole needs to be evaluated before progress in the workflow is possible. We can then consider the step operator (`>>*`) with its three cases (`OnAction`, `OnValue` and `OnException`). The step operator can take one or more steps. Each time the step operator is used, it builds up the workflow, because it represents a transition in the workflow. Each `TaskStep` in the second argument of the operator represents a separate state and a transition to the task that reacts on this state. This state relates to the predicate and action given to the `TaskStep`. This is further illustrated using the `PlanMeeting` example from (Plasmeijer et al., 2012) in the Appendix (Figure A.1)

We notice that task composition is closely related to workflow elements in NS. However, the line that separates the concept of such a workflow element with the concept of an action element in iTasks is not entirely clear.

In the `PlanMeeting` example, we see this clearly in the `decide` task step. This step calls the `pick` task, which in turn starts a (tiny) workflow, as it calls the step operator. This leads to the idea that an iTask can fulfill both a functional task and a task concerning the workflow of the application, possibly violating the theorem of Separation of Concerns. For now, we will consider iTasks purely as workflows. In Section 5.3 we will elaborate on this.

There is more to the step operator, however. The `TaskSteps` it takes as argument indicate what task must be executed when something happens. Internally, this takes place based on the state of the iTask that served as argument. This includes how to handle exceptions. Such a construct clearly relates to the concept of trigger elements in NS. A `TaskStep` controls the task state and checks whether a certain action has to be triggered. In Normalized Systems theory, this is precisely what defines a trigger element.

5.3 User Interaction and Communication

In iTask systems, user interaction is achieved using editors (Section 3.3). Internally, one core editor function (`edit`) is defined. Besides that, iTask offers a handful of predefined editors that are derived from this core editor. These are functions like `updateInformation`, `viewInformation`, `enterChoice`, etc. These interactive tasks make user interaction possible in a completely automatic way.

Custom user interaction is made possible by giving an interaction task a *views* parameter. This parameter can be defined independent of the interaction task. The framework also offers types for user interface controls and an annotation operator (`<<@`) to tweak the layout.

In addition, sending information from server to client (and vice versa) is completely abstracted from as well. The programmer does not need to be concerned about what protocol or format to use, as this is taken care of by the framework. Nonetheless, third party protocols can be defined if one wants to integrate an iTask application with another application. The code for this can be separated from the application code.

Related NS concepts identified for the following discussion involve: the Separation of Concerns theorem, action elements, connector elements.

5.3.1 Discussion

The abstraction of user interaction by means of editors completely separates it from the rest of the application. This indicates the presence of some user connector element in the framework. Naturally, this connector element must

be implemented at the front-end level of the application. As a consequence, the connector element and the way it is implemented are not a factor in the evolvability at the level of the iTask specification.

The same applies for the client-server communication. There must be a protocol connector element that ensures the data elements can be interacted with externally. However, this is hidden inside the framework and therefore not important at the level of an iTask specification.

This leaves us to discuss the role of the editors at the level of iTask specifications. Earlier, we found that iTasks can be considered workflows, but seem to be able to introduce functional tasks. However, with the introduction of editors (which are tasks themselves) the separation becomes more clear. Editors do not represent a workflow, but rather something more atomic. As they also state *what* can be done with a certain data element, they characterize a functionality. This leads us to the idea that editors represent action elements in iTask systems.

5.4 Summary

The iTask framework shows a lot of similarities with the Normalized Systems methodology. First off, data is handled by encapsulating task values in `TaskResults`. `TaskResults` act as data elements in the iTask application and offer a large deal of data version transparency. Persistency and remote access is offered by the use of SDSs. This is an automatic process and seems stable in that sense.

`Tasks` can be seen as workflow elements. However, the line between workflow elements and action elements is not very sharp. Editors (such as `updateInformation`, `enterChoice`) seem the best definition of action elements in iTask applications. Workflows are built up using the step operator (`>>*`) on `TaskSteps`. `TaskSteps` are obvious forms of trigger elements, as they control state and choose what action should be taken next.

Both user interaction and communication are hidden away from the iTask programmer. Therefore, at the level of an iTask definition, these do not have effect on the evolvability of iTask applications.

Element	Realized by
Data element	Encapsulated task values in `TaskValue`. `TaskValues` are wrapped in a `Task` and achieve data version transparency.
Workflow element	Workflow elements are realized by `Tasks`. This happens in combination with the step operator (`>>*`).

