Experiences with the Automatic Discovery of Violations to the Normalized Systems Design Theorems

Kris Ven, Dieter Van Nuffel, Philip Huysmans, David Bellens, Herwig Mannaert
Department of Management Information Systems
University of Antwerp
Prinsstraat 13
2000 Antwerp, Belgium
{*kris.ven,dieter.vannuffel,philip.huysmans,david.bellens,herwig.mannaert*}*@ua.ac.be*

***Abstract*—Evolvability is an important concern for the design and development of information systems. The Normalized Systems theory has recently been proposed and aims to ensure the high evolvability of information systems. The Normalized Systems theory is based on the systems theoretic concept of stability and proposes four design theorems that act as constraints on the modular structure of software. In this paper, we explore the feasibility of building a tool that is able to automatically identify violations to these Normalized Systems design theorems in the source code of applications. Such a tool could help developers in identifying limitations to the evolvability of their applications. We describe how a prototype of such a tool was developed and report on the evaluation of this tool consisting of the analysis of the source code of four open source software applications. Our results demonstrate that it is feasible to automatically identify violations to the Normalized Systems design theorems. In addition, the results show that there is considerable variety in how well the different theorems are adhered to by various software applications. We also identified some issues and limitations with the current version of the tool and discuss how these issues can be addressed in a future version.**

***Keywords*-normalized systems; modularity; software architecture; quality**

I. Introduction

Contemporary organizations are operating in increasingly volatile environments and must be able to respond quickly to changes in their environment in order to gain a competitive advantage [2], [3]. Since organizations are becoming increasingly dependent on information technology (IT) to support their operations, the evolvability of the IT infrastructure will determine to a large extent how quickly organizations are able to adapt. It has indeed been shown that IT offers opportunities to organizations to increase their agility and flexibility [4]–[6]. Organizations therefore require increasing levels of evolvability of their information systems. Unfortunately, information systems struggle to provide the requested levels of evolvability, often due to poorly designed software architectures [7].

The Normalized Systems theory has recently been proposed by Mannaert and Verelst [8] and aims to address these evolvability issues. The Normalized Systems theory is concerned with how information systems can be developed based on the systems theoretic concept of stability [8]–[10]. It argues that the main obstacle to evolvability is the existence of so-called *combinatorial effects*. Combinatorial effects occur when the effort to apply a specific change increases as the system grows [8], [10]. The Normalized Systems theory eliminates these combinatorial effects by defining clear design theorems. These Normalized Systems design theorems act as constraints on the modular structure of software. Adhering to these theorems results in information systems that exhibit stability.

Organizations currently have a large number of in-house developed information systems in use. These information systems are likely to contain combinatorial effects that limit their evolvability. These combinatorial effects exist due to violations to the Normalized Systems design theorems. Organizations will therefore be looking towards ways to identify these combinatorial effects in their code base and to devise solutions to improve the evolvability of their information systems. Manually inspecting the source code may be a possibility, but is likely to be a very time-consuming task. The automatic identification of combinatorial effects therefore seems to be a very interesting alternative. In this paper, we explore the feasibility of building a tool to automatically identify violations to the Normalized Systems design theorems in the source code of applications. Although our main focus—similar to our previous research [9], [10]—is on information systems, this tool could be used to perform an evaluation of any type of software application. In this paper, we describe the development and evaluation of a prototype of such a tool. In our previous work, we already described the evaluation of this prototype using a single case [1]. Our current work further builds on this research by analyzing the source code of four open source software applications.

The rest of this paper is structured as follows. In Section II, we describe the previous work related to this study and focus on providing an introduction to the Normalized Systems theory. The methodology of our study is described in Section III. Section IV describes the development of our

tool. The evaluation of the tool is described in Section V. The results from the evaluation are discussed in Section VI. Finally, our conclusions are offered in Section VII.

II. Previous Work

In this section, we provide an overview of the literature related to our current study. We start with an introduction to the topic of software evolvability. Next, we focus on providing a background on the Normalized Systems theory.

A. Software Evolvability

It is a well-known problem in software engineering that the structure of software degrades and becomes more complex over time as changes are applied to it. One of the main challenges with respect to the evolvability of information systems is Lehman's Law of Increasing Complexity which states that: "*As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.*" [11] 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—unless preventive measures are taken [11]–[13]. This is clearly an important concern for information systems development. There is widespread belief that the software architecture determines the evolvability of software to a large extent [14]. As a result, a number of frameworks have appeared in literature that attempt to evaluate software architectures based on a number of quality attributes, including evolvability [15], [16]. Some of the most well-known evaluation methods include the *Architecture Trade-off Analysis Method (ATAM)* [17] and the *Software Architecture Analysis Method (SAAM)* [18]. Unfortunately, it has been noted that a theoretical foundation for studying software evolvability and evolution is largely missing [19]. As a first step towards such a theoretical foundation, Lehman derived a list of definitions, theorems and axioms with respect to software evolution based on a large empirical research project spanning multiple years [19].

The approaches mentioned above typically define a set of principles that should ensure the evolvability of software systems. Unfortunately, some of these principles are defined rather informally and leave considerable room for interpretation. As a result, these approaches struggle to consistently achieve evolvability in realistic software development environments. This is frequently a consequence of the fact that it is difficult to reach consensus among practitioners about how a principle should exactly be applied in practice. For example, when asked to evaluate alternative designs for a software system based on a principle such as loose coupling, practitioners frequently disagree on the best solution. Several tools exist that calculate a set of metrics of a software system in order to provide an idea of the evolvability of the software system. However, such assessments require a white-box approach. A statement that the software is more or less evolvable based on such assessments therefore have a limited meaning. The Normalized Systems theory is similar to these previous approaches in taking evolvability as the primary concern for developing software systems. The main difference with these previous approaches is that the Normalized Systems theory is based on the systems theoretic concept of stability and aims to provide clear principles on software evolvability. Such clear principles avoid the situation in which developers or software architects disagree on the exact interpretation of a principle. By stating that a software system is compliant with the Normalized Systems theory, a more black box assessment of evolvability is therefore possible, since this defines to which anticipated changes the software is stable.

B. Normalized Systems

In this section, we will provide a brief background on the Normalized Systems theory. However, the aim of this section is not to fully explain Normalized Systems, or to elaborate on the theorems and their rationale. Instead, we further build upon the previous work that is available in this area. For more details, we refer the reader to our previous work describing the Normalized Systems theory [8]–[10], [20]–[22].

The basic assumption of the Normalized Systems approach is that information systems should be able to evolve over time and should therefore be designed to accommodate change. This implies that the software architecture should not only satisfy the current requirements, but should also support future requirements. The Normalized Systems approach uses the systems theoretic concept of *stability* as the basis for developing information systems [8]–[10], [20]. In systems theory, stability refers to a system in which a bounded input function results in bounded output values, even as $t \rightarrow \infty$ (with t representing time). When applied to information systems, this means that applying a specific change to the information system should always require the same effort, irrespective of the size of the information system or the point in time at which the change is applied. The Normalized Systems approach further relies on the *assumption of unlimited systems evolution* [8]–[10]. This means that the system becomes ever larger in the sense that the number of modules become infinite or unbounded as $t \rightarrow \infty$. This may seem an overstated assumption, but actually, it is quite logical as even the introduction of a single module or dependency every twenty years corresponds to an infinite amount for an infinite time period.

