The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability

Herwig Mannaert *, Jan Verelst, Kris Ven
Department of Management Information Systems, University of Antwerp, Prinsstraat 13, 2000 Antwerp, Belgium

A R T I C L E   I N F O

Article history:
Available online 7 December 2010

Keywords:
Systems theory
Normalized systems
Stability

A B S T R A C T

Evolvability is widely considered to be a crucial characteristic of software architectures, particularly in the area of information systems. Although many approaches have been proposed for improving evolvability, most indications are that it remains challenging to deliver the required levels of evolvability. In this paper, we present a theoretical approach to how the concept of systems theoretic stability can be applied to the evolvability of software architectures of information systems. We define and formalize the transformation of a set of basic functional requirements into a set of instantiations of software constructs. We define this transformation using both a static and a dynamic perspective. In the latter perspective, we formulate the postulate that information systems should be stable against new requirements. Based on this postulate, we derive a number of design theorems for software implementation. Using this transformation we use theoretical arguments to derive that these theorems contribute to achieving stability.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

In today's increasingly volatile environments, evolvability is becoming a crucial characteristic of information systems. Unfortunately, current information systems struggle to provide the requested levels of evolvability. One of the challenges that contributes to this issue is the existence of Lehman's Law of Increasing Complexity which states: "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it." [1, p. 1068]. This law implies that over time, the structure of software will become more complex, thereby requiring increasing effort to add new functionality to an existing system [1–5]. However, the existing empirical evidence for the law of increasing complexity is not 100% conclusive, particularly in the context of open source software (see e.g., [6–9]). Nevertheless, the law relates to common software engineering knowledge that systems over time have a tendency to become increasingly complex and finally become legacy systems that need to be replaced. The consequences of this law become harder to accept in times in which organizations are seeking to increase their efficiency. The increasing rate of change namely suggests an increasing rate of structure degradation, and correspondingly requires increasing resources. Many studies have therefore focused on how this can be prevented. Some of these studies have taken stability as their starting point for studying evolvability (see e.g., [10–14]). However, the meaning of stability varies across different studies [15].

In this paper, we present a theoretical approach to how the concept of systems theoretic stability — as defined and used in systems theory — can be applied to the evolvability of software architectures of information systems. In the fields of signal

---

* Corresponding author. Tel.: +32 3 265 41 89; fax: +32 3 265 40 64.
E-mail addresses: herwig.mannaert@ua.ac.be (H. Mannaert), jan.verelst@ua.ac.be (J. Verelst), kris.ven@ua.ac.be (K. Ven).

1 We mainly focus on information systems in this paper since evolvability is becoming a critical issue for organizations. Nevertheless, the approach described in this paper is applicable to any type of software application. Software architecture has been defined as “the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time” [16]. It is further claimed that “software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the “load-bearing walls” of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate the cost of modifications” [16]. In general terms, software architecture refers to the high-level design of a system.
processing, systems theory, and control theory, the generic concept of stability is defined as \textit{BIBO} (\textit{Bounded Input Bounded Output}) stability and is considered one of the most fundamental properties of a system \cite{17,18}. It implies that a bounded input function should result in bounded output values, even as $T \rightarrow \infty$ (with $T$ representing time). We consider this concept an interesting starting point to study the evolvability of information systems, since this calls for the elimination of any ripple effects. Although we do not claim that the approach described in this paper will be able to eliminate all instabilities, we argue that the approach is instrumental in identifying and significantly reducing the number of instabilities. In addition, we argue that striving to achieve systems theoretic stability of software with respect to changes is a worthwhile goal. Although this paper uses a theoretical approach, future studies can be conducted to provide empirical evidence.

The rest of the paper is structured as follows. In Section 2, we define and formalize the transformation of a set of basic functional requirements into a set of instantiations of software constructs using both a static and a dynamic perspective. We further postulate that information systems should be stable against new requirements. In Section 3, we derive a number of design theorems for software implementation based on this postulate. Related work is discussed in Section 4. Finally, our conclusions are offered in Section 5.

2. Basic implementation model

Functional requirements are realized through the instantiation of software constructs during the implementation process. These constructs — such as functions or classes — are provided by a programming language. Similar to how systems and control theory studies all types of systems based on transformations from input functions to output functions \cite{17,18}, we propose to study the software implementation process as a transformation of functional requirements into software primitives (i.e., the instances of the software constructs).

It is important to note upfront that we will consider these functional requirements to be in an elementary form and very close to traditional software implementation concepts. This allows us to study the process of software coding in its most elementary form. Hence, we do not consider high-level requirements that are obtained by system analysts from traditional requirements gathering techniques (including interviews with stakeholders, questionnaires, and use cases). Examples of such types of requirements could be “We need a system to create and send invoices to customers” or “We need a document management system to support our business processes”. Such requirements are translated in terms that are very close to implementation-related concepts. Each programming language allows one to implement actions that take data structures as input and produce data structures as output. Some programming languages use different constructs for both (e.g., structures and functions), while other programming languages use identical constructs (e.g., classes). The example requirement concerning the invoice system above could, for example, be translated into: “We need a data structure for an invoice that stores the date, customer, VAT, total amount, and the list of products that were sold. We need another data structure to store information on each product (i.e., description, price, and amount). We need an action to calculate the appropriate VAT, an action to calculate the total amount of the invoice, and another action to create a PDF document of the invoice.”. It is clear that this latter type of requirement is much easier to translate into code than the former. However, they are defined as functional requirements and do not make any use of — nor any reference to — specific software programming constructs. How the high-level requirements are translated into this more elementary form is outside the scope of this paper. Although realistic business requirements may seem completely different and at a completely other level of abstraction than our basic requirements, we should not forget that traditional programming languages force us to translate these business requirements into our elementary requirements.

