Table 1: Constructs, Propositions, Explanations, and the Scope of the Dimension-Oriented Theory

Constructs	
C1	Functional Requirements Space
C2	Functional Requirements Classes (Problem Dimensions)
C3	System Functions
C4	Atomic Functional Requirements
C5	Data Output Functional Requirement Class
C6	Data Input Functional Requirement Class
C7	Event Trigger Functional Requirement Class
C8	Business Logic Functional Requirement Class
C9	Data Persistence Functional Requirement Class
C10	UI Navigation Functional Requirement Class
C11	External Call Functional Requirement Class
C12	Communication Functional Requirement Class
C13	User Interface Functional Requirement Class
C14	UI Logic Functional Requirement Class
C15	Data Validation Functional Requirement Class
C16	External Behavior Functional Requirement Class
C17	Core Functional Requirement Class (Core Problem Dimension)
C18	Non-Core Functional Requirement Class (Non-core Problem Dimension)
C19	Data Input Item
C20	Data Persistence Item
C21	Compound Requirement
Propositions	
P1	The *Functional Requirements Space* is composed of 14 known *Functional Requirements Classes*, including *Data Output*, *Data Input*, *Event Trigger*, *Business Logic*, *Data Persistence*, *UI Navigation*, *External Call*, *Communication*, *User Interface*, *UI Logic*, *Data Validation*, *External Behavior*, *Post-Condition*, and *Data Source*.
P2	The Nine functional requirements classes of *Data Output*, *Data Input*, *Event Trigger*, *Business Logic*, *Data Persistence*, *Data Validation*, *User Interface*, *User Interface (UI) Logic*, and *User Interface (UI) Navigation* belong to *Core Functional Requirement Classes*.
P3	The five functional requirements classes of *External Call*, *Communication*, *External Behavior*, *Post-Condition*, and *Data Source* belong to the *Non-core Functional Requirement Classes*.
P4	For every *Event Trigger*, there exists one corresponding *System Function*, *Compound Requirement*, or *Atomic Functional Requirement*.
P5	For every requirement of type *Data Validation*, there is a high probability that at least one corresponding requirement of type *Data Output* should also exist.
P6	For every *System Function*, there is a high probability that at least one requirement of type *Data Persistence* should also exist.
P7	For every *System Function*, there is a high probability that at least one requirement of type *Data Input* should also exist.
P8	For every *System Function*, there is a high probability that at least one requirement of type *Data Output* should also exist.
P9	For every *System Function*, there is a high probability that at least one requirement of type *Business Logic* should also exist.
P10	For every *Data Input Item*, there is a high probability that at least one corresponding *Data Persistence Item* should also exist.
P11	For every *Data Input Item*, there is a high probability that at least one corresponding requirement of type *Data Validation* exist.
P12	For every *System Function*, there exists at least one corresponding *Event Trigger*.
P13	For every violation of a business rule, there exists a system reaction as a corresponding compound requirement, consisting of a set of atomic requirements.

Table 2: Table 1 Continued

Explanations	
E1	The functional requirements space in the domain of enterprise applications can be logically divided into two partitions: (a) core functional requirements classes and (b) non-core functional requirements classes. As of yet, there are 9 known core and 5 known non-core classes of functional requirements. The sets of of core and non-core classes of functional requirements will be updated with future observations and discoveries. See E2 and E3 as to why core and non-core classes exist in the domain.
E2	Enterprise software systems, in order to support their corresponding business processes in the real-world, frequently need to run system functions (through Event Triggers), which collect information (Data Input), check the validity of inputted data (Data Validation), execute the prescribed business rules pertaining to the supported business process (Business Rules), store information as a result of executing the system functions (Data Persistence), and interact with users through displaying the results of executing the system functions, error messages, or success messages (Data Output). In GUI-based enterprise systems, Data Inputs and Data Outputs are tied to user interfaces; Inputs are collected through a user interface and system outputs are displayed on the user interface. These give rise to the three user interface-related classes of requirements including User Interface, User Interface Logic, and User Interface Navigation. The sequence of event trigger, input, data validation, business rule execution, output, and persistence, or a variation of this sequence, is very common in business systems, giving rise to the 9 core classes of functional requirements.
E3	In addition to the core functional requirements classes, enterprise systems occasionally need to support features that are not common in the domain, but specific to particular applications within the domain. Although compiling an exhaustive list of non-core requirements classes would require surveying every imaginable application in the domain and therefore an impossible task to carry on, in our studies so far, we have discovered 5 classes of non-core requirements.
E4	Event triggers, by definition, force the execution of either a single atomic requirement, a compound requirement, or a system function (e.g., a collection of atomic and compound requirements).
E5	When the execution of a data validation rule results in the detection of a violation of the corresponding correctness rules (i.e., detection of erroneous input), as a best practice, the user of the system needs to be notified, through appropriate error messages and possibly be given instructional messages, describing a course of action that can be taken to rectify the problem. Therefore, the existence of a requirement of type data validation, with a high probability, implies the existence of one or more related requirements of type data output.
E6	Most business services, and system functions as their implementations in enterprise systems, need to store information, typically collected from end users or generated as a result of a transaction with the end user. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing the data persistence aspects of the system function.
E7	Most business services, and system functions as their implementations in enterprise systems, need to collect information, typically from end users. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing the data input aspects of the system function.
E8	From P7 and P11, we can deduce that it is very likely that a system function will have at least one requirement of type data validation. According to P5, the existence of data validation requirements creates a high probability for the existence of corresponding data output requirements. Therefore, from P7, P11, and P5, we can conclude P8. Moreover, System functions often need to notify their end users of the successful completion of the system function through requirements of type data output, giving rise to further opportunities for the existence of requirements of type data output in system functions.
E9	Most business services, and system functions as their implementations in enterprise systems, involve the execution of business rules. Therefore, it is very likely for system functions to have one or more statements of requirements, describing the related business rules.
E10	Business services, and system functions as their implementations in enterprise systems, are very likely to need to store information collected from end users to conduct business and complete transactions. Therefore, it is very likely for input data items to be stored in the system. From a specification point of view, input data items reappear in statements of data persistence requirements.
E11	To prevent data entry errors, when such errors are possible, as a best practice, system functions need to validate inputted data items before consumption and storage in the system. Therefore, data input items may need one or more relevant statement of requirements describing the data validation rules.
E12	All System Functions should be accessible through at least one Event Trigger.
E13	Systems need to respond to violations of the business rules. This response is specified in the form a set of requirements.
Scope	
S1	The Domain of Enterprise Systems