Information systems exhibiting stability with respect to a defined set of changes are called *Normalized Systems* [8], [10]. In contrast, when changes do require increasing effort as the system grows, *combinatorial effects* are said to occur [8], [10]. In order to obtain stable information systems, these combinatorial effects should be eliminated. In order to

identify and avoid most of these combinatorial effects, a set of four *design theorems* was developed [8]–[10], [20]. It is important to note that it has been formally proven that these theorems contribute to achieving systems theoretic stability in software [9]. We will now briefly describe each of these theorems. More details are beyond the scope of this paper and can be found in the literature [8]–[10], [20].

The first theorem, *separation of concerns*, requires that every change driver or concern is separated from other concerns. This theorem allows for the isolation of the impact of each change driver. This principle was informally described by Parnas already in 1972 as what was later called *design for change* [23]. This theorem implies that each module can contain only one submodular task (which is defined as a change driver), but also that workflows should be separated from functional submodular tasks. For instance, consider a function F consisting of task A with a single version and a second task B with N versions; thus leading to N versions of function F. The introduction of a mandatory version upgrade of task A will not only require the creation of the additional task version of A, but also the insertion of this new version in the N existing versions of function F. The number N is clearly dependent on the size of the system, and thus implies a combinatorial effect.

The second theorem, *data version transparency*, requires that data is communicated in version transparent ways between components. This requires that this data can be changed (e.g., additional data can be sent between components), without having an impact on the components and their interfaces. For instance, consider a data structure D that is passed to N versions of a function F. If an update of the data structure is not version transparent, it will also demand the adaptation of the code that accesses this data structure. Therefore, it will require new versions of the N existing processing functions F. The number N is clearly dependent on the size of the system, and thus implies a combinatorial effect. Data version transparency can, for example, be accomplished by appropriate and systematic use of web services instead of using binary transfer of parameters. This also implies that most external APIs cannot be used directly, since they use an enumeration of primitive data types in their interface.

The third theorem, *action version transparency*, requires that a component can be upgraded without impacting the calling components. Consider, for instance, a processing function P that is called by N other processing functions F. If a version upgrade of the processing function P is not version transparent, this will cause besides upgrading P, the adaptation of the code that calls P in the various functions F. Therefore, it will require new versions of the N existing processing functions F. The number N is clearly dependent on the size of the system, and thus implies a combinatorial effect. Action version transparency can be accomplished by appropriate and systematic use of, for example, polymorphism or a facade pattern.

The fourth theorem, *separation of states*, requires that actions or steps in a workflow are separated from each other in time by keeping state after every action or step. For instance, consider a processing function P that is called by N other processing functions F. Suppose the calling of the function P does not exhibit state keeping. The introduction of a new version of P, possibly with a new error state, would force the N functions F to handle this error, and would therefore lead to N distinct code changes. The number N is clearly dependent on the size of the system, and thus implies a combinatorial effect. This theorem suggests an asynchronous and stateful way of calling other components. Synchronous calls—resulting in pipelines of objects calling other objects that are typical for object-oriented development—result in combinatorial effects.

It must be noted that each of these theorems is not completely new, and even relates to the heuristic knowledge of developers. However, formulating this knowledge as theorems that identify these combinatorial effects aids to build information systems that contain a minimal number of combinatorial effects. A remarkable aspect of these theorems is that a violation of each one of these theorems, by any developer at any moment during development or maintenance, results in a combinatorial effect. This suggests how difficult it is to realize software without combinatorial effects [8], [10].

The design theorems show that software constructs, such as functions and classes, by themselves offer no mechanisms to accommodate anticipated changes in a stable manner [8], [10]. The Normalized Systems approach therefore proposes to encapsulate software constructs in a set of five higher-level software elements [8], [10]. These elements are modular structures that adhere to these design theorems, in order to provide the required stability with respect to the anticipated changes [8], [10]. From the second and third theorem it can straightforwardly be deduced that the basic software constructs, i.e., data and actions, have to be encapsulated in their designated construct. As such, a *data element* represents an encapsulated data construct with its get- and set-methods to provide access to its information in a data version transparent way. So-called cross-cutting concerns, for instance access control and persistency, should be added to the element in separate constructs. The second element, *action element*, contains a core action representing a single functional task or change driver. Four different implementations of an action element can be distinguished: *standard* actions, *manual* actions, *bridge* actions and *external* actions [24]. In a standard action, the actual task is programmed in the action element and performed by the same information system. In a manual action, a human act is required to fulfill the task. The user then has to set the state of the life cycle data element through a user interface after the completion of the task. A process step can also require more complex

behavior. A single task in a workflow can be required to take care of other aspects which are not the concern of that particular flow [8], [10]. Therefore, a separate workflow will be created to handle these concerns. Bridge actions create these other data elements going through their designated flow. When an existing, external application is already in use to perform the required task, the action element would be implemented as an external action. These actions call other information systems and set their end state depending on the external systems' reported answer. Arguments and parameters of an action element need to be encapsulated as separate data elements, and cross-cutting concerns such as logging and remote access should be added as separate constructs. Based upon the first and fourth theorem, workflow has to be separated from other action elements [8], [10]. These action elements must be isolated by intermediate states, and information systems have to react to states. To enable these requirements, three additional elements are identified. A third element is a *workflow element* containing the sequence in which a number of action elements should be executed in order to fulfill a flow. A consequence of the stateful workflow elements is that state is required for every instance of use of an action element, and that the state therefore needs to be linked to or be part of the instance of the data element serving as argument. A *trigger element* is a fourth element that controls the states (both regular and error states) and checks whether an action element has to be triggered. Finally, the *connector element* ensures that external systems can interact with data elements without allowing an action element to be called in a stateless way [8], [10].

It is important to note that the basic underlying motivation of the Normalized Systems theory is to strive towards establishing an objective and scientific foundation to analyze the evolvability characteristics of information systems. The previous studies on this topic can be considered a first initial step towards turning software engineering into a classical engineering science that is based on laws and exhibits predictability [9].

III. Methodology

Our research has been conducted using the design science methodology. Our research goal is to develop a tool for the automatic identification of violations to the Normalized Systems design theorems in the source code of information systems. The design science methodology is appropriate in this case, since design science is primarily aimed at *solving* problems by developing and testing *artifacts*, rather than *explaining* them by developing and testing *theoretical hypotheses*. The design science research tradition focuses on tackling ill-structured problems in a systematic way [25]. Peffers et al. consider information systems to be an applied research discipline, meaning that theory from disciplines such as economics, computer science and social sciences are frequently used to solve problems between information technology and organizations [26]. In this research, we will use the Normalized Systems theory as the basis to develop a tool to identify potential issues with respect to the evolvability of software. Hence, we start from a solid theoretical foundation to develop a tool that has a large potential to be used in practice.

March and Smith have developed a classification scheme to position design science research efforts. This scheme identifies 4 different research outputs (i.e., construct, model, method and instantiation) and 4 different research activities (i.e., build, evaluate, theorize and justify) [27]. Our research is concerned with the *build* and *evaluate* phases of an *instantiation* artifact. The instantiation refers in this case to a tool to identify violations to the Normalized Systems design theorems. If such a tool could be developed, it would illustrate the feasibility of the automatic identification of violations. The importance of building instantiations has been emphasized by Newell and Simon, by writing: "*Each new program that is built is an experiment. It poses a question to nature, and its behavior offers clues to the answer*" [28].