2.1. The static transformation

In this section, we define and formalize the static transformation of a set of basic functional requirements into a set of instantiations of software constructs at a certain point in time.

2.1.1. Functional requirements specification

We distinguish between three basic requirements for an information system. A first requirement is concerned with the fact that information systems must be able to store and represent information. We therefore propose the following requirement:

\textbf{Requirement 1.} An information system needs to be able to represent instances of data entities $D_m$. A data entity consists of a number of data fields $\{a_i\}$. Such a field may be a basic data field, i.e., representing a value, or a reference to another data entity.

In procedural programming languages, a data entity corresponds to a structure or a record, while a data field refers to the fields within this structure or record. In object-oriented programming languages, a data entity corresponds to a class and the data field refers to an attribute of the class. For example, a requirement could be that the information system needs to be able to store information on a customer (e.g., company, contact person, address, and telephone number). This requires that a customer data entity be created with the required data fields. It is important to note that various data entities $D_i$ correspond to different types of data, such as a Customer, an Order, or an Invoice. Every single data entity can have multiple instances, for example the various customers.
The second requirement is concerned with the fact that information systems must be able to perform actions or operations using information. We therefore propose the following requirement:

**Requirement 2.** An information system needs to be able to execute processing actions $P_n$ on instances of data entities. A processing action consists of a number of consecutive processing tasks $\{t_l\}$. Such a task may be a basic task – i.e., a unit of processing that can change independently – or an invocation of another processing action.

In order to allow for modular structure, we distinguish between basic tasks and their aggregation in processing actions. We propose to base the identification of tasks on the concept of change drivers: a task is something that is subject to an independent change. Although we do not require the upfront and correct identification of all change drivers, it allows us to study the effects of separating and/or combining independent change drivers. In procedural programming languages, an action refers to a function or procedure, while in object-oriented programming languages, a processing action corresponds to a method of a class. Each function, procedure, or method consists internally of a number of lines of code that implement the action. These lines of codes can be grouped in separate tasks, based on the fact that they refer to different change drivers. For example, a requirement could be that the information system needs to calculate the VAT for a given invoice. The processing action that implements this requirement may contain multiple change drivers (e.g., when external libraries or systems are used) and thus tasks.

A final requirement is concerned with the fact that information systems must be able to receive input from and produce output to its environment. We therefore propose the following requirement:

**Requirement 3.** An information system needs to be able to input or output values of instances of data entities through connectors $C_l$.

Connectors are processing actions that read values from a peripheral device (such as a keyboard) and assign them to attributes of instances of data entities, or that write values of attributes of these instances to a peripheral device. Similar to processing actions, connectors refer to functions, procedures, or methods. For example, a requirement could be that the information system needs to print a list of customers, which requires that a connector is created.

The functional requirements of an information system can be described as a set of requirements $R$, consisting of a subset of data entities $\{D_m\}$, a subset of processing actions $\{P_n\}$, and a subset of connectors $\{C_l\}$. This model may appear to be a simplification to some extent, as in reality additional cross-cutting concerns may need to be implemented for persistency and access control. These additional constructs correspond in our model to additional processing actions.

### 2.1.2. Software implementation transformation

We study the implementation of software as the transformation of our basic functional requirements into software primitives (i.e., instantiations of the software constructs of a programming environment). We can represent this implementation transformation $I$ as

$$ I = I(R). \quad (1) $$

In this formula, $I$ is expressed as the set of instantiations of constructs (i.e., structures, functions or classes). In a standard procedural or object-oriented programming language, we propose the following straightforward transformation of functional requirements into software primitives:

1. Every data entity $D_m$ is transformed into a data structure $S_m = I(D_m)$, i.e., an instantiation of a software construct for data as provided by the programming language. Examples of such data structures are the struct in C, the record in Pascal, and a class containing member variables in an object-oriented language like Java.
2. Every processing action $P_n$ is transformed into a processing function $F_n = I(P_n)$, i.e., an instantiation of a software construct for processing as provided by the programming language. Examples of such processing functions are the function in C, the procedure in Pascal, and a method contained in a class in an object-oriented language like Java. Processing functions receive input and produce output in terms of instances of data structures.
3. Every input and output connector $C_p$ of a data structure is transformed into a processing function $F_p = I(C_p)$ of the programming language in the same way as a processing action. Such connector functions receive input and produce output in terms of instances of data structures. Although this is to some extent a simplification, this approach is sufficient since processing actions receive input and produce output in terms of data entities. Connectors additionally provide the possibility to read and write the values of the various fields of an instance of a data entity to receive input from, or produce output to, the outside world.

An information system is a set of software primitives $\delta$, consisting of a subset of data structures $\{S_m\}$, and a subset of processing functions $\{F_n\}$. It is important to note that this conceptualization in terms of data entities and processing functions is based on the elementary computation level that is used by current programming languages. This elementary computational model is — at least in an implicit way — present when making use of standard programming languages. Procedural programming languages force the programmer to translate requirements in terms of structures and procedures, while object-oriented programming languages use a single software construct for both data and functions (i.e., the object-oriented class). Similarly, the concept of a task refers to a part of a processing function (i.e., a set of lines of code).
2.2. The dynamic perspective

We now study the dynamic perspective by defining and analyzing the marginal transformation of a set of additional functional requirements into a set of additional instantiations of software constructs in the course of time. It is important to note that we intend to test the impact of a change on the evolvability of an information system to the limit. The reason for this is that we want to identify the root causes of instability. This is achieved in systems theory by considering an infinite period of time, since the root causes of instability will then manifest themselves immediately at a macroscopic level. In the same way, we strive for the identification of the root causes of instability for software evolution. As a result, we will be considering an infinite period of time.