2 A Formalism for Dimension-Orientation

Improving the precision, completeness, and consistency of documentation in software engineering has been a major goal for the software engineering community [2]. The use of mathematical expressions in software engineering artifacts has been a major part of the efforts directed towards achieving this goal as mathematics is the best way to achieve precision [2]. A mathematical basis for design allows many types of quantitative evaluations of an IT artifact, including optimization proofs, analytical simulation, and quantitative comparisons with alternative designs [14, 16]. In agreement with these considerations, in this subsection, we will express the propositions of the refined dimension-oriented theory, as presented in Table 1, in a precise and formal mathematical language. We will use a form of probabilistic logic to capture the notion of strong implication, which is present in several of our theory propositions. This can be accomplished by extending the first order logic with the notion of strong implication denoted as $\rightharpoonup$. In logical propositions involving implications, where the consequent is highly probable, but nor certain, we will replace the standard implication symbol ($\rightarrow$) with the strong implication symbol ($\rightharpoonup$), indicating a strong probability for the consequent, if the antecedent is true. Table 3 defines the set variables and predicates that will be used in our formal definitions. Note that since our theory concerns the atomic statements of requirements in the domain of enterprise systems, we define the following universal set, which contains all the elements in the universe of discourse:

$$U = \bigcup_{i=1}^{n} R_i = \bigcup_{i=1}^{n} Req(A_i)$$

where A_i is an imaginable enterprise system and $Req(A_i)$ returns the set R_i of all atomic functional requirements in enterprise system A_i. This universal set U represents the functional requirements space in the domain of enterprise systems, which is the subject of the dimension-oriented theory. Note that atomicity is a property of all the elements of the sets R_i and consequently the universal set U. We will define our axioms as follows:

Part-Whole Axiom:
$$\forall x, y[Prt(x, y) \leftrightarrow Whl(y, x)]$$

Non-Inclusion of Self Axiom:
$$\forall x[\neg Prt(x, x)]$$

Atomicity Axiom:
$$\forall x \neg \exists y[Atm(x) \leftrightarrow Prt(y, x)]$$

Compoundedness Axiom:
$$\forall x \exists y[Cmp(x) \leftrightarrow Prt(y, x)]$$

Single-Dimensionality Axiom:
$$\forall x \neg \exists y \neg \exists z[Sdm(x) \rightarrow x \in y \wedge x \in z]$$

where in the Single-Dimensionality Axiom $x \in U, y \in D, z \in D, y \neq z$.

Completeness Axiom:
$$\forall x \exists y[x \in y]$$

where in the Completeness Axiom $x \in R, y \in D$.

This last axiom is known as the Completeness Axiom as it asserts that every atomic requirement in a particular system A_i is assigned to a problem dimension from the set D of all known problem dimensions. That is, set D is complete over all the requirements of A_i. This implies that there exist a classifier function that maps every atomic requirement from system A_i to exactly one of the classes of set D of functional requirements classes in enterprise systems.

Note that from the Atomicity and the Compoundedness axioms above we have:

Dichotomy Theorem:
$$Cmp(x) \leftrightarrow \neg Atm(x)$$

An inherent and important property of atomic requirements, then, is that they are necessarily single-dimensional. That is:

Single-Dimensionality of Atoms Theorem:
$$\forall x[Atm(x) \rightarrow Sdm(x)]$$

The functional requirements space is partitioned into subsets, each representing a class of functional requirements - a dimension. As shown in the formulae P1 below, and stated in Proposition P1 of the theory, the classes of functional requirements are further categorized into two classes of core and non-core classes. Formula P2 and P3 further show the known classes of core and non-core functional requirements, as described in propositions P2 and P3, respectively. The remaining formulas correspond to their respective propositions from Table 1.

$$D = C \cup N \quad \text{(P1)}$$

$$C = O \cup I \cup T \cup B \cup P \cup V \cup U_1 \cup U_2 \cup U_3 \quad \text{(P2)}$$