Consistent with the design science methodology, an iterative approach will be followed in this research [26], [29], [30]. We started by first defining and motivating the problem based on the literature on Normalized Systems. Therefore, the research entry point is objective-centered, and is concerned with developing a tool to identify violations to the Normalized Systems design theorems [26]. Based on the Normalized Systems design theorems, we derived a number of violations that may occur in Java applications. In this first iteration, it is not our aim to create an exhaustive list of potential violations. Hence, we provide a lower bound for the existence of such violations in information systems. This constitutes a contribution towards the Normalized Systems approach, since this provides insight into which concrete violations to the Normalized Systems design theorems can be found in practice. Next, we investigate the feasibility of building a tool that can automatically identify manifestations of these violations in the source code of information systems.

Finally, we conduct a first evaluation of the tool. Evaluation is considered to be a key element in the design process [31]. To this end, we evaluate the tool by applying it to a set of Java applications, interpreting the resulting output and verifying the violations in the source code. The correctly identified violations confirm the utility of this tool. This first version of the tool is an important milestone, as it will give valuable feedback on the feasibility of the automatic inspection of the source code with respect to violations to the Normalized Systems design theorems. Furthermore, we will use the lessons learned from this first evaluation to improve the efficiency of our tool. In the following iterations, we will further develop and refine our tool. Future improvements include, for example, detecting a larger number of violations to the Normalized Systems design theorems. These future

versions will be evaluated using other applications as a test case. We seek to further evaluate the tool in the future by applying it to larger and more complex applications.

IV. Tool Development

In the *build* phase of our research, we iteratively developed a tool prototype for the automatic identification of violations to the Normalized Systems design theorems. Before developing the tool, we first needed to determine for which programming language we wanted to build the tool. Since each programming language has its own constructs and syntax, different violations are possible in different programming languages. Therefore, separate parsers should be developed for each programming language. We decided to focus on the Java programming language. This choice was motivated by a number of reasons. First, Java is a popular programming language that is used by a large number of applications, including traditional GUI applications and web-based applications. Second, Java EE is a popular framework to build enterprise applications, making it very relevant in an organizational context. Third, the reference implementation for Normalized Systems was also built in the Java EE environment [8], [10].

A graphical overview of the architecture of this tool is shown in Figure 1. It shows that the tool consists of two main components—`NSTVdoclet` and `NSTVdetect`—that have to be run in succession. The former component is responsible for parsing the source code, while the latter component is responsible for actually analyzing the source code for manifestations of violations to the Normalized Systems design theorems. This approach allowed us to decouple the parsing of the source code and the actual inspection of the source code, which is consistent with the separation of concerns theorem.

As shown in Figure 1, the first step of the analysis consists of processing the Java source code by the `NSTVdoclet` component. The `NSTVdoclet` component is written as a custom doclet to `javadoc`. The `javadoc` tool is part of the Java 2 SDK. By default, it generates documentation in HTML format of the API of a Java application. The `javadoc` tool is, however, easy to extend by creating custom doclets that provide output in an alternate format. The `NSTVdoclet` component filters the information obtained by `javadoc` since not all this information is required by `NSTVdetect`. Next, the output is written away in a temporary database. The information contained in this database is an internal representation of the source code that is to a large extent independent on a specific programming language (e.g., in terms of classes that have methods that take parameters of a certain type and that possibly throw an exception). This method has three main advantages. First, it allows us to reuse the source code parsing algorithm of `javadoc`. This avoids having to write a custom Java source code parser. In addition, the output provided by `javadoc` is clearly documented at the API-level, making it easy to parse and process this information. Second, most Java applications ship with an `ant` build file that allows the automatic compilation of Java source code. In most cases, this `ant` build file includes a `javadocs` target that generates the API-documentation for the application using `javadoc`. If such a build target is available, it is quite easy to specify in the build file that a custom `javadoc` doclet must be used. This ensures that parsing the source code does not require much effort, on the condition that the standard `javadoc` documentation can be generated. In general, it is sufficient to modify the `javadoc` task to indicate that a custom doclet that must be used by specifying the `doclet` and `docletpath` attributes (see also Figure 2).

In the second step of the analysis, the `NSTVdetect` component processes the information in this database and analyzes it to identify manifestations of violations to the Normalized Systems design theorems. Consistent with the separation of concerns theorem, the `NSTVdetect` component delegates the responsibility of the actual detection of these manifestations of violations to an extensible set of modules. Each module analyzes the internal representation of the source code for manifestations of a specific violation. Each module writes its output to a separate report file.

An $M-N$ relationship exists between these violations and the Normalized Systems design theorems: a single design theorem can be violated in several ways, while a single violation can refer to more than one theorem. Our tool currently supports the detection of manifestations of three violations that may occur in Java applications. The identification of these violations is based upon—and consistent with—previous work [8]–[10]. Although the current list of violations is not exhaustive, it includes common violations against the Normalized Systems design theorems and covers all four design theorems. This list can be further expanded in the future. As such, the current list represents a lower bound of the violations to the Normalized Systems design theorems that exist in Java applications. We will now discuss these violations and how they are detected by each module in more detail.

A. Import Multiple Concerns Violation

A first violation occurs when a class combines more than one concern by using the `import` statements in Java. Such a class violates the separation of concerns theorem and therefore results in combinatorial effects. Java classes can import and use functionality from external technology environments and packages by using the `import` instruction. This may introduce dependencies on these external technologies in an implicit way since each of these technologies can change independently in the future. Consider a specific concern that is combined with one or more other concerns in N different classes. If this concern changes in the future (e.g., when it is decided to use an alternative external technology),

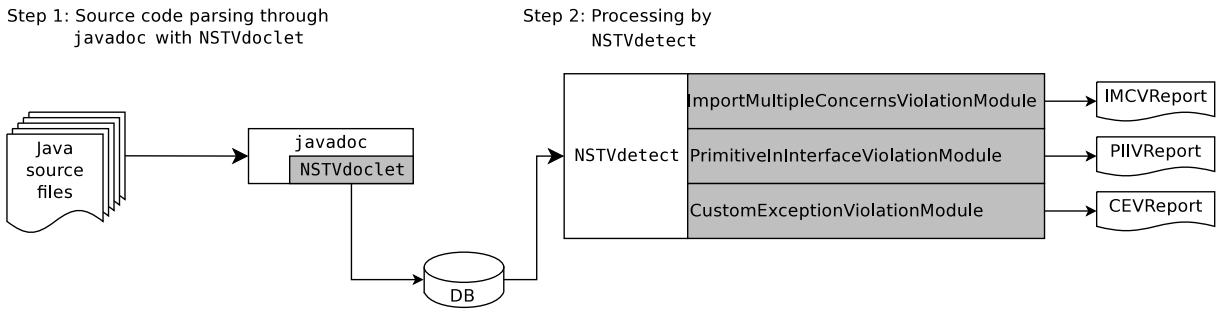

Figure 1. Tool Architecture

Original `javadoc` task for iText:

```
<javadoc
   destdir="${itext.docs}"
   author="true" maxmemory="128m"
   private="true">
```

Modified `javadoc` task for iText:

```
<javadoc
   destdir="${itext.docs}"
   author="true" maxmemory="128m"
   private="true"
   doclet="ua.mis.NSTVdoclet.main.NSTVdoclet"
   docletpath="nstvdoclet.jar">
```

Figure 2. Example of modification of ant build file

then this change has an impact on N different classes. Since N becomes unbounded over time, the impact of this change will increase over time as well, thereby resulting in a combinatorial effect. The separation of concerns design theorem requires that each change driver or concern is isolated from other concerns, so that each concern can evolve independently. This implies that each module should contain only one change driver [8]–[10]. As a result, a class should not combine two or more concerns.