2.2.1. Evolution of functional requirements

Through the course of time, the functional requirements for a system can evolve. In general, three types of situations may present themselves: (1) the addition of new requirements; (2) the modification of existing requirements; and (3) the obsolescence of existing requirements. In this paper, we are mainly concerned with the addition of new requirements to an information system since this allows us to study whether information systems remain stable even when growing over time. If the system is stable, unused parts of the system due to obsolete requirements have no impact on the rest of the system. However, these unused parts can have other negative effects since the system will continue to increase in size. We therefore consider the deletion of obsolete requirements to be an automated process instead of a change to the information system. This process is similar to garbage collection techniques used in programming environments that remove existing object instances when no run-time references remain to such an instance. We therefore envisage a similar garbage collection process that removes unused parts when no references remain to them. Systematic and structured garbage collection will therefore archive and/or remove unused parts without impacting the stability of the system. Modifications are indirectly included as they are considered the combination of additions and deletions.

Concerning data entities, we distinguish between the following types of additional functional requirements:

Requirement 4. An existing information system representing a set of data entities \( \{D_m\} \) needs to be able to represent:
- a new version of a data entity \( D_m \) that corresponds to including an additional data field \( a_i \).
- an additional data entity \( D_m \).

Concerning processing actions, we distinguish between the following types of additional functional requirements:

Requirement 5. An existing information system providing a set of processing actions \( \{P_n\} \) needs to be able to provide:
- a new version of a processing task \( t_j \) whose use may be mandatory.
- a new version of a processing action \( P_n \) whose use may be mandatory.
- an additional processing task \( t_j \).
- an additional processing action \( P_n \).

We use the term “new version of a task” to refer to both new versions of the same task over time (e.g., bug fixes) and alternative implementations of the task (e.g., different encryption algorithms). We make no such distinction and consider them both different versions of a single processing task. We also do not make a distinction between a version that has replaced the previous version and a version that needs to coexist for reasons of backwards compatibility. In our analysis, they are all different versions of a single processing task. Although it is a good practice to maintain as few versions as possible at the same time, there are various circumstances that can and will often force information systems to maintain different versions at the same time. For instance, software products with a large customer or install base — especially in distributed applications — will have to maintain several different versions at the various customers. In order to study evolvability to the limit, we must therefore assume that multiple versions of tasks can exist at the same time.

The mandatory use of a new version of an action or task means that only the use of the new version is allowed throughout the entire information system and that the use of the old version is no longer allowed. Examples are versions of processing tasks that are not compliant with new legislation, such as VAT percentages or auditing rules, or versions that require an operating system that is no longer supported.

We further posit the assumption of unlimited systems evolution, namely that the information system evolves for an infinite amount of time \( (T \rightarrow \infty) \) and that the total number of requirements and their dependencies will become unbounded too. This may seem an overstated assumption, but actually, even the introduction of a single functional requirement every twenty years corresponds to an infinite amount for an infinite time period. Moreover, this assumption of unlimited systems evolution is a very relevant assumption to make in the context of programming-in-the-large, as it exposes modular structures whose evolvability is limited to programming-in-the-small. In systems theory, the problem of achieving limited and controllable output functions — i.e., limited amount of coding — is tackled by identifying the instabilities, the inputs and system interactions that could lead to an unbounded output. The identification of such instabilities is the key to achieving stable, bounded, limited, predictable, and controllable output functions. As the sources of instability do not always manifest themselves immediately on a macroscopic level, the analysis in systems theory is based on an infinite time interval, even though all systems operate in a limited time interval.
For the sake of our analysis of the evolvability of information systems, we consider a worst-case scenario where the following numbers will become unbounded:

- The number of data entities.
- The number of processing actions.
- The number of processing actions receiving a specific data entity as input and the number producing it as output.
- The number of processing actions invoking a single processing action.
- The number of versions of a single processing task.

2.2.2. Change dimensions in software primitives

It is important to realize that the impact of a single additional functional requirement is not necessarily limited to the addition or modification of a single software primitive. As will be shown below, the whole point of our reasoning is that the implementation transformation is fairly straightforward from a static point of view, but becomes complicated and often even uncontrollable in a dynamic context. Although the impact of changes depend on the programming language used as well as the way of working of software engineers, our aim is to strive towards establishing an objective and scientific foundation to analyze the evolvability characteristics of information systems.

Consider the information system as a set of software primitives \( A \), consisting of a subset of data structures \( \{ S_m \} \), and a subset of processing functions \( \{ F_n \} \). Both data structures and processing functions will have to change over time. However, both the new version and the older version(s) may need to exist in parallel. For instance, different versions of a processing function — corresponding to different encryption algorithms or payment systems — may need to be supported. Even different versions of a specific data structure may be required, as certain processing functions may not be able to receive the new version of the data structure as input.

In order to be able to represent various versions of data structures or processing functions, we introduce the temporal or version dimension. As we do not allow modifications of data fields, we only distinguish different versions \( v \) for a data structure as a whole: \( S_m(v) \). For the processing functions, we consider both changes or versions at the function level, and at the task level. Therefore, we consider different versions in various dimensions:

1. the version \( w \) of the interface of the function \( F_n \).
2. the version \( k \) of the various tasks \( t_i \) of the function.