Table 3: Set Variables

Set	Variable
Set of Functional Requirements Classes in Enterprise Systems	D
Set of Core Functional Requirement Class in Enterprise Systems	C
Set of Non-Core Functional Requirement Class in Enterprise Systems	N
Set of Atomic Functional Requirements for a particular Enterprise System A_i	R
Set of Compound Functional Requirements for a particular Enterprise System A_i	M
Set of System Functions for a Particular Enterprise System A_i	F
Returns True if atomic requirement x is an Event Trigger	T(x)
Returns True if atomic requirement x is a Data Validation	V(x)
Returns True if atomic requirement x is a Data Persistence	P(x)
Returns True if atomic requirement x is a Data Output	O(x)
Returns True if atomic requirement x is a Data Input	I(x)
Returns True if atomic requirement x is a Business Logic	B(x)
Returns True if atomic requirement x is a UI Navigation	Unv(x)
Returns True if atomic requirement x is a UI Logic	Uil(x)
Returns True if atomic requirement x is a UI	Uin(x)
Returns the set R_i of atomic functional requirements for System A_i	Req(A_i)
Returns the set of data input items for a data input requirement x	Itm(x)
Requirement x triggers atomic or compound requirement, or system function y	Trg(x,y)
Requirement x is part of requirement y	Prt(x,y)
Requirement x is whole of (i.e., includes) requirement y	Whl(x,y)
Requirement x outputs a message about requirement y	Out(x,y)
Requirement x specifies a validity rule for data item y	Vld(x,y)
Requirement x specifies the storage for data input item y	Str(x,y)
Requirements set Y specifies the response to Violation of Requirement x	Rsp(x,Y)
Returns True if requirement x is atomic	Atm(x)
Returns True if requirement x is compound	Cmp(x)
Returns True if requirement x is single-dimensional	Sdm(x)
Set of Data Output Atomic Requirements $\{r \in R : O(r)\}$	O
Set of Data Input Atomic Requirements $\{r \in R : I(r)\}$	I
Set of Event Trigger Atomic Requirements $\{r \in R : T(r)\}$	T
Set of Business Logic Atomic Requirements $\{r \in R : B(r)\}$	B
Set of Data Persistence Atomic Requirements $\{r \in R : P(r)\}$	P
Set of UI Navigation Atomic Requirements $\{r \in R : Unv(r)\}$	U_3
Set of External Call Atomic Requirements	N_1
Set of Communication Atomic Requirements	N_2
Set of User Interface Atomic Requirements (UI) $\{r \in R : Uin(r)\}$	U_1
Set of UI Logic Atomic Requirements $\{r \in R : Uil(r)\}$	U_2
Set of Data Validation Atomic Requirements	V
Set of External Behavior Atomic Requirements	N_3
Set of Post-Condition Atomic Requirements	N_4
Set of Data Source Atomic Requirements	N_5

$$N = N_1 \cup N_2 \cup N_3 \cup N_4 \cup N_5 \quad \text{(P3)}$$

$$\forall r_i \exists r_j \exists t \exists m [r_i \in R \wedge T(r_i) \rightarrow r_j \in R \wedge r_i \neq r_j \wedge Trg(r_i, r_j) \vee m \in M \wedge Trg(r_i, m) \vee t \in F \wedge Trg(r_i, t)] \quad \text{(P4)}$$

$$\forall r_i \exists r_j [r_i \in R \wedge V(r_i) \rightharpoonup r_j \in R \wedge Out(r_j, r_i)] \quad \text{(P5)}$$

$$\forall t \exists r [t \in F \rightharpoonup (Prt(r, t) \wedge P(r))] \quad \text{(P6)}$$

$$\forall t \exists r [t \in F \rightharpoonup (Prt(r, t) \wedge I(r))] \quad \text{(P7)}$$

$$\forall t \exists r [t \in F \rightharpoonup (Prt(r, t) \wedge O(r))] \quad \text{(P8)}$$

$$\forall t \exists r [t \in F \rightharpoonup (Prt(r, t) \wedge B(r))] \quad \text{(P9)}$$

$$\forall r_i \forall d \exists r_j [r_i \in R \wedge I(r_i) \wedge d_i \in Itm(r_i) \rightharpoonup r_j \in R \wedge P(r_j) \wedge Str(r_j, d)] \quad \text{(P10)}$$

$$\forall r_i \forall d \exists r_j [r_i \in R \wedge I(r_i) \wedge d \in Itm(r_i) \rightharpoonup r_j \in R \wedge V(r_j) \wedge Vld(r_j, d)] \quad \text{(P11)}$$

$$\forall t \exists r [t \in F \rightarrow r \in R \wedge T(r) \wedge Trg(r, t)] \quad \text{(P12)}$$

$$\forall r_i \exists r_j [r_i \in R \wedge B(r_i) \rightharpoonup r_j \in M \wedge Rsp(r_j, r_i)] \quad \text{(P13)}$$

3 Dimension-Oriented Requirements Engineering (DORE)

Having developed a formalized theory that is capable of making predictions about the requirements space in the domain of enterprise systems, we now put the theory's predictions into practical use by using them as input to the design process for a requirements engineering process. The result is the following process, called the Dimension-Oriented Requirements Engineering (DORE), for the development of requirements specifications for enterprise systems. The requirements engineer compiles and negotiates a list of system functions (e.g., use cases, features, high-level requirements, or system capabilities) needed by project stakeholders for a business system and then follows the following process to develop a complete specification for each individual system function. Note how the process steps are justified based upon the theory's propositions.