The *Import Multiple Concerns Violation* module determines which concerns are used by each class based on the imported libraries (using the `import` statements in Java). We consider that the `import` statements in Java may provide a rough, but useful indication of which external technologies are used by a specific class, and therefore which concerns are addressed in the class. Before running the analysis, the researcher must define which concerns are present in the application, as well as which libraries fall under each concern. Depending on the application, one concern could, for example, be the use of the Java Swing packages for the graphical user interface, while a second concern could be the use of the Java JDBC packages to support database access. According to the separation of concerns theorem, both concerns should not be combined in a single class. This is consistent with the concept of multi-tier architectures. Another concern could be the use of another application, such as Cocoon to provide a web-based user interface. Based on this definition of concerns, it is determined how many different concerns are combined in each class. The researcher must define these concerns with care to ensure that the libraries correspond to the various concerns in the application as much as possible, in order to minimize the number of false positives identified by this module.

B. Primitive in Interface Violation

The second violation occurs when the interface of a method contains a primitive data type or a class of the type `java.lang.String`. Such a method violates both the data version transparency and action version transparency theorem and therefore results in combinatorial effects. Consider a method that is called by N other methods in the application and that contains one or more primitive data types or the `java.lang.String` class in its interface. If the functionality of this method is extended in the future, this extra functionality may require additional information to be sent to the method. However, the data that is sent to this method is not data version transparent. Since primitive data types or the `java.lang.String` class can only contain single values, it is not possible to send additional information without having an impact on this data structure. The method is not action version transparent either since the interface of the method will need to change to accept this additional information. It is therefore not possible to upgrade to a new version of the method without having an impact on the rest of the system. Hence, if additional data is needed by this method, this will have an impact on the N methods that call this method. In each of these N methods, the additional data needs to be initialized and the method call has to be adapted according to the modified interface of the method. Since N becomes unbounded over time, the impact of this change will increase over time as well, thereby resulting in a combinatorial effect. To resolve this issue, it is better to encapsulate the method parameters in a dedicated object with a default constructor. This constructor assigns neutral values to each of the parameters, which can be overwritten by calling the appropriate set methods. Future changes would

then have no effect on methods calling the method [8]–[10].

We further illustrate this principle with a practical example. Consider a method that allows the user to search for a specific string in a set of files and that takes a single parameter of the type `java.lang.String` that specifies the text to search for. Next, assume that the developers want to extend the search functionality in the future by allowing users to make use of regular expressions. In other words, the method should support searching based on normal text as well as regular expressions. In that case, the interface of the method should be extended with a `boolean` variable to indicate whether the string is a regular expression. This will, however, affect all other methods calling the search method. The data is not data version transparent since its structure does not allow to send additional data without any additional changes to the system. In addition, the method is not action version transparent since the interface of the method will change, and it is not possible to upgrade to a new version of the method without having an impact on the rest of the system. One could argue that overloading could be used in this case, by having one method with as interface `(java.lang.String)` and another method with as interface `(java.lang.String, boolean)`. However, this solution also has limitations with respect to evolvability since this approach is only feasible when the unique interface of the method can be guaranteed. If the same method would be extended with the functionality to make the search case-sensitive or not, another method with as interface `(java.lang.String, boolean)` would need to be added. However, this is impossible, since a method with this signature has already been defined. To resolve this issue, it is better to encapsulate the search parameters in a new class `SearchConfiguration` that can be extended with additional fields as new functionality is added to the search method. The default constructor of the `SearchConfiguration` object should assign a neutral value to newly added parameters (e.g., to indicate that a search string is not a regular expression). By calling the appropriate `set` method, the default settings can be overwritten. Future changes would then have no effect on methods calling the search method. This solution would be compliant with the data and action version transparency theorems.

The *Primitive in Interface Violation* module therefore inspects the interface of each non-private method and determines whether the interface includes one or more primitive data types or the `java.lang.String` class.

C. Custom Exception Violation

The third violation occurs when a method throws a custom exception (i.e., an exception that is not part of the default Java environment). The Java programming language provides the exception mechanism to handle errors that occur during the execution of a method. If an exception is thrown by a method, the calling method must process this error, either by catching and handling the error internally, or by throwing the exception further upward the stack. This constitutes a violation to the separation of states theorem and therefore results in combinatorial effects. Consider a method that is called by N different methods in the application. If the developer working on this method decides to introduce a new error state by having the method to throw a new exception, then this has an impact on the N methods that call this method, since they are forced by the Java environment to either catch or throw this exception further upward the stack. Hence, the error handling takes place in N different places. Since N becomes unbounded over time, the impact of this change will increase over time, thereby resulting in a combinatorial effect. Instead, the error state should be stored and error handling should be performed by a separate and dedicated module [8]–[10].

The *Custom Exception Violation* module therefore determines how many custom exceptions are thrown by all methods. We consider the use of standard Java exceptions (e.g., `java.lang.Exception` and `java.io.IOException`) to be acceptable, since they are related to the background technology being used. Even in this case, the use of these exceptions should be kept to a minimum. The use of custom exceptions should be avoided since such errors should be handled in a stateful way.

V. Case Study

We now discuss the *evaluation* phase of the design research process. In order to evaluate our tool, we analyzed the source code of a number of Java applications. These applications needed to satisfy three criteria. First, we focused on Java applications that are distributed under an open source license since this provides us with free access to the source code of these application. Second, the applications should represent a moderate development effort. Applications should not be too small to be disregarded as a toy example, but should also not be too large and too complex to complicate the evaluation of our tool. Third, we preferred to select applications that are quite popular and widely adopted to use applications that are used in real-life settings, rather than laboratory applications.

Based on these criteria, we selected four applications: (1) *Apache Lucene*, a fully-fledged text search engine; (2) *jEdit*, a programmers' text editor that supports a large number of programming languages; (3) *JabRef*, a bibliography reference manager that is used to edit BibTeX files; and (4) *iText*, a library that can be used by applications to facilitate the creation and manipulation of PDF documents. Details on the source code of these four open source software products can be found in Table I.

It must be noted that these four programs represent rather simple applications since they are written from scratch in Java, use a limited number of external libraries, and are

Table I
DETAILS OF SELECTED OPEN SOURCE SOFTWARE PRODUCTS

Name	Version	Classes	Methods	LOC
Apache Lucene	3.0.0	367	2,744	81,290
jEdit	4.3.1	477	1,980	163,015
JabRef	2.5	980	5,988	98,982
iText	5.0.5	583	5,663	140,883

not based on advanced frameworks. As a result, they do not exhibit the complexity of contemporary information systems, on which the Normalized Systems theory focuses. However, our aim in this paper is on exploring the feasibility of building a tool that is able to automatically identify violations to these Normalized Systems design theorems in the source code of applications. Our previous research has shown that even small applications are likely to contain several violations to the Normalized Systems theorems [1]. Therefore, applying our tool to analyze a complex information system is likely to result in a very large set of violations. Interpreting these results would complicate the evaluation of our tool. As a result, the four applications selected above are suitable for our purpose.

It is important to note that we do not want to make any claims with respect to the quality of the four applications in our case study. Instead, we want to perform an evaluation of our tool and its ability to automatically identify manifestations of violations to the Normalized Systems design theorems.

A. Import Multiple Concerns Violations

As mentioned in Section IV-A, we must first specify which concerns are present in a given application before running the analysis with `NSTVdetect`. To this end, we developed a shell script to extract a list of the unique package names that were imported by all classes of a given application from the temporary database created by `NSTVdoclet`. The task of the researcher is then to group these import statements in a set of concerns. The concerns that were identified for each of the four applications in our case study—as well as the packages that relate to each concern—are displayed in Table II. In this analysis, all packages belonging to the application itself were not considered to be a separate concern. In the case of jEdit, for example, all packages belonging to the `org.gjt.sp.jedit.*` package were considered part of the application itself, and not a separate concern.