This implies that we have various dimensions of variability in the set of versions of a single processing function. For instance, in case we have different tasks for a specific function \( F_n \), this function becomes dependent on the version of the interface \( w \), and on the versions \( k \), \( l \) of the various tasks \( t_1, t_2, \ldots, t_m \):

\[
F_n(w, t_{1,k}, \ldots, t_{a,l}).
\]

The interface \( w \) is possibly dependent of the various versions \( k, \ldots, l, m \) of the data structures \( S_{1}^{\text{in}}, \ldots, S_{n}^{\text{in}}, S_{m}^{\text{out}} \) that are passed as input or returned as output:

\[
w(S_{1}^{\text{in}}, \ldots, S_{n}^{\text{in}}, S_{m}^{\text{out}}).
\]

This multidimensional variability in time or versions is the underlying reason that the impact of a single additional functional requirement is not necessarily limited to the addition or modification of a single software primitive. For instance, the introduction of a new version of a data structure without supporting the old one, may require new versions of all processing functions that take this data structure as input. The introduction of a new version of the interface of a processing function without supporting the old one, may similarly require new versions of all processing functions that invoke this processing function. On the other hand, supporting the old versions may entail the duplication of future changes. For instance, the mandatory version upgrade of a task, may have to be performed in every function where a version of that task is present. The mandatory upgrade of a function may similarly have to be performed in the various versions of that function.

It should be noted that the dimensions of variability increase significantly in an object-oriented programming environment if we do not use the simplified transformation — defining a separate class for every data entity and processing action — that we proposed. In case we would combine many processing functions in a single class, this class \( C \) would exhibit the version variability of both its data fields and its various methods:

\[
C(S_m(v), \{ F_n(w, t_{1,k}, \ldots, t_{a,l}) \}).
\]

Considering an individual method of the class, the method version becomes at least dependent on the versions of the data fields of the class and of a constructor of the class:

\[
F_n(w, t_{1,k}, \ldots, t_{a,l}, S_m(v), F_{\text{constructor}}(u, t_{1,r}, \ldots, t_{b,l})).
\]

In order for a non-static method \( X \) belonging to class \( A \) to be called by an object \( B \), an object of class \( A \) must first be created by calling its constructor. Hence, there is indeed a (version) dependency between method \( X \) and the constructor of class \( A \) since if the interface of the constructor changes, object \( B \) can no longer call method \( X \) unless the call to the constructor is modified.
2.2.3. Stability of implementation transformation

We apply the systems theoretic concept of BIBO stability to the software implementation transformation by demanding that a bounded set of additional functional requirements \( R_m \) should result in a bounded amount of changes or additions \( \delta_m \) to the set of software primitives, even for an unlimited period of evolution \( T \to \infty \):

\[
\delta_t = I(R \cup R_m) = \delta \cup \delta_m. \tag{6}
\]

Since the assumption of unlimited systems evolution implies that the number of primitives and their dependencies become unbounded, the concept of stability demands that the amount of impacts caused by an additional functional requirement cannot be related to the size of the system: it has to remain constant over time as the system grows. In other words, stability demands that the impact of a change is only dependent on the nature of the functional change itself. Conversely, functional changes causing impacts that are dependent on the size of the system as well as the nature of the change correspond to instabilities of the information system. We call these instabilities combinatorial effects, and require that they are eliminated from the system in order to attain stability.

To clarify the interpretation of an instability as a combinatorial effect, consider a simple first-order model of a dynamic system [17,18]. The evolution in time of a system variable \( y(t) \) with an external input function \( x(t) \) is described by the following differential equation for the continuous case:

\[
\frac{dy(t)}{dt} = x(t) + ay(t) \tag{7}
\]

or the following differential equation for the discrete case:

\[
y[k + 1] - y[k] = x[k] + ay[k]. \tag{8}
\]

In both cases, systems theory tells us that the system is stable if and only if \( a \leq 0 \), demanding that only the external input function \( x \) contributes to the overall system function \( y \). If an interaction or positive feedback mechanism exists between the external input function \( x \) and the overall system state \( y \), the system function \( y \) will become unbounded and the system is considered to be unstable.

In our software implementation transformation, we consider the straightforward implementation of the set of additional or marginal functional requirements \( I(R_m) \) to be the external input \( x(k) \) at a point in time \( k \). The set of additional software primitives \( \delta_m = \delta_t \setminus \delta \) corresponds to the growth of the system state function \( y(k + 1) - y(k) \). However, as we have discussed in the previous section, these additional (versions of) software primitives may require modifications to \( \delta \) and therefore additional versions of \( \delta \) other software primitives. These additional versions of software primitives that are not directly related to the additional functional requirements correspond to the additional term that is related to the size of the existing system:

\[
\delta_m = \delta_t \setminus \delta = I(R_m) \cup \delta \tag{9}
\]

To obtain a scalar equation, we use cardinalities of sets and a coefficient \( a \):

\[
|\delta_m| = |\delta_t| - |\delta| = |I(R_m)| + a|\delta| \tag{10}
\]

or using the discrete variable \( k \) to represent ongoing development iterations:

\[
|\Delta \delta| = |\delta[k + 1]| - |\delta[k]| = |I(R_m[k])| + a|k||\delta[k]|. \tag{11}
\]

Such a coefficient \( a \) is in general dependent on the iteration cycle \( k \), thereby making the model slightly more complicated than the first order model of Eq. (8). Eq. (11), however, clearly expresses the nature of instability in an evolving information system. Instabilities in the evolution of an information system occur when the number of additional software primitives is not only dependent on the amount of additional functional requirements, but also on the set of existing software primitives at that point in time. These dependencies on the size of the system are caused by the dimensions of variability due to the various versions. They are combinatorial effects between the additional functional requirements, and the various existing versions of software primitives.