1. **Identify and Specify Triggers - Justification: P12 -** Identity all the alternative ways (i.e., stimuli) to trigger the execution of the system function (e.g., system behavior). A system behavior can be activated either by:

   (a) User generated trigger - an end user signifying the system, for instance, through generating a user interface event, such as clicking on a button, link, or menu item, or typing in a command at a command prompt.

   (b) System generated trigger - a system variable reaching a predefined threshold. For instance, the automatic activation of a system function to place an order for some quantity of a good, when the inventory level for that good reaches a predefined minimum level. Another example would be a recurring time trigger that automatically activates a system behavior in predefined periodic time intervals.

   For each identified event trigger, add a uniquely identifiable and atomic statement of requirement to the system function specification, describing the event trigger. Tag each such requirement as belonging to the event trigger class of functional requirements.

2. **Identify and Specify Inputs to System Functions - Justification: P7 -** For each documented user generated event trigger in Step 1, identify whether or not the event trigger requires to pass any information to the system function. In other words, does the system behavior require user-provided external input before it can be activated? A system service, such as the one in a typical banking system that opens a bank account for a customer, requires personal information about the clients as well as information about the type of account being opened. In contrast, a reporting system function that is activated by a user interface event and displays a particular report all based on default values or the existing data of the system is an example of a trigger that does not requires user input. An event trigger

that merely makes the user interface for a particular system function accessible to the system user is another example of a simple event trigger without any user data input. The description for many system use cases start with such an event trigger, where the end user of the system notifies the system of his or her intention to execute a system function by generating a user interface event. In response to such an event, the system displays the input form to collect the required information for the execution of the system function. The user, after entering all the information, eventually generates another event trigger to execute the service, but unlike the previous event trigger, this latter trigger passes the information entered by the user to the system function.

For each identified system function with input add a uniquely identifiable and atomic statement of requirement to the system function specification, explicitly listing all the input data items that must be entered into the system by the end user. Tag each such requirement as belonging to the data input class of functional requirements.

3. **Identify and Specify Data Validations - Justification: P11 -** For every data input item in every data input requirement specified in Step 2, identify which data inputs require validation to detect and prevent data entry problems and what those validation rules are for each such data input item.

   For every identified data input item with validations specify each data validation rule as a uniquely identifiable and atomic statement of requirement. Tag each such requirement as belonging to the data validation class of functional requirements.

4. **Identify and Specify Outputs for Violated Validation Rules- Justification: P5 -** For every data validation requirement, specified in Step 3, identify whether an alert message needs to be displayed and what the format and content of the message should be, in case the validation rule is violated, to inform the end user of the erroneous data entry and guide the user to rectify the problem.

   For every data validation requiring relevant output messages add a uniquely identifiable and atomic statement of requirement, explicitly specifying the content and format of the message to be displayed. Tag each such requirement as belonging to the data output class of functional requirements.

5. **Specify User Interface for Inputs and Outputs - Justification: P2 -** For each system function that requires external inputs from users (identified in Step 2), add requirements to the system function specification that describe the interface (e.g., GUI such as an input screen) that will be needed to collect the input data items specified in Step 2. The specification of the user interface requirements for a system function will include a collection of uniquely identifiable and atomic statements of requirements that together specify the various aspects of the user interface including the user interface components that will be used for data entry, the style of the data input (e.g., entering text or selecting from a list), the location of the components, and cosmetic aspects such as the layout, font, and color of the data input screen. One or more user interfaces might be needed for a single system function. Conversely, multiple system functions can share the same user interface. Visual descriptions of user interface requirements, such as screen mock-ups, are an option here. Some user interface requirements can also be derived from the data output requirements, documented in Step 4, as these outputs are displayed through the user interface and, therefore, can add new statements of requirements to the specification. An examples of such a requirement would be a statement describing where an output should be placed on a screen. Note that in command line systems the interactions between the system and the user takes place through a series of data inputs and outputs rather than a graphical interface.

6. **Identify and Specify Event Trigger for UI Level Data Validations -** In GUI-based systems, for every data validation requirement specified in Step 3, determine if the rule needs to be checked at the graphical user interface level, at the system back end, or both.

   For every validation rule identified as requiring validation at the UI level, add a uniquely identifiable and atomic statement of requirement that describes the user interface event trigger for the corresponding user interface data input component. Tag each such requirement as belonging to the event trigger class of functional requirements.

7. **Identify and Specify the Business Rules - Justification: P9 -** Identify the set of business rules that are relevant to the system function or business service. Business rules typically specify calculations or rule checking using existing data of

the system and/or the new data passed to the system function. Statements of functional requirements describing business rules define the rules of the application domain. These are the rules that govern the operations in an enterprise domain (e.g., banking, insurance, accounting, inventory management, order processing, etc.).

Document each identified business rule as a uniquely identifiable and atomic statement of requirement. Tag each such requirement as belonging to the business rule class of functional requirements.

8. **Identify and Specify System Reaction for Violated Business Rules - Justification:P13 -** For every business rule, specified in Step 7, identify the appropriate system response/reaction to the, in case the business rule is violated. This system reaction is specified as a set of one or more requirements. The reaction requirements set can include requirements of various types. As an example of a simple response to the violation of a business rule, a system may display an alert message to the end user. This will lead to the addition of a requirement of type output to the system function specification. As another example, if the system needs to roll back changes to a database as a result of a violation of a business rule, this rollback response can be specified as a requirement of type data persistence and added to the system function specification.