A summary of the output from the *Import Multiple Concerns Violations* module is shown in Table III. According to the separation of concerns theorem, a class should not address more than one concern. The results from our analysis show a relatively low to moderate number of manifestations of this violation. The percentage of classes that are compliant with the separation of concerns theorem is (in decreasing order) 98.9% (363 out of 367 classes) for Lucene, 85.6% (499 out of 583 classes) for iText, 82.8% (395 out of 477 classes) for jEdit, and 63.9% (626 out of 980 classes) for JabRef. Lucene therefore performs very well, although it must be noted that only three concerns were identified for this application. JabRef has the highest number of violations with 36.1% of its classes, but also has the largest number of concerns. It therefore appears that there may be a relationship between the number of concerns that are identified for an application and the number of violations to the separation of concerns theorem. Overall, we can conclude that the separation of concerns theorem is rather to very well adhered to in all four applications.

A more detailed analysis showed that in those classes in which more than two concerns are combined, the Java IO concern is frequently combined with other concerns, such as Java Swing (e.g., JabRef and jEdit) or Java Net (JabRef and iText). Although most of these concerns are related to the default Java SDK API, it does create dependencies on different packages within the API. This data also suggests that file system functions (Java IO and Java Net) are combined with user interface functions (Java Swing). This may neglect the concept of multi-tiers and would therefore require attention in a further screening of the source code.

B. Primitive in Interface Violation

A summary of the output of the *Primitive in Interface* module is shown in Table IV. It can be seen that the percentage of methods that do not contain any manifestations of this violation and that do not contain any primitive data types or the `java.lang.String` class in their interface is (in decreasing order) 69.6% (4170 out of 5988) for JabRef, 61.7% (1693 out of 2744) for Lucene, 57.1% (1130 out of 1980) for jEdit, and 55.6% (3146 out of 5663) for iText. However, these percentages were calculated by including those methods that do not take any parameters and therefore require no input. If we exclude those methods from our analysis, the percentage of valid methods is 46.7% (1592 out of 3410) for JabRef, 36.0% (591 out of 1642) for Lucene, 31.8% (1172 out of 3689) for iText, and 30.6% (374 out of 1224) for jEdit. As could be expected, this lowers the proportion of valid methods considerably. Since these numbers are rather small, it can be concluded that the data and action version transparency theorems are not well adhered to in all four products.

C. Custom Exception Violation

A summary of the output of the *Custom Exception Violation* module is shown in Table V. As already mentioned in Section IV-C, we considered the use of standard Java exceptions to be acceptable, since they represent the background technology being used. This means that methods throwing `java.lang.Exception` and `java.io.IOException` exceptions were not considered a violation. The results show that the percentage of methods

Table II
LIST OF CONCERNS IDENTIFIED IN THE SELECTED APPLICATIONS

Application	Concern	Description
Lucene	Java IO	`java.io.*`
	Java Net	`java.net.*`
	Java Security	`java.security.*`
jEdit	Java Swing	`java.awt.*`, `javax.swing.*`
	Java Beans	`java.beans.*`
	Java IO	`java.io.*`, `java.nio.*`
	Java Net	`java.net.*`
	Java Security	`java.security.*`
	Java XML	`org.xml.sax.*`
	Microstar	`com.microstar.*`
JabRef	Java Swing	`java.awt.*`, `javax.swing.*`
	Java Beans	`java.beans.*`
	Java IO	`java.io.*`, `java.nio.*`
	Java Net	`java.net.*`
	Java SQL	`java.sql.*`
	Java XML	`javax.xml.*`, `org.w3c.dom.*`, `org.xml.sax.*`
	Java Plugin	`org.java.plugin.*`
	antlr	`antlr.*`, `org.antlr.*`
	glazedlists	`ca.odell.glazedlists.*`
	jgoodies	`com.jgoodies.*`
	ritopt	`gnu.dtools.ritopt.*`
	microba	`com.michaelbaranov.microba.*`
	jempbox	`org.jempbox.*`
	pdfbox	`org.pdfbox.*`
iText	Java Swing	`java.awt.*`, `javax.swing.*`
	Java IO	`java.io.*`, `java.nio.*`
	Java Net	`java.net.*`
	Java Security	`java.security.*`
	Bouncy Castle	`org.bouncycastle.*`
	dom4j	`org.dom4j.*`, `org.w3c.dom.*`, `org.xml.sax.*`

Table III
IMPORT MULTIPLE CONCERNS VIOLATIONS

	Number of		
Application	Concerns	Classes	Percentage
Lucene	0	110	30.0%
	1	253	68.9%
	2	4	1.1%
	Total:	*367*	*100.0%*
jEdit	0	174	36.5%
	1	221	46.3%
	2	59	12.4%
	3	12	2.5%
	4	11	2.3%
	Total:	*477*	*100.0%*
JabRef	0	306	31.2%
	1	320	32.7%
	2	228	23.3%
	3	86	8.8%
	4	34	3.5%
	5	6	0.6%
	Total:	*980*	*100.0%*
iText	0	273	46.8%
	1	226	38.8%
	2	74	12.7%
	3	7	1.2%
	4	3	0.5%
	Total:	*583*	*100.0%*

that do not throw any custom exceptions is (in decreasing order) 97.3% (5828 out of 5988) for JabRef, 95.4% (5401 out of 5663) for iText, 94.8% (2600 out of 2744) for Lucene, and 92.1% (1823 out of 1980) for jEdit. Interestingly, if we only consider those methods that throw at least one exception, it shows that the percentage of valid methods is 81.2% (621 out of 765) for Lucene, 69.5% (364 out of 524) for JabRef, 61.2% (414 out of 676) for iText, and 32.9% (77 out of 234) for jEdit. As could be expected, this lowers the percentage of valid methods. This decrease is most notable for jEdit which appears to make quite extensive use of custom exceptions. Overall, a rather mixed image therefore emerges with respect to the adherence to the separation of states theorem.

VI. DISCUSSION

Although the tool to automatically detect violations to the Normalized Systems design theorems is still a prototype, our evaluation has shown that there is much potential for such automated analysis. Compared to our original study [1], we evaluated our tool by analyzing the source code of four applications, instead of a single application. This provides more trust in the fact that the tool can be applied to a large set of software programs. Our evaluation has also shown that our `NSTVdoclet` tool can be easily integrated with `javadoc`, and offers sufficient information for identifying manifestations of violations to the Normalized Systems design theorems. Much information about the structure of software can already be derived from the API information obtained by `javadoc`. Focusing on the API-level has