Action element	Editors seem the best definition of action elements in iTasks. Since editors are tasks themselves, the line between action elements and workflow elements is not entirely clear.
Trigger element	`TaskSteps` are obvious forms of trigger elements in iTask. They control state and check what action needs to be called.
Connector element	Connector elements are fully hidden away in iTask. The internal functionality of editors realizes user interaction. Communication concerns are abstracted from.

Table 5.1: **The relation between NS elements and iTask constructs**

Action Version Transparency

In our discussion, we tried to explain the theorems as complete as possible. However, a theorem that got overlooked was the Action Version Transparency theorem. We may argue that the mere addition of a `Task` does not lead to combinatorial effects. The addition of this task to a workflow implies defining a `TaskStep` using the task. Adding the step to an existing list of `TaskSteps` (as a new *possible* step) only implies adding another value in the list. However, inserting the step into the workflow separately (as an intermediate task), may require a change to each of the possible steps following it. Therefore, we may argue that the theorem of Action Version Transparency is complied to if the amount of steps is limited.

Theorem	Realized by
Separation of Concerns (SoC)	Separation of Concerns is achieved using `Tasks` as workflows and editors as actions. Editors are atomic and therefore compliant to SoC. The supporting tasks of `TaskResults` are abstracted from and are of no concern on the level of iTask definitions.
Data Version Transparency (DVT)	For a large part, DVT is achieved by the generic nature of `Tasks` and only truly holds if we may assume the amount of task steps on each task is limited.
Action Version Transparency (AVT)	Assuming the amount of task steps is limited, we achieve AVT because addition of a new (version of) a task only leads to coding changes to the task steps.
Separation of States (SoS)	A state is linked internally to a `Task`. This is abstracted from on the level of iTask definitions. Task steps provide state handling in the form of triggers.

Table 5.2: **The relation between NS theorems and iTask constructs**

Chapter 6

Related Work

Despite the novelty of Normalized Systems, a fair amount of research has been done on the theory. A mentionable related research is a master thesis by Krouwel (2010); (Krouwel & Op 't Land, 2011). In the research, Normalized Systems theory is combined with an enterprise modelling methodology called DEMO. Much like in our thesis, the research compares Normalized System with another methodology and attempts to find similarities in the two domains. However, the research does not focus on the technicalities of implementing a stable system in a technological environement.

The feasibility study by Ven et al. (2010) can be mentioned, because it explores the theory of Normalized Systems in a specific technology environment. This study puts focus on the Java programming language though, and we've seen that not all of these findings can be put to use in functional languages with different constructs.

Remaining in the subject of Normalized Systems, we find the work by de Bruyn et al. (2012). They make use of the NS theory by exploring its design patterns as an effective facilitation tool for, among others, more efficient documentation and the development of new applications. The works above show that the theory is general enough to work with a wide range of fields. Similarly, this thesis describes how it can be used in functional programming.

To our knowledge, this is the first thesis that attempts to combine the theory of Normalized Systems with functional programming (Clean, particularly) and the novel paradigm of task oriented programming. Of course there has been research about evolvability in functional programming, however. Examples include: (Krishnamurthi, Felleisen, & Friedman, 1998), (Lämmel & Visser, 2002), (Lämmel & Jones, 2003), (Antoy & Hanus, 2002) and many more. These works involve finding good design practices in functional programming.

Chapter 7

Conclusions & Future Work

In this thesis, we explored the Normalized Systems theory in a functional programming context. To this extent, we have delved into its concepts, such as primitives, and its four theorems and described them using constructs of the functional language Clean. Additionally, we examined the Normalized Systems elements in an attempt to develop patterns for these elements in Clean and functional languages in general.

The other part of this thesis focused on the evolvability of iTask systems, using Normalized Systems theory. Interestingly, the two approaches have a lot in common. Concepts described in Normalized Systems can be found back in the paradigm of Task Oriented Programming and its implementation. We conclude that iTask offers a way of implementing applications in a reasonably stable way.

Nevertheless, the iTask framework is not entirely compliant to all Normalized Systems elements. Mannaert et al. (2012, p. 114) point this out, as their design patterns were described in Java EE and Cocoon and future efforts could focus on developing patterns for different technology environments. A similar effort has been made in this thesis, but this can benefit from more research. The latter applies not only to the pattern descriptions in Clean, but also to the transformation of iTask applications into systems that fully adhere to the Normalized Systems theory. Hereto, the framework needs to be analyzed in more detail and a solution is necessary to resolve the previously discussed overlap of concepts.