It must be noted that \( a[k] \) in Eq. (11) may be equal to zero for some iterations. However, \( a[k] \) will not remain zero from a certain point in time onwards, but will systematically have positive values for \( k \to \infty \). Although an information system can be stable at a specific point in time, it is very likely that new instabilities are introduced in the system at a later time when additional requirements are implemented. As a result, the values of \( a[k] \) will not become infinitesimals and the overall integral or sum of \( a[k] \) will not be bounded over an infinite interval. In that case, there is no systems theoretic or BIBO stability.

3. Design theorems for software stability

In this section, we study how information systems can avoid the instabilities or combinatorial effects that we have been discussing. First, we clearly state our goal by postulating that there cannot be any instabilities.
Postulate 1. An information system should not have instabilities: a bounded amount of additional functional requirements cannot lead to an unbounded amount of additional (versions of) software primitives.

Based on this postulate, we will derive a number of design theorems for software implementation. These theorems will be proven by a simple reductio ad absurdum. For each design theorem, we will present a proof and discuss its implications, as well as provide an overview of examples of manifestations of the theorems. These examples illustrate how the theorems relate to well-known design knowledge, thereby proving that they are not new. The main contribution of these theorems is to derive from a single starting point — and therefore unify — several design rules that are in line with the heuristic knowledge of developers. The merits of such unification are clear from other scientific areas (e.g., theoretical physics). Nevertheless, in software engineering, such unification is rather uncommon. An exception may be Lehman, who showed interest in investigating how his eight laws interrelate [19]. Although the theorems are well-known, they are formulated in a much more precise way than is traditionally the case. These design principles are traditionally formulated in much more vague terms, and are — possibly related to this — systematically violated in practice.

3.1. Separation of concerns

The Separation of Concerns theorem is concerned with how tasks are implemented within processing functions. Remember that we identify tasks with processing functions based on the concept of change drivers. A single change driver corresponds to a single concern in the application. In order to achieve stability, a processing function should not address more than one concern and should thus not include more than one task. We therefore formulate the following theorem:

Theorem 1. A processing function can only contain a single task in order to achieve stability.

3.1.1. Proof

Consider a function $F_n$, schematically represented in Fig. 1, with a single version of the interface, a first task with a single version, and a second task with $L$ versions:

$$F_n(w = a, t_{1,1}, t_{2,1}, \ldots, t_{2,L}) \text{ with } l = 1, \ldots, L.$$ (12)

The introduction of a mandatory version upgrade of the task $t_{1,1}$ to $t_{1,2}$ will not only require the creation of the additional task version $I(R_m) = t_{1,2}$, but also the insertion of this new version in the $L$ existing versions of $F_n$:

$$S_m = I(R_m) \cup \{F_n(w = a, t_{1,2}, t_{2,1}, \ldots, t_{2,L})\}_{l=1,\ldots,L}.$$ (13)

The number $L$ refers to the number of versions that exist of the second task $t_2$. According to the assumption of unlimited systems evolution, $L$ will increase over time and will become unbounded, and so will the number of versions of $t_2$. As a result, the number of additional software primitives to implement a given change is not only dependent on the change, but also on the number of versions of $t_2$ — and therefore on the size of the system — and becomes unbounded.

3.1.2. Implications

This theorem expresses the need for the separation of all tasks, in order to obtain — in more general terms — Separation of Concerns [20,21]. This theorem essentially describes the required transition of submodular tasks — as identified based on the concept of change drivers — into functions at the modular level. Hence, each change driver represents another concern for the information system.

Consider a number of functions $F_n, F_m, \ldots, F_p$, represented in Fig. 2, each containing a single task. Suppose those functions contain a piece of code that has not yet been identified as a change driver and that needs to be changed (for instance due to an unexpected lack of support in a new operating system). In this case, the change has to be made in the various functions that contain this piece of code. If this functionality is separated into its own task $t_{1,1}$ in a new function $F_q$, this instability is neutralized for the future. Any future change to this functionality will then only have an impact on $F_q$.

3.1.3. Examples

A first example of this design theorem in literature is the use of a messaging or integration bus to integrate the use of various messaging protocols. The theorem clearly forbids the direct transformation between two external protocols, as such
A transformer module would be aware of two external protocols and therefore subject to two change drivers. The solution is to use an internal reference format or open standard protocol.

A second example of this design principle is the use of an external workflow, as implied by workflow management systems. Few designers will argue that the workflow sequence is not a separate change driver. As a result, the sequence in which a number of actions are processed is in most cases separated from the implementation of the individual actions.

A third example of this design principle is the concept of multiple tiers. The use of multi-tier architectures, such as distributed and/or client–server technology, is aimed at the separation of presentation logic, application or business logic, and database logic [21]. Since such tiers typically use an additional technology to facilitate the implementation of this tier, every tier represents a separate change driver and should be separated.

3.2. Data version transparency

The Data Version Transparency theorem is concerned with how data structures are passed to processing functions. Data Version Transparency is the property that data entities can have multiple versions without affecting the processing functions that consume or produce them. In other words, it should be possible to upgrade a data entity (e.g., by adding a new field) without affecting the processing functions that makes use of that data entity. Hence, data entities should be version transparent to processing functions.