   For every requirement identified as being part of the system response to the violated business rule, add a uniquely identifiable and atomic statement of requirements. Tag each such requirement with its proper class of functional requirements.

9. **Identify and Specify Data Persistence - Justification: P6 and P10 -** Identify a list of items and their values that need to be persisted as a result of executing the system function. Data persistence requirements specify all database create, read, update, and delete operations. The list of items requiring persistence often includes, but is not limited to, many of the data input items listed under the data input requirements, documented in Step 2.

   Add one or more uniquely identifiable atomic statements of requirements to the specification of the system function, describing the required persistence operation, explicitly stating the list of entities or items along with their values. Tag each such statement of requirement as belonging to the data persistence class of functional requirements.

10. **Identify and Specify Data Outputs for System Function Results - Justification: P2, P8 -** The execution of system functions results in either a successful completion or a failure. In either case, the user of the system needs to be notified of the failure or success of the service. In the case of successful execution, results need to be displayed to the user. Success and failure messages as well as the results of the execution of system functions are described using statements of requirements that specify the content and format of the outputs. Tag each such requirement as belonging to the data output class of functional requirements.

11. **Specify UI Navigation - Justification: P2 -** If the fulfillment of a condition transfers the controls of the application from the current user interface to another user interface of the system (e.g., another screen) then document this transition as a uniquely identifiable and atomic statement of requirement. Tag each such requirement as belonging to the UI navigation class of functional requirements. UI navigation requirements describe the flow of the application. An example of such actions includes the violation of a data validation or business rule which takes the user to an error screen.

12. **Specify UI Logic - Justification: P2 -** For every user interface, such as a system screen, determine if the interface will engage in a dynamic interaction with the end user. For instance, a system screen may dynamically update itself and present a different set of user interface components based on an end user's selection of an item on the screen. Document the rules for such user interface interactions as uniquely identifiable and atomic statements of requirements. Tag each such requirement as belonging to the UI logic class of functional requirements.

4 Controlled Evaluation of DORE

In the previous steps, we used the dimension-oriented theory as a theoretical foundation, on top of which, we designed the DORE approach to requirements engineering. To evaluate the practical effectiveness of this requirements method in real-world software engineering situations, we must put it into practice by applying it to the requirements phase in enterprise system development projects. This will allow us to collect evidence on the effectiveness or ineffectiveness of the method. Such evidence can then be used to improve the design of the method. However, as a

first step in evaluation, it is advisable to test the process in one or more controlled or simulated smaller-scale proof-of-concept projects before introducing the method to a real-life software engineering context, where, due to schedule and cost constraints, the stakes are high. Successful applications of the method to pilot projects as well as collecting and applying feedback from such controlled evaluations to improve the method will help us to increase our confidence on the effectiveness of the process in real-world large-scale projects. Accordingly, we applied the DORE method to the requirements phase of an information system development project to evaluate its effectiveness in producing complete requirements specifications. The project, wherein the DORE method was tested, was a web-based information system that would combine social networking capabilities with Geographical Information System (GIS) capabilities, allowing its end users to shares various categories of stories and news with friends in their network. Stories were geographically tagged, so it would allow the users to visually navigate the network of friends and stories through an interactive world map. The system included 11 system functions and 90 statements of functional requirements. A description of this system can be found in [19]. Our criterion for the effectiveness of the requirements method is the completeness of the specification that it produces. In our case study, which is reported in [19], the DORE method helped the requirements engineer to discover and specify a significant majority of the system functional requirements (over 90%) during the requirements phase, which is very encouraging. In a number of cases, the statements of requirements were misclassified (i.e., assigned to the incorrect class), which were discovered and fixed during the requirements inspection process. We updated the descriptions of the requirements classes as well as the process steps to minimize the possibility of incorrect classifications. We also felt that a short training workshop and training material could have significantly prevented the occurrence of such cases. We incorporated these insights into our plan for the preparation of the method before its introduction to large-scale industrial projects. Planning is currently underway for the introduction of the DORE method to a number of industrial projects. Training on DORE is part of this plan.

The development of theories and theory-based methods in software engineering involve continuous cycles of refinements. Theories drive the development of software engineering methods, while the practical application of such methods provide feedback to refine their underlying theories. Once we evaluate the DORE method in industrial projects, we will use the results from such industrial evaluations to further improve the DORE process.

The dimension-oriented theory presented in this paper captured the essence of functional requirements in enterprise systems and empirical evidence thus far confirms that it is capable of making fairly accurate predictions about its domain of discourse. However, it must be acknowledged that since each software engineering setting is unique, software engineering theories might need local adaptations to be directly useful in concrete cases [23]. A corollary of this is that the software engineering methods that are based on such theories might need local adaptations as well to reflect the adaptations in their underlying theories. Therefore, although we believe the dimension-oriented requirements method should be sufficient for most enterprise systems, it is expected that local adaptations to specific organizational and project contexts might further increase its effectiveness.