References

Achten, P., Koopman, P., & Plasmeijer, R. (2015). An introduction to task oriented programming. In V. Zsók, Z. Horváth, & L. Csató (Eds.), *Central european functional programming school: 5th summer school, cefp 2013, cluj-napoca, romania, july 8-20, 2013, revised selected papers* (pp. 187–245). Cham: Springer International Publishing. Retrieved from http://dx.doi.org/10.1007/978-3-319-15940-9_5 doi: 10.1007/978-3-319-15940-9_5

Achten, P., & Plasmeijer, M. (1994). A framework for deterministically interleaved interactive programs in the functional programming language clean. In *Proc. computing science in the netherlands, csn* (Vol. 94, pp. 21–22).

Antoy, S., & Hanus, M. (2002). Functional logic design patterns. In Z. Hu & M. Rodríguez-Artalejo (Eds.), *Functional and logic programming: 6th international symposium, flops 2002 aizu, japan, september 15–17, 2002 proceedings* (pp. 67–87). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-45788-7_4 doi: 10.1007/3-540-45788-7_4

de Bruyn, P., Huysmans, P., Oorts, G., Nuffel, D. V., Mannaert, H., Verelst, J., & Oost, A. (2012). Using normalized systems patterns as knowledge management. In *The seventh international conference of software engineering advances (icsea) 2012, november 18-23, lisbon, portugal* (pp. 28–33).

Herraiz, I., Rodriguez, D., Robles, G., & Gonzalez-Barahona, J. M. (2013, December). The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Comput. Surv.*, *46*(2), 28:1–28:28. Retrieved from http://doi.acm.org/10.1145/2543581.2543595 doi: 10.1145/2543581.2543595

Krishnamurthi, S., Felleisen, M., & Friedman, D. P. (1998). Synthesizing object-oriented and functional design to promote re-use. In E. Jul (Ed.), *Ecoop'98 — object-oriented programming: 12th european conference brussels, belgium, july 20–24, 1998 proceedings* (pp. 91–113). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/BFb0054088 doi: 10.1007/BFb0054088

Krouwel, M. R. (2010). *Towards the agile enterprise: A method to come from a demo model to a normalized system, applied to government subsidy schemes* (Unpublished master's thesis). TU Delft, The Netherlands.

Krouwel, M. R., & Op 't Land, M. (2011). Combining demo and normalized systems for developing agile enterprise information systems. In *Enterprise engineering working conference* (pp. 31–45).

Lämmel, R., & Jones, S. P. (2003). *Scrap your boilerplate: a practical design pattern for generic programming* (Vol. 38) (No. 3). ACM.

Lämmel, R., & Visser, J. (2002). Design patterns for functional strategic programming. In *Proceedings of the 2002 acm sigplan workshop on rule-based programming* (pp. 1–14). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/570186.570187 doi: 10.1145/570186.570187

Lehman, M. M. (1996). Laws of software evolution revisited. In *Software process technology* (pp. 108–124). Springer.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997, Nov). Metrics and laws of software evolution-the nineties view. In *Software metrics symposium, 1997. proceedings., fourth international* (p. 20-32). doi: 10.1109/METRIC.1997.637156

Mannaert, H., Verelst, J., & Ven, K. (2008). Exploring the concept of systems theoretic stability as a starting point for a unified theory on software engineering. In *Software engineering advances, 2008. icsea'08. the third international conference on* (pp. 360–366).

Mannaert, H., Verelst, J., & Ven, K. (2011, December). The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability. *Sci. Comput. Program.*, *76*(12), 1210–1222. Retrieved from http://dx.doi.org/10.1016/j.scico.2010.11.009 doi: 10.1016/j.scico.2010.11.009

Mannaert, H., Verelst, J., & Ven, K. (2012). Towards evolvable software architectures based on systems theoretic stability. *Software: Practice and Experience*, *42*(1), 89–116.

Marlow, S., Jones, S. P., Moran, A., & Reppy, J. (2001). Asynchronous exceptions in haskell. In *Acm sigplan notices* (Vol. 36, pp. 274–285).

Martin, R. C. (2003). *Agile software development: Principles, patterns, and practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Michels, S., & Plasmeijer, R. (2012). Uniform data sources in a functional language. In *Submitted for presentation at symposium on trends in functional programming, tfp* (Vol. 12).