Table IV
PRIMITIVE IN INTERFACE VIOLATIONS

Application	Violations[a] per Method	All methods		Methods with parameters	
		Method Count	Percentage	Method Count	Percentage
Lucene	0	1693	61.7%	591	36.0%
	1	719	26.2%	719	43.8%
	2	167	6.1%	167	10.2%
	3	103	3.8%	103	6.3%
	4	39	1.4%	39	2.4%
	5	12	0.4%	12	0.7%
	6	3	0.1%	3	0.2%
	7	7	0.3%	7	0.4%
	8	1	0.0%	1	0.1%
	Total:	*2744*	*100.0%*	*1642*	*100.0%*
jEdit	0	1130	57.1%	374	30.6%
	1	535	27.0%	535	43.7%
	2	187	9.4%	187	15.3%
	3	60	3.0%	60	4.9%
	4	50	2.5%	50	4.1%
	5	9	0.5%	9	0.7%
	6	7	0.4%	7	0.6%
	7	2	0.1%	2	0.2%
	Total:	*1980*	*100.0%*	*1224*	*100.0%*
JabRef	0	4170	69.6%	1592	46.7%
	1	1334	22.3%	1334	39.1%
	2	294	4.9%	294	8.6%
	3	132	2.2%	132	3.9%
	4	34	0.6%	34	1.0%
	5	18	0.3%	18	0.5%
	6	6	0.1%	6	0.2%
	Total:	*5988*	*100.0%*	*3410*	*100.0%*
iText	0	3146	55.6%	1172	31.8%
	1	1622	28.6%	1622	44.0%
	2	452	8.0%	452	12.3%
	3	157	2.8%	157	4.3%
	4	150	2.6%	150	4.1%
	5	52	0.9%	52	1.4%
	6	56	1.0%	56	1.5%
	7	14	0.2%	14	0.4%
	8	11	0.2%	11	0.3%
	9	3	0.1%	3	0.1%
	Total:	*5663*	*100.0%*	*3689*	*100.0%*

[a] Number of primitive and `java.lang.String` data types used in interface

several advantages. First, the Normalized Systems approach is concerned with the modular structure of software. Hence, inspecting the structure of classes and the interface of methods is consistent with this view. Second, the API-level represents a medium-level view on the modular structure of software. The package level can considered to be too high-level, as much information is abstracted away on this level. Conversely, considering the actual source code level may be too low-level.

By applying our tool to four different open source software applications, we were also able to determine how these applications differ in their adherence to the Normalized Systems design theorems. The results show that there is considerable variety in how well the different theorems are adhered to. Our data showed that the separation of concerns theorem—a well-accepted principle by practitioners—was rather well to very well adhered to by all four applications. The data and action version transparency theorems were, however, not well adhered to by the four applications since many methods made use of primitive data types in their interface. A rather mixed view was present with the separation of states theorem, where some applications made relatively little use of custom exceptions (e.g., Lucene), while other applications made rather intensive use of them (e.g., jEdit). Such violations are not fatal, but identify potential sources for combinatorial effects that limit the evolvability of the software. Given some limitations of the tool, we do not intend our results to be an assessment of the evolvability of the four applications. Instead, the aforementioned analysis was meant to be an evaluation of the tool.

The identification of violations by our tool was based on the Normalized Systems theory. The Normalized Systems theory states that in order to guarantee evolvability, all combinatorial effects must be eliminated from the source code of

Table V
CUSTOM EXCEPTION VIOLATIONS

Application	Violations[a] per Method	All methods: Method Count	All methods: Percentage	Methods with exceptions: Method Count	Methods with exceptions: Percentage
Lucene	0	2600	94.8%	621	81.2%
	1	134	4.9%	134	17.5%
	2	5	0.2%	5	0.7%
	3	5	0.2%	5	0.7%
	Total:	*2744*	*100.0%*	*765*	*100.0%*
jEdit	0	1823	92.1%	77	32.9%
	1	154	7.8%	154	65.8%
	2	3	0.2%	3	1.3%
	Total:	*1980*	*100.0%*	*234*	*100.0%*
JabRef	0	5828	97.3%	364	69.5%
	1	124	2.1%	124	23.7%
	2	16	0.3%	16	3.1%
	3	20	0.3%	20	3.8%
	Total:	*5988*	*100.0%*	*524*	*100.0%*
iText	0	5401	95.4%	414	61.2%
	1	248	4.4%	248	36.7%
	2	13	0.2%	13	1.9%
	3	1	0.0%	1	0.1%
	Total:	*5663*	*100.0%*	*676*	*100.0%*

[a] Number of custom exceptions thrown

applications. To realize this, the Normalized Systems theory posits four theorems that must be adhered to. The violations detected by our tool are based on these four theorems. As a result, the violations identified by our tool represent potential issues with respect to the evolvability of the application. The seriousness of these violations depends on the changes that will be applied to the software in the future. If a violation is present in a certain part of the application that will not change in the future, then the violation will have no impact on the evolvability of the software. For instance, when an interface of a method contains a primitive data type, but the interface will remain constant over time, then this violation does not impact the evolvability of the software. However, when the interface does change, it will require a modification in all parts of the software that call this method. In that case, a combinatorial effect is present that impacts the evolvability of the software. For a detailed discussion of the impact of a violation to the Normalized Systems theory, we refer the reader to earlier work [9], [10].

Given the considerable amount—and nature—of violations to the Normalized Systems theory identified in our case study, it seems likely that it would require much work to resolve these issues. Given the limited availability of resources, it is very unlikely that resolving all violations is feasible. Instead, developers could use the output of this tool to identify parts in the application that require specific attention. They could then attempt to normalize specific parts of the application, so that these parts in themselves become stable for the future. Within each part, however, combinatorial effects would still be allowed and not all violations would be addressed. These results therefore provide further empirical support for the statement that building information systems without combinatorial effects is extremely difficult, and that constructs of traditional programming languages offer no protection against violating the Normalized Systems theorems [10]. It seems unlikely that it is feasible to fully normalize an existing application given the limitations in time and budget available in practice. However, the Normalized Systems approach further provides a set of five software elements that are proven to be free of combinatorial effects and that can be used a building blocks for new applications [8], [10]. With those elements, it is possible to build applications that are largely free of combinatorial effects. As illustrated in previous work, a set of seven complex real-life applications have been developed in the process of refining the Normalized Systems theory [10]. In addition, independent applications that are compliant with the Normalized Systems theory are currently being built by several external organizations in Belgium and The Netherlands.

A. Lessons Learned

Based on the case study, several lessons can be learned about the feasibility of automatically detecting violations to the Normalized Systems design theorems. These lessons may be useful for future versions of the tool.

First, by separating the parsing of the source code by `NSTVdoclet` and the analysis by the `NSTVdetect` tool in two components, it may be relatively easy to add support for additional programming languages in the future. Evidently, other programming languages would require a different implementation of the `NSTVdoclet` component to parse the source code. The output of this component is currently largely language-independent: information about the source code is stored in terms of classes, methods,

and parameters. These concepts apply to all object-oriented programming languages. The only concept that is not supported by all object-oriented programming languages is the Java exception. Hence, the database format should provide the possibility to support language-specific extensions. The `NSTVdetect` tool can be extended with additional modules that provide support for other programming languages. Some existing modules, such as the *Primitive in Interface Violation* module also applies for programming languages such as C++ or C#. Other modules may only apply to a specific set of programming languages. In that case, each module must specify to which programming language(s) it applies, so that the `NSTVdetect` component can decide whether the module should be invoked when performing a specific analysis.

Second, the output obtained by the tool provides us with feedback on the current implementation of the modules that test for manifestations of violations to the Normalized Systems design theorems. Concerning the *Import Multiple Concerns Violation* module, we have observed that applications frequently combined the Java Swing concern with the Java IO and/or Java Net concerns. This may suggest that input/output instructions are combined with the user interface. However, a closer inspection of the source code showed that this was a false positive. For example, in order to provide icons in toolbars in the user interface, Java Swing provides the `ImageIcon` class. It is common to initialize a new object of this class by using the constructor taking a `java.net.URL` object as parameter. This, for example, allows the icon file to be part of the jar file that contains the application. This explicitly couples the Java Swing and Java Net concerns. To avoid this, one of the other constructors provided by the `ImageIcon` class should be used instead, for example, by sending the raw image data as a byte array. Similar violations may occur when objects of the class `java.io.File` are passed as a parameter to a method. In future research, we may try to find ways to automatically identify and report such instances to avoid manual inspection.