5 Evaluating Theoretical and Methodological Aspects in DORE

The ultimate criterion to judge dimension orientation as a software engineering paradigm, and in particular DORE, is how useful it is to the software engineering practitioners in the software industry. However, there are useful evaluation frameworks that provide criteria for the evaluation of the theoretical and methodological aspects of a theory or approach. In this section, we will use two such widely-used frameworks, namely the software engineering theory goodness criteria by Sjoberg et al. [23] and the design science research guidelines by Hevner et al [14], to evaluate the goodness of the dimension-oriented theory as well as the proposed research and design methodology, which was used to develop the DORE method, respectively.

5.1 The Goodness of the Dimension-Oriented Theory

Sjoberg et al. [23] provide a framework for evaluating empirically-based theories in software engineering that includes the followings six criteria:

- Testability
- Empirical Support
- Explanatory Power
- Parsimony
- Generality
- Utility

We will briefly discuss each of these criteria in the context of the dimension-oriented theory presented in this paper. Testability, refers to the degree to which the empirical refutation of a theory is possible [23] and is the criterion that distinguishes science from non-science [18, 23]. Sjoberg et al. [23] emphasize the following three criteria for testability:

- The constructs and propositions of a theory should be clear and precise such that they are understandable, internally consistent, and free from ambiguities
- It must be possible to deduce hypotheses from the theory's propositions, so that they may be confirmed or dis-confirmed
- The theory's scope conditions must be explicitly and clearly specified, so that the domain or situations in which the theory should be confirmed or dis-confirmed and applied is clear.

The constructs and propositions of the dimension-oriented theory are clearly stated in Table 1. We further mathematically formalized the theory constructs and propositions to achieve precision and avoid ambiguities. It is easy to derive hypotheses from this theory's propositions that can be tested for validity. In fact, this is precisely what we did when we evaluated the predictive power of the theory in our previous work. The theory's domain was clearly stated to be the domain of business information systems. Therefore, we believe that the dimension-oriented theory, presented in this paper, meets the testability criterion for a theory.

The empirical support criterion refers to the degree to which a theory is supported by empirical studies that confirms its validity [23]. We used data from 18 software projects to build and test the dimension-oriented theory. We believe these 18 cases provided us a large enough empirical data set that supports the theory. However, as with any other theory, replicated studies to test the theory can increase our confidence to the validity of the theory by aggregating further evidence to support the theory. We plan for further replications of the study as future work.

The explanatory power of a theory refers to the degree to which a theory accounts for and predicts all known observations within its scope. It also refer to how well the theory relates to what is already well known [23]. In previous work, we evaluated the predictive power of the dimension-oriented theory and demonstrated that it has a high predictive power in the domain of enterprise systems.

Parsimony refers to the degree to which a theory is economically constructed with a minimum of concepts and propositions [23]. We tried to achieve some degree of parsimony by detecting and removing redundant constructs and propositions from the theory. For instance, in an earlier version of the theory, we had a Proposition P5, which was later removed in the subsequent version of the theory because Proposition P12 subsumed it and therefore Proposition P5 was not necessary.

Generality refers to the breadth of the scope of a theory and the degree to which the theory is independent of specific settings [23]. A theory with a large scope, such as the domain of all software projects, will have a larger explanatory breath and thus a broader applicability, but may demand more effort in operationalizing the theory to a given situation. In contrast, a lesser generality might make a theory immediately applicable [23]. As we discussed earlier in this paper, our main goal from theory building was to inform the design process for a requirements method for the domain of enterprise systems and therefore direct applicability was our top priority. Accordingly, we limited our theory to the domain of enterprise systems. However, as future work, we plan to conduct similar studies in other software domains to understand the degree to which the dimension-oriented theory holds for other domains.

The utility of a theory refers to the degree to which a theory supports the relevant areas of the software industry [23]. Our goal in developing the dimension-oriented theory was to support the requirements process in the domain of enterprise systems. We believe the theory completely supports this area and thus has a high practical utility.

5.2 Conformance to Design Science Guidelines

The work reported in this paper nicely fits within the design science paradigm [14], which provides a framework for information systems (IS) research. Hevner et al. [14] established the following seven guidelines that set the requirements for effective design science research:

- Guideline 1: Design as an Artifact
- Guideline 2: Problem Relevance
- Guideline 3: Design Evaluation
- Guideline 4: Research Contributions
- Guideline 5: Research Rigor
- Guideline 6: Design as a Search Process
- Guideline 7: Communication of Research

We will briefly discuss what each guideline means and how we addressed the guideline. The first guideline, design as an artifact, states that design science research must produce a viable Information Technology (IT) artifact in one of the following four forms, which is created to address an important organizational problem [14]:

- Construct (i.e., vocabulary and symbols)
- Model (i.e., abstractions and representations)
- Method (i.e., algorithms and practices)
- Instantiation (i.e., implemented and prototype systems)

These four types of artifacts provide guidelines and prescriptions that allow us to understand and solve problems inherent in the development and implementation of information systems within organizations [15, 17]. Constructs provide the vocabulary and symbols used to define problems and solutions [14] and as such have a significant impact on the way in which tasks and problems are conceived [1, 20]. In our work, the concept of a problem dimension is the main construct. It allows us to view and describe software engineering problem spaces, such as the functional requirements space, as a set of problem dimensions, which in turn impacts how we perceive and attempt to solve software engineering problems, hence the name Dimension-Oriented Software Engineering (DOSE). The search for an effective problem representation is crucial to finding an effective design solution [26]. Simon [22] states that "solving a problem simply means representing it so as to make the solution transparent".