**Theorem 2.** A data structure that is passed through the interface of a processing function needs to exhibit version transparency in order to achieve stability.

**3.2.1. Proof**

Consider a data structure $S_{1,1}$ that is passed, as schematically represented in Fig. 3, through the interface of $L$ processing functions $F_l$:

$$F_l(w(S_{1,1}, \ldots, t_{1,1}) \text{ with } l = 1, \ldots, L)$$

(14)

The introduction of a mandatory version upgrade of the data structure from $S_{1,1}$ to $S_{1,2}$ will require the creation of the additional data structure version $I(R_m) = S_{1,2}$. In case the data structure is not version transparent, it will also demand the adaptation of the code that accesses this data structure in the various functions $F_l$. Therefore, it will require new versions of the $L$ existing processing functions $F_l$:

$$\delta_m = I(R_m) \cup \{F_l(w(S_{1,2}, \ldots, t_{1,2}) \text{ with } l = 1, \ldots, L)\}$$

(15)

The number $L$ refers to the number of processing functions that make use of the data entity $S_1$. According to the assumption of unlimited systems evolution, $L$ will increase over time and will become unbounded, and so will the number of processing functions that make use of the data entity $S_1$. As a result, the number of additional software primitives to implement a given change is not only dependent on the change, but also on the number of processing functions that make use of the data entity $S_1$ — and therefore the size of the system — and becomes unbounded.
3.2.2. Implications

This theorem expresses the need for the encapsulation of data structures, in order to wrap the various versions of the data structure. As a result, new versions of the data structure can be supported without requiring changes to the various processing functions that receive this data structure as input. This way, Data Version Transparency can be obtained.

3.2.3. Examples

The feature of Data Version Transparency is supported in nearly every technology environment and several examples of this theorem can therefore be found in practice.

A first example is the use of XML-based technology such as web services. This implementation has the advantage that new versions of a data entity do not require recompilation of the processing actions that use it and provides version transparency at run-time level.

A second example is the concept of information hiding in object-oriented programming [20,22]. Moreover, in the JavaBean component architecture, data fields or properties are not directly accessible, but can be read and written through set and get accessor methods. The principle of Data Version Transparency is therefore directly supported and encouraged by the constructs of modern component architectures.

A final example can be found in various legacy environments in which data structures are passed using URL’s, property files, or tag-value pairs.

3.3. Action version transparency

The Action Version Transparency theorem is concerned with how processing functions are called by other processing functions. Action Version transparency is the property that a processing function can have multiple versions without affecting any of the other processing functions that call this processing function. In other words, it should be possible to upgrade a processing function (e.g., to implement a new version of a task) without affecting the processing functions that call this processing function. Hence, processing functions should be version transparent to other processing functions.

**Theorem 3.** A processing function that is called by another processing function needs to exhibit version transparency in order to achieve stability.

3.3.1. Proof

Consider a processing function $F_{0,1}$ that is called, as schematically represented in Fig. 4, by $L$ other processing functions $F_l$:

$$F_l(w, t_{l,1}) \quad \text{with } l = 1, \ldots, L.$$  \hspace{1cm} (16)

The introduction of a mandatory version upgrade of the processing function from $F_{0,1}$ to $F_{0,2}$ will require the creation of the additional processing function version $I(R_m) = F_{0,2}$. In case the processing function is not version transparent — i.e., the interface $w = \alpha$ changes to $w = \beta$ — it will also require the adaptation of the code that calls this processing function in the various functions $F_l$. Therefore, it will require new versions of the $L$ existing processing functions $F_l$:

$$\delta_m = I(R_m) \cup \{F_l(w, t_{l,2})\}_{l=1,\ldots,L}.$$  \hspace{1cm} (17)

The number $L$ refers to the number of processing functions that call the processing function $F_0$. According to the assumption of unlimited systems evolution, $L$ will increase over time and will become unbounded, and so will the number of processing functions that call $F_0$. As a result, the number of additional software primitives to implement a given change is not only dependent on the change, but also on the number of processing functions that call $F_0$ — and therefore the size of the system — and becomes unbounded.
3.3.2. Implications

This theorem expresses the need for the encapsulation of processing actions, in order to wrap the various processing action and task versions. Action Version Transparency implies that a separate processing function is needed to wrap the processing actions representing task versions. This way, Action Version Transparency can be obtained.

3.3.3. Examples

Action Version Transparency is also supported by nearly every technology environment and several examples of this theorem can be found in practice.

A first example is the use of polymorphism in object-oriented languages such as Java. Polymorphism is widely accepted to be a good programming practice [23].

Second, in procedural languages such as C, Action Version Transparency is usually implemented through wrapper functions.

Finally, the principle of Action Version Transparency is also used by Interface Definition Languages (IDL) in defining function and method interfaces. These IDLs are used by technologies for distributed or component-based applications written in frameworks such as CORBA and COM.

3.4. Separation of states

The Separation of States theorem is concerned with how calls between processing functions are handled. The contribution of state keeping to stability is based on the removal of coupling between modules that is due to errors or exceptions. Consider a function that looks up the product code. Suppose that this function was originally based on an in-memory table or file and that the implementation is changed to calling a distributed service on the network. In case the interface remains the same, in accordance with both Theorems 2 and 3, there is no impact from an evolvability point of view based on calling this function. However, such a new implementation can entail a completely new and previously non-existing error state (e.g., “the product code service is not available on the network”). It may be desirable — and even necessary — to perform a specific action in response to this error state. If no state is kept in a separate data entity accessible to other functions, the only option is to trigger this action from the calling function. As the number of functions calling this product code function grows with the system and may become unbounded, the impact of dealing with this new error state will become unbounded as well. In case the error state is kept in a separate data entity, a single function could be used to trigger the appropriate response action for the unavailability of the product code service, irrespective of what the calling function was. In other words, the concern is separated. The state keeping data entity should exhibit data version transparency in order to avoid combinatorial effects with the function(s) that need(s) to react to this particular error state. In this way, both the state keeping data entity and the function reacting to this error state can be replaced by new versions and evolve independently.