The output also allows us to consider whether the use of import statements is a good basis to identify concerns in an application. On the one hand, we believe this is indeed a quick and convenient way to identify the primary concerns that are present in an application. The Normalized Systems theory states that the use of an external technology always implies a different concern that should be separated [10]. Since external technologies must be made available in Java application through the `import` statement, this provides a good way to identify concerns. On the other hand, this method may neglect the internal structure of the application to some extent (when different concerns are present within the application itself), and it also leaves some room for deciding on which concerns are present within the application. This may result in false positives or false negatives in the detection process. It may therefore be worthwhile to consider other methods for identifying concerns within an application.

Concerning the *Primitive in Interface Violation* module, we identified a large number of methods that include one or more primitives in their interface. It appears that a manual inspection is required to investigate whether it is worthwhile to resolve these issues by making the data and methods version transparent. In case the interface can be expected to remain stable, it may not be worth the effort to encapsulate the parameters in their own dedicated object. Nevertheless, developers should remain aware that not addressing this issue can mean that additional methods may call this method in the future, thereby leading to an increase in combinatorial effects. It is theoretically possible that some of the methods that include primitive data types have a corresponding wrapper method that is version transparent and that should be called instead of the underlying method. In other words, it is possible that the source code includes both a version non-transparent method and an additional version transparent method. Our tool is currently not able to detect such instances. However, given the large number of manifestations of this violation found in all four applications, it can be expected that this is not done very often. Moreover, any non-version transparent public method can be called by additional methods in the future, thereby leading to an increase in combinatorial effects.

With respect to the *Custom Exception Violation* module, we can easily identify those methods that throw a custom exception. Further investigation of the source code of the four applications with respect to this issue showed that in several cases the calling method did not do anything when a method throws an exception, except for logging the error. However, since this external method throws an exception, the class that contains the calling method must import the package containing the exception. For example, if an external method throws the `java.io.IOException`, it must be imported by the class containing the calling method. Interestingly, this may further contribute to the *Import Multiple Concerns Violation*, if that class also imports other concerns. We have indeed noticed that several user interface classes import the `java.io.IOException` class since they must be able to react to exceptions thrown by methods of other classes. Although we considered the use of the default Java exceptions to be acceptable, the same reasoning applies to custom exceptions or exceptions from external technologies. This further emphasizes the $M-N$ relationship between the design theorems and violations (see Section IV).

B. Limitations

Since this tool is still a prototype, we acknowledge several limitations with respect to our findings.

A first important limitation is that the tool currently provides a lower bound of manifestations of violations to the Normalized Systems design theorems. The results therefore

provide a first-cut and rough assessment of violations to the Normalized Systems design theorems at the API-level. This assessment can increase sensibilisation about—and give a first impression of—the code quality of an application with respect to evolvability. Currently, we have distinguished between three violations against the Normalized Systems design theorems in Java applications. Each module of `NSTVdetect` checks for manifestations of a specific violation. All modules share a common interface and receive an internal representation of the source code as input. The tool can be extended in the future with new modules that check for additional violations to the Normalized Systems design theorems in order to detect a larger number of violations to the Normalized Systems design theorems.

A second limitation is that there is a risk for the existence of false positives reported by the tool. Although our experiences suggest that it is feasible to automatically detect several violations to the Normalized Systems design theorems, it still remains necessary to perform a manual inspection of the source code afterwards. This manual inspection provides more insight into the seriousness of the issues identified in the analysis. This is especially the case for the import multiple concerns violation since the choice of how libraries are grouped into concerns is to some extent arbitrary. For example, the Java API can considered to be rather stable. Hence, importing packages from several parts of the Java API may not constitute a very large risk with respect to combinatorial effects. A manual inspection is therefore required to investigate whether some issues reported by the tool are false positives. Notwithstanding the fact that some manual work is still required, our tool significantly reduces the effort compared to manually inspecting the code base for violations. In addition, the tool can quickly highlight potentially problematic parts in the source code that should be analyzed manually with priority.

A third limitation is that the definition of which concerns are present in the application is still to some extent arbitrary, since the researcher must first define which concerns are present in the application based on the import statements that are used in the application. This approach first rises the question as to whether the use of a library is an appropriate indication of a concern. A concern is considered to be a separate change driver or, in other words, a technology that can change independently from the background technology. The use of an external library represents the use of an external technology and therefore always represents a different change driver or concern. Although there may be other concerns that do not correspond to the use of a library, our tool is therefore able to provide at least a lower bound of the concerns present in a given application. A second question with respect to this approach is whether all concerns are correctly identified. In this case study, we probably did not select the best concerns to evaluate the evolvability of the four applications. For example, since Java is the background technology, it would make sense not to identify Java as a separate concern. In the case study, we only considered the `java.lang.*` and `java.util.*` packages to be part of the background technology in order to identify a larger number of concerns that would allow us to better evaluate our tool. In case the tool would be used to assess the evolvability of a software product, it would make sense to only identify external technologies as a separate concern. In addition, we have assumed that all packages related to the application itself are part of the background technology. For example, in the case of JabRef, all packages below `net.sf.jabref.*` were ignored when creating the list of concerns. Depending on the application being assessed, it may be interesting to further distinguish between multiple concerns in the application itself, to allow different parts of the application to evolve independently.

A final issue concerns the question of how far software developers should go in adhering to the Normalized Systems theorems. We are aware that some of these theorems, their implications, and the violations identified by this tool may seem rather radical at first sight. As mentioned in Section II-B, the Normalized Systems theory uses the assumption of unlimited systems evolution [8]–[10]. This means that the code base of the application will continue to increase over time. The aim of the Normalized Systems design theorems is to eliminate all combinatorial effects. Since combinatorial effects are very easily introduced into the source code, very strict and clear design theorems are required to eliminate them [8]–[10]. In this respect, the Normalized Systems theory encourages software developers to strive towards applying these theorems to the greatest extent possible [9], [10]. In practice, some trade-off is likely to take place to judge whether the additional effort of containing combinatorial effects is warranted by the likelihood that a future change would manifest itself. For example, it is possible that it is reasonable to use primitive data types in the interface of some methods that are not exposed to outside applications and that can be expected not to require additional data in the future (i.e., to have a stable interface). Similarly, the use of custom exceptions may be appropriate in cases when the method is unlikely to be called by other parts of the application. However, such decisions should be carefully considered. Developers should also be aware that not adhering to the Normalized Systems design theorems may have a negative impact on the future evolvability of the software. In order to fully comply with the Normalized Systems design theorems, at least all the violations identified in the source code should be addressed.

Notwithstanding these limitations, we feel that this tool can be very useful to investigate the quality of an application at the API-level with respect to evolvability using the Normalized Systems design theorems.