We argue that representing software problem spaces as a set of problem dimensions indeed makes their solutions transparent. We developed a theory that revolves around the notion of problem dimensions. The laws and the explanations of the proposed theory explain and predict the nature and the relationships among the problem dimensions in the domain of enterprise systems and, therefore, provide a model for the domain of enterprise systems. In previous work, we demonstrated that this model is a fairly accurate representation of systems in the domain of enterprise systems. We then used this empirically-verified model in building a requirements practice around it (Subsection 3) - the dimension-oriented requirements engineering. In Subsection 4, we further demonstrated the feasibility and effectiveness of the new requirements process by instantiating it in the context of a proof-of-concept information system development project. Therefore, our work involved the creation of all four types of IT artifacts that are the subject of the design science research paradigm.

The second guideline, the problem relevance, states that the objective of design-science research is to develop technology-based solutions to important and relevant business problems. In the introduction to this paper, we discussed that the quest for software engineering approaches that are economically viable is both important and relevant to the business organizations, who depend on information technology to achieve their business goals, namely increasing revenue and decreasing cost. The dimension-oriented requirements process described in this paper, is a step toward achieving a more effective and productive approach for developing high-quality information systems. A high-quality and complete specification can avoid unnecessary rework and, therefore, make a significant contribution to achieving a more economic approach to information systems development.

The third guideline, design evaluation, emphasizes that the utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods including observational methods such as case and field studies, analytical methods such as static analysis, architecture analysis, optimization, and dynamic analysis, experimental methods such as controlled experiments and simulation, testing-based methods such as functional (black box) testing and structural (white box) testing, and descriptive methods such as informed arguments and scenarios [14].

Evaluation is an integral part of the research and design methodology that was followed to produce the dimension-oriented theory as well as the dimension-oriented requirements engineering approach presented in this present work. For instance, we evaluated (a) the validity of the hypothesis underlying the dimension-oriented theory, (b) the validity and the predictive power of the dimension-oriented theory that is formed based on the accepted hypothesis, (c) the effectiveness of the method that has been built based on this theoretical foundation in a controlled environment, and (d) the resulting DORE method by putting it into practice in the real environment. Results from every form of evaluation is fed back to the design process, and is used to improve the process.

The fourth guideline, research contributions, states that effective design-science research must provide clear and verifiable contributions in the areas of design artifact, design foundations, and/or design methodologies. The work reported in this paper make a number of such contributes including a design methodology, a set of constructs to view and represent software engineering problem spaces such as the functional requirements space, a domain model for

enterprise systems, a theory, a requirement engineering method, and a series of evaluations.

The fifth guideline, research rigor, states that design-science research relies upon the application of rigorous methods both in the construction and evaluation of the design artifact [14]. The design methodology presented in this paper relies on well-established scientific research methods including, hypothesizing, hypothesis testing via adherence to appropriate data collection and analysis techniques, theory building and evaluation as well as other established best practices in empirical software engineering research.

The sixth guideline, design as a search process, states that the search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment [14]. Hevner et al. [14] emphasize that the search for the best, or optimal, design is often intractable for realistic information systems problems. Therefore, it involves a heuristic-based iterative search process for a satisfactory solution. The requirements process, presented in this paper, was designed through several cycles of design, evaluation, and refinements. We developed a theory to inform and facilitate the initial design of the process and then used feedback from several types of evaluations to refine and improve both the process and its underlying theory. The end result is a repeatable requirements process that is capable of guiding the requirements engineer through the task of creating a complete functional specification for systems in the domain of enterprise systems. The design of the process steps is based on the laws of the domain and helps to minimize specification defects in the form of missing requirements hence producing a complete specification of functional requirements.

The seventh guideline, the communication of research, emphasizes that design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences [14]. Technology-oriented audiences need sufficient details in order to be able to implement the described design artifact. We have described a step by step requirements process (see Subsection 3), which should be easy for requirements engineers to follow to produce functional specifications for their enterprise projects. Hevner et al. [14] also emphasize that it is important for such audiences to understand the processes by which the artifact - in our case, the DORE requirements process - was constructed and evaluated. The benefits of such an understanding is that it establishes repeatability of the research project and builds the knowledge base for further research extensions by design-science researchers in information systems. We described our design methodology in previous work and rigorously followed it to design the DORE requirements process. Therefore, we believe it should be possible for information systems researchers to reuse the proposed design methodology to design novel software engineering methods or extend the work reported in this paper. In the related work section of this paper, we will discuss some possible directions for future work.

Management-oriented audiences, on the other hand, need sufficient details in order to decide whether organizational resources should be committed to constructing, or purchasing, and using the artifact within their specific organizational context. In our work, we attempted to achieve this by conducting case studies and collecting evidence on the effectiveness of the proposed requirements methods. Data from our case studies thus far confirm that the process can help IS development organizations to be more productive in creating functional specifications for enterprise systems (i.e., economic incentive), while helping them to produce higher-quality functional specifications (i.e., more complete), which in turn can increase the quality of the resulting information systems.