Nuffel, D. V., Huysmans, P., Bellens, D., & Ven, K. (2010, Aug). Translating ontological business transactions into evolvable information systems. In *Software engineering advances (icsea), 2010 fifth international conference on* (p. 58-63). doi: 10.1109/ICSEA.2010.16

Oorts, G., Ahmadpour, K., Mannaert, H., Verelst, J., & Oost, A. (2014). Easily evolving software using normalized system theory: A case study.

In *The ninth international conference on software engineering advances* (pp. 322–327). Retrieved from https://www.thinkmind.org/download.php?articleid=icsea_2014_12_50_10219

Oorts, G., Huysmans, P., Bruyn, P. D., Mannaert, H., Verelst, J., & Oost, A. (2014, Jan). Building evolvable software using normalized systems theory: A case study. In *System sciences (hicss), 2014 47th hawaii international conference on* (p. 4760-4769). doi: 10.1109/HICSS.2014.585

Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., & Koopman, P. (2012). Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on principles and practice of declarative programming* (pp. 195–206). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2370776.2370801 doi: 10.1145/2370776.2370801

Van Weelden, A., & Plasmeijer, R. (2002). Towards a strongly typed functional operating system. In *Symposium on implementation and application of functional languages* (pp. 215–231).

Ven, K., Van Nuffel, D., Bellens, D., & Huysmans, P. (2010). The automatic discovery of violations to the normalized systems design theorems: A feasibility study. In *Software engineering advances (icsea), 2010 fifth international conference on* (pp. 38–43).

Appendix A

Appendix

```
safeDiv :: Real Real -> Real
safeDiv x y = if (y == 0.0)
              (abort "Can not divide by zero")
              (x / y)

f1 :: Real
f1 = safeDiv 1.0 0.0
f2 :: Real
f2 = safeDiv 1.0 1.0
  [...]
f_N :: Real
f_N = safeDiv 1.0 N
```

Listing A.1: Separation of States violation (a)

```
:: Exception x = Just x
               | Error String

safeDiv :: Real Real -> Exception Real
safeDiv x y = if (y == 0.0)
              (Error "Can not divide by zero")
              (Just (x / y))

f1 :: Exception Real
f1 = safeDiv 1.0 0.0
f2 :: Exception Real
f2 = safeDiv 1.0 1.0
  [...]
f_N :: Exception Real
f_N = safeDiv 1.0 N
```

Listing A.2: Separation of States violation (b): N coding changes are necessary to deal with the new Exception state.

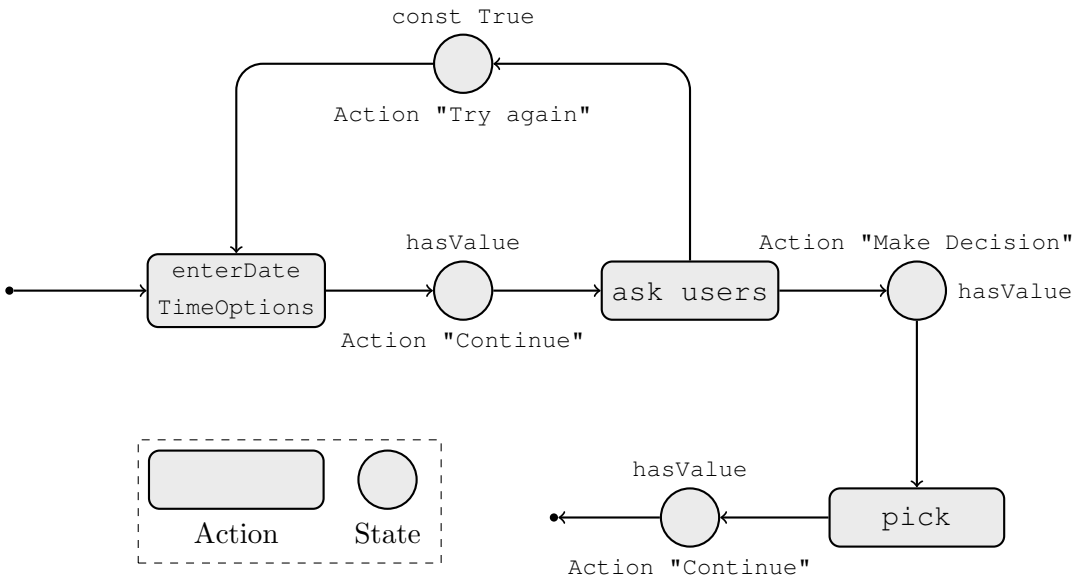

Figure A.1: the workflow of the `PlanMeeting` example. Actions relate to tasks and states relate to task steps and their given predicates.