Theorem 4. Calling a processing function within another processing function needs to exhibit state keeping in order to achieve stability.

3.4.1. Proof

Consider a processing function $F_{0,1}$ that is called, as schematically represented in Fig. 5, by $L$ other processing functions $F_l$:

$$F_l(w, t_{l,1}) \quad \text{with} \quad l = 1, \ldots, L.$$  \hfill (18)

The introduction of a mandatory version upgrade of the processing function from $F_{0,1}$ to $F_{0,2}$ and its task from $t_{0,1}$ to $t_{0,2}$ will require the creation of the additional processing function version $I(R_m) = F_{0,2}(w, t_{0,2})$. However, it may also imply a possible new error state that was nonexistent in the previous version. In case the calling of the function $F_{0,2}$ does not exhibit state keeping, this error state is only known when the function returns, and all the functions calling $F_{0,2}$ need to deal with
this error state. This demands the adaptation of the code in the various functions \( F_i \). Therefore, it will require new versions of the \( L \) existing processing functions \( F_i \):

\[
\delta_m = \{ R_m \} \cup \{ F_i(w, t_{i,2}) \}_{i=1, \ldots, L}.
\]

The number \( L \) refers to the number of processing functions that call the processing function \( F_0 \). According to the assumption of unlimited systems evolution, \( L \) will increase over time and will become unbounded, and so will the number of processing functions that call \( F_0 \). As a result, the number of additional software primitives to implement a given change is not only dependent on the change, but also on the number of processing functions that call \( F_0 \) — and therefore the size of the system — and becomes unbounded.

3.4.2. Implications

This theorem expresses the need for the definition of action states, in order to isolate atomic tasks and to obtain Separation of States.

The state of a processing function has to be kept for every call of the function. Therefore, this state needs to be part of — or linked to — the data structure(s) that served as argument. In this way, the execution of processing actions can be sequenced or scheduled as a state machine: the presence of a certain state in a target data entity will trigger the execution of the processing action for that argument data entity. It is conceivable that both the calling processing function and the callee store a state in order to allow the system to distinguish between the overall state of the workflow and a more detailed outcome of the specific processing action.

Although not required by this theorem, this state-keeping can be performed in a persistent way in order to guarantee the integrity from an operational and transactional point of view. In case the information system crashes, for example due to a power failure, it is the only way to avoid the loss of the information that specific actions have been executed. Whether state is stored persistently not, state keeping allows exception handling not by escalating the error up the calling hierarchy, but by storing the state and having another processing function react to it.

3.4.3. Examples

A first example of this design principle is the use of asynchronous communication systems. Asynchronous communication is a generally accepted design principle in distributed information systems to avoid the escalation of communication problems into application failures [24].

A second example is the concept of asynchronous processing that is more general than distributed communications [24]. In asynchronous processing, the request is stored in a persistent entity, the request is submitted without expecting a response in the synchronous call, and a listening thread is opened to receive the response. This asynchronous model does not make assumptions of the underlying technology and can easily be implemented using synchronous communication middleware such as RPC, CORBA, or web services. As a third example of this principle, workflow systems can be mentioned. The principle of Separation of State corresponds to stateful workflow where intermediate states within a workflow of operations are stored.

4. Related work

Evolvability has been addressed in a very wide range of areas in software engineering, including design patterns, software architectures, refactoring, software product lines, software factories and Lehman’s work on laws of software evolution. We consider Lehman’s work to be extraordinarily valuable, as he pioneered the idea that laws having direct implications for software management can be derived from a set of axioms. Lehman’s approach is also similar to ours since he also sought unification of various well-known design facts by investigating how his eight laws interrelate [19]. The main difference between our approach and Lehman’s well-known work is that he focused mainly on “independent, behavior based statements derived from observation of the real world” [19], and that he does not focus on how precisely his laws are reflected in a software product (i.e., code, or a conceptual model). Our approach is based on theoretical grounds (systems theory) and describes in more detail at the product level where instabilities occur that possibly contribute to the law of increasing complexity. Eden and Mens [25] propose a measurement of flexibility, based on the number of required code changes, and apply this measure to investigate the flexibility of various programming paradigms and design patterns. We agree with their analysis that more objective and precise definitions are needed to evaluate the evolvability paradigms and patterns. Also, their approach is focusing on the analysis of anticipated changes and the number of required code changes, which is similar to our approach. However, their study is not grounded in systems theory, and is not based on the analysis of combinatorial effects, nor on addressing the unification of the design principles discussed in this paper.