VII. Conclusion and Future Work

In this paper, we have explored the feasibility to automatically identify violations to the Normalized Systems design theorems. To this end, we developed a prototype of a tool that is able to detect violations in Java applications at the API-level. A case study was performed to evaluate the tool by analyzing the source code of four open source Java applications. A first contribution of our study is that we have shown that it is indeed possible to detect violations to the Normalized Systems design theorems in an automated manner. A manual inspection should, however, still provide more insight into the seriousness of the issues identified in the analysis and to identify any possible false positives. A second contribution is that we have performed an analysis of the source code of four software applications. The results showed that the tool can be applied to a range of software applications and that there is considerable variety in how well the different theorems are adhered to by various software applications. Our results show that all four applications adhere rather well to the separation of concerns theorem. However, we identified a larger number of violations to the data and action version transparency theorems. With respect to the separation of states theorem, a rather mixed picture emerged. It can therefore be expected that the output of this tool will be useful to assess the current design of applications and to identify potential limitations to their evolvability. However, given the current limitations of our tool, we do not want to make any claims with respect to the quality of the four applications in our case study. Instead, this case study provided valuable feedback that will be used to further improve our tool. A third contribution is that we have distinguished between three violations to the Normalized Systems design theorems that may occur in Java applications. Although this list is not exhaustive, our results already show that quite a large number of manifestations of these violations can be found in the source code of Java applications. One of the limitations of our tool is that it currently provides a lower bound for the existence of violations, and is not able to detect all possible violations to the Normalized Systems design theorems.

In future research, we intend to further develop this tool to improve its ability to automatically detect violations to the Normalized Systems design theorems. Our current research efforts focus on two different topics.

First, we are are extending our list of violations to identify a larger proportion of violations to the Normalized Systems design theorems. Instead of the top-down approach used in this paper (i.e., by starting from the Normalized Systems design theorems and deriving which violations could be found in the source code of applications), we are currently using a bottom-up approach. This approach consists of reviewing the source code of a number of open source software applications with the aim of identifying fragments of the source code that represent manifestations of violations to one or more of the Normalized Systems design theorems. Based on these observations, different violations will be identified. This approach therefore has an empirical foundation, instead of the theoretical approach followed in this paper. For each new violation, an additional module can be added to the `NSTVdetect` tool.

Second, we are further developing the architecture underlying our tool. One focus area is the intermediate format of the database that is used to save the structure of the source code. We are currently investigating if a suitable ontology exists that can be used to store this information in a language-independent format, while still allowing for language-specific features to be added. This would facilitate the support of different programming languages by the tool. Another area is to evaluate whether the information obtained by `javadoc` is sufficient to identify the new violations that are being discovered in our work on the previous topic.

As the tool further evolves, we will also apply our tool to evaluate more complex information systems that incorporate various frameworks (e.g., software component models, user-interface frameworks, and communication frameworks). This would allow us to assess how many violations to the Normalized Systems theory are detected in typical information systems, compared to the rather simple projects included in our case study.

This work should greatly facilitate the automatic identification of combinatorial effects in large-scale, real-life information systems. This tool could be used by organizations to analyze their information systems for the manifestation of violations to the Normalized Systems design theorems. Based on the output of this tool, organizations can take measures to improve the evolvability of their information systems.

References

[1] K. Ven, D. Van Nuffel, D. Bellens, and P. Huysmans, "The automatic discovery of violations to the normalized systems design theorems: A feasibility study," in *Proceedings of the 5th International Conference on Software Engineering Advances (ICSEA 2010),* August 22–27, 2010, Nice, France, J. G. Hall, H. K. Kaindl, L. Lavazza, G. Buchgeher, and O. T. Takaki, Eds. Los Alamitos, CA: IEEE Computer Society, 2010, pp. 38–43.

[2] D. J. Teece, G. Pisano, and A. Shuen, "Dynamic capabilities and strategic management," *Strategic Management Journal*, vol. 18, no. 7, pp. 509–533, 1997.

[3] K. M. Eisenhardt and J. A. Martin, "Dynamic capabilities: What are they?" *Strategic Management Journal*, vol. 21, no. 10/11, pp. 1105–1121, 2000.

[4] S. Neumann and L. Fink, "Gaining agility through IT personnel capabilities: The mediating role of IT infrastructure capabilities," *Journal of the Association for Information Systems*, vol. 8, no. 8, pp. 440–462, 2007.

[5] V. Sambamurthy, A. Bharadwaj, and V. Grover, "Shaping agility through digital options: Reconceptualizing the role of information technology in contemporary firms," *MIS Quarterly*, vol. 27, no. 2, pp. 237–263, jun 2003.

[6] L. Fink and S. Neumann, "Exploring the perceived business value of the flexibility enabled by information technology infrastructure," *Information & Management*, vol. 46, no. 2, pp. 90–99, mar 2009.

[7] J. L. Zhao, M. Tanniru, and L.-J. Zhang, "Services computing as the foundation of enterprise agility: Overview of recent advances and introduction to the special issue," *Information Systems Frontiers*, vol. 9, no. 1, pp. 1–8, 2007.

[8] H. Mannaert and J. Verelst, *Normalized Systems—Re-creating Information Technology Based on Laws for Software Evolvability*. Kermt, Belgium: Koppa, 2009.

[9] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, in press. DOI: 10.1016/j.scico.2010.11.009.

[10] ——, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, in press. DOI: 10.1002/spe.1051.

[11] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sept. 1980.

[12] L. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[13] M. Lehman and J. Ramil, "Rules and tools for software evolution planning and management," *Annals of Software Engineering*, vol. 11, pp. 15–44, 2001.

[14] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 269–274, 1995.

[15] R. Bahsoon and W. Emmerich, "Evaluating software architectures: Development, stability, and evolution," in *Proceedings of ACS/IEEE International Conference on Computer Systems and Applications*, 2003.

[16] M. Ali Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation," in *Proceedings Australian Software Engineering Conference (ASWEC)*, 2004, pp. 309–318.

[17] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *Proceedings of the Fourth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'98)*, 1998.

[18] R. Kazman, G. Abowd, L. Bass, and M. Webb, "SAAM: A method for analyzing the properties of software architectures," in *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 81–90.

[19] M. M. Lehman, "Approach to a theory of software process and software evolution: Position paper," in *FEAST 2000 Workshop, Imperial College, London, July 10–12, 2000*, 2000.

[20] H. Mannaert, J. Verelst, and K. Ven, "Exploring the concept of systems theoretic stability as a starting point for a unified theory on software engineering," in *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA 2008),* Sliema, Malta, October 26-31, 2008, H. Mannaert, T. Ohta, C. Dini, and R. Pellerin, Eds. Los Alamitos, CA: IEEE CS Press, 2008, pp. 360–366.

[21] ——, "Exploring concepts for deterministic software engineering: Service interfaces, pattern expansion, and stability," in *International Conference on Software Engineering Advances*, Cap Esterel, France, Aug. 25–31, 2007.

[22] ——, "Towards rules and laws for software factories and evolvability: A case-driven approach," in *International Conference on Software Engineering Advances*, Tahiti, French Polynesia, Nov. 1–2, 2006.

[23] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[24] D. Van Nuffel, H. Mannaert, C. De Backer, and J. Verelst, "Towards a deterministic business process modelling method based on normalized systems theory," *International Journal on Advances in Software*, vol. 3, no. 1/2, pp. 54–69, 2010.

[25] J. Holmström, M. Ketokivi, and A.-P. Hameri, "Bridging practice and theory: A design science approach," *Decision Sciences*, vol. 40, no. 1, pp. 65–87, February 2009.

[26] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[27] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decision Support Systems*, vol. 15, no. 4, pp. 251–266, 1995.

[28] A. Newell and H. Simon, *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[29] H. A. Simon, *The Sciences of the Artificial*, 3rd ed. Cambridge, Massachusetts: MIT Press, 1996.

[30] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[31] A. Hevner and S. Chatterjee, *Design Research in Information Systems: Theory and Practice*, ser. Integrated Series in Information Systems. Springer, 2010, vol. 22.