Several efforts have also been undertaken to define the term stability in the context of software engineering, both at design and source code level. Yau and Collofello [10] cite early work on stability measures by Soong [26], Myers [27] and Haney [28]. This work dates back from as early as 1972. Yau and Collofello [10] define measures of logical stability of a program and its modules, based on intra-module and inter-module change propagation. In 1985, they presented stability measures at the design level, again based on potential ripple effect characteristics. Their measures are based on the concepts of data abstraction and information hiding, as introduced by Parnas [20]. Li et al. [11] define three metrics
— System Design Instability (SDI), Class Implementation Instability (CII), and System Implementation Instability (SII) — in an attempt to study object-oriented software evolution. The SDI metric is defined as the “percentage of classes whose names have changed + percentage of newly added classes + percentage of deleted classes” [11, p. 375]. Alshayeb and Li [12] use the SDI metric to study system design evolution in agile software processes. Oilage et al. [13] define a revision of SDI — called SDIe — including the concept of entropy and claim that it is easier to calculate and more accurate than SDI. They validate the metric in the context of agile processes. Other work based on the SDI metric or other metrics is Roden et al. [29], Mattsson and Bosch [30], Elish et al. [31] and Tonu et al. [32].

Figueiredo et al. [14] study design stability in software product lines. More specifically, they determine whether the use of aspect-oriented programming prolongs the design stability, based on modularity, change propagation and feature dependency. Their approach is based on analyzing scenarios, without the use of metrics.

Notwithstanding the research described in this paragraph, the amount of literature on stability remains limited. For example, a search in the Web of Science in March 2010 for titles including “stability” and topics on “systems theory” results in merely 13 results when restricting the scope to the computer science and information systems areas. Kelly mentions in her literature survey that “the word stability is used in many different ways in the context of software engineering, often with no precise definition. […] The meaning of stability is generally implied, accepted, but not defined.” [15, p. 316]. Moreover, none of these publications has a basis in system theory, nor do they use a definition of stability that is based on system theoretic foundations. Typically, they are highly tied to a specific metric for stability. From this perspective, our approach can be considered a contribution.

5. Conclusions

In this paper, we have applied the concept of systems theoretic stability to the evolvability of software architectures of information systems. Our investigation shows a remarkable difference between the straightforward static perspective, and the far-from straightforward dynamic perspective. This shows that approaches or methodologies for the development of information systems that do not systematically consider the dynamic perspective solve only a fraction of the real software engineering issues. This study has a number of contributions.

First, since previous studies on stability did not have a grounding in systems theory, we consider our application of systems theoretic stability in the context of software engineering to be a contribution. More specifically, we have shown how systems theoretic stability can assist in striving towards establishing an objective and scientific foundation to analyze the evolvability characteristics of information systems.

Second, we derived 4 design theorems that identify combinatorial effects. In themselves, these theorems are not new and are even consistent with known design principles in theory and practice. However, the way in which they can be derived from a single postulate based on systems theory shows how these superficially very different design theorems can be integrated. We consider this to be an important contribution since it illustrates that stability and combinatorial effects are very fundamental concepts implicitly underlying current software engineering knowledge. One of the goals of science is to derive, explain, and unify theorems from basic assumptions. Therefore, our attempt to derive and unify various software design principles that are currently presented as empirical and independent rules should be considered a worthy pursuit from a scientific point of view. Moreover, these theorems are formulated in a much more precise way than is traditionally the case.

Third, we have used a theoretical approach to expose the fundamental issues in software evolvability, while most other authors have used empirical methods. Our basic underlying motivation is that we should strive towards establishing an objective and scientific foundation to analyze the evolvability characteristics of information systems. More specifically, we should strive towards a theoretical framework to analyze, judge, and improve — from the point of view of evolvability — the various software languages, frameworks, interfaces, and ways of working of software engineers. We consider this an important contribution as it may assist in striving to turn software engineering into a classical engineering science that is based on laws and exhibits predictability. Our efforts can be considered a first initial step in this direction. The approach and design theorems described in this paper can therefore give impetus to a new line of empirical research that can test the hypotheses that can be derived from this approach.

The results of this paper are also relevant to software practitioners and software managers in particular. Managers are confronted with limited resources (e.g., in terms of cost, supply of software engineers, and time) to maintain information systems. Given these bounded resources, managers must be able to process a given set of change requests. Our definition of stability is based on the concept of BIBO stability as it is defined in systems theory and is very relevant in this regard. In systems theory, the aim is to achieve output functions that are bounded, limited, finite, predictable, and controllable. The problem of achieving limited and controllable output functions — i.e., limited amount of coding and therefore resources in our case — is tackled in systems theory by identifying the instabilities, the inputs and system interactions that could lead to an unbounded output. Precisely the identification of such instabilities is the key to achieving stable, bounded, limited, predictable, and controllable output functions. The approach in this paper therefore provides guidance to software managers on how stability can be approached, allowing themselves to make maximum use of the limited resources that are available.

Our study also has a number of limitations. First of all, it is important to note that we argue that we should strive towards realizing the ideas presented in this paper. Although the long-term goal should be to eliminate all combinatorial effects, and therefore all instabilities, this may be very difficult to achieve in practice. Similarly, it remains to be seen to which extent
systems and control theory can be applied to discipline and control software engineering efforts. Nevertheless, the approach described in this paper should help in identifying and in significantly reducing the number of instabilities which would already considerably improve the evolvability of information systems. Future research can provide more insight into these issues. Second, the ideas presented in this paper need to be further validated in empirical studies. One interesting approach would be the development of highly-structured design patterns. It is evident from the discussion in this paper that it will be very difficult to build stable information systems by using the standard constructs of programming languages. It would be useful if design patterns can be developed based on these constructs in a way that it can be proven that the resulting design patterns do not contain any instabilities. This would imply that applications that are built using these design patterns are also largely free of combinatorial effects. Future empirical studies could then be conducted in which experts perform a code review of these applications in order to verify the absence of the aforementioned instabilities.

References