6 Conclusions and Directions for Future Work

Our position is that a fruitful avenue for designing more effective software engineering processes is to build domain-specific software engineering theories that can then be used as a solid scientific foundation for the design of various life cycle processes within the domains of interest. Accordingly, in this paper, we presented a domain-specific approach for requirements engineering in the domain of business information systems, called Dimension-Oriented Requirements engineering (DORE), and demonstrated how every step in the DORE process is backed by theory propositions and their corresponding explanations from the dimension-oriented theory of requirements space. More details about the dimension-oriented theory and its practical applications in various areas of software engineering can be found in our previous works (see [3, 11, 4, 10, 5, 6, 9, 8, 7, 13, 12], for example). In this paper, we demonstrated yet another practical application of this theory in the area of requirements engineering for the domain of business information systems. We also evaluated the methodological and theoretical aspects of the work reported here through the two frameworks of design science research [14] and software engineering theory goodness criteria [23].

In future work, we plan to use dimension-orientation as a theoretical foundation to improve other software life cycle processes, including architec-

ture [7][25]. Work on a dimension-oriented approach to test case development is currently underway. We also plan to apply dimension-orientation to other software domains, including the domain of scientific simulation applications [21][24]. Finally, we plan to introduce the DORE requirements process into a number of industrial projects and use the resulting feedback to further improve the DORE process.

References:

[1] R.J. Boland, Design in the Punctuation of management science, in *Managing as Design: Creating a Vocabulary for Management Education and Research*, R. Boland (ed.), Frontiers of Management Workshop, Weatherhead School of Management, June 2002.

[2] X. Feng, and D.L. Parnas, and T.H. Tse, T. O'Callaghan, A Comparison of Tabular Expression-Based Testing Strategies, in IEEE Transactions on Software Engineering, IEEE Computer Society, Vol. 37, No. 5, September/October 2011, pp. 616-634.

[3] A. Ghazarian, Characterization of Functional Software Requirements Space: The Law of Requirements Taxonomic Growth, in *Proceedings of 20th IEEE International Requirements Engineering Conference (RE'2012)*, Chicago, USA, September 2012.

[4] A. Ghazarian, Coordinated Software Development: A Framework for Reasoning about Trace Links in Software Systems, *Proc. of the IEEE 13th Int'l Conf. on Intelligent Engineering Systems (INES 2009)*, April 2009, pp. 39–44.

[5] A. Ghazarian, Effects of Source Code Regularity on Software Maintainability: An Empirical Study, *Proc. of the IASTED Int'l Conf. on Software Engineering and Applications (SEA 2010)*, Marina del Rey, USA, November 2010.

[6] A. Ghazarian, A Probabilistic Mathematical Model to Measure Software Regularity, *Proceedings of the 15th IASTED International Conference on Software Engineering and Applications (SEA 2011)*, Dallas, December 2011, USA.

[7] A. Ghazarian, A Domain-Specific Architectural Foundation for Engineering of Numerical Software Systems, *WSEAS Transactions on Systems*, No. 7, Vol. 10, July 2011, pp. 193–208.

[8] A. Ghazarian, Traceability Patterns: An Approach to Requirement-Component Traceability in Agile Software Development, *Proc. of the 8th WSEAS Int'l Conf. on Applied Computer Science (ACS'08)*, Venice, 2008, pp. 236–241.

[9] A. Ghazarian, A Formal Scheme for Systematic Translation of Software Requirements to Source Code, *Proceedings of WSEAS Applied Computing Conference (ACC 2011)*, Angers, France, November 2011, pp. 44–49.

[10] A. Ghazarian, A Matrix-Less Model for Tracing Software Requirements to Source Code, *Int'l Journal of Computers*, NAUN, ISSN: 1998-4308, Issue 3, Volume 2, 2008, pp. 301–309.

[11] A. Ghazarian, M. Sagheb-Tehrani, and A. Ghazarian, A Software Requirements Specification Framework for Objective Pattern Recognition: A Set-Theoretic Classification Approach, *Proc. of the 16th IEEE Intl Conf. on Engineering of Complex Computer Systems (CECCS 2011)*, USA, 2011, pp. 211–220.

[12] A. Ghazarian, R. Chughtai, Dimension-Oriented Inspection of Use Case-Based Requirements Specifications, *Proceedings of the European Conference of Computer Science (ECCS12)*, WSEAS/NAUN, Paris, France, December 2012.

[13] A. Ghazarian, Dimension-Driven Software Development Through Traceability Patterns, *Proceedings of the European Conference of Computer Science (ECCS12)*, WSEAS/NAUN, Paris, France, December 2012.

[14] A.R. Hevner, and S.T. March, and J. Park, and S. Ram, Design Science in Information Systems Research, MIS Quarterly, Vol. 28, No. 1, March 2004, pp. 75-105.

[15] S. T. March, and G. Smith, Design and Natural Science Research on Information Technology, Decision Support Systems (15:4), December 1995, pp. 251-266.

[16] F. Neri, Learning and Predicting Financial Time Series by Combining Evolutionary Computation and Agent Simulation, Applications of Evolutionary Computation, EvoApplications, LNCS 6625, pp. 111–119, Springer, Heidelberg (2011).

[17] J. Nunamaker, and M. Chen, and T. D. M. Purdin, Systems Development in Information Systems Research, *Journal of Management Information Systems* (7:3), inter 1991a, pp. 89-106.

[18] K. Popper, The Logic of Scientific Discovery, Hutchison, London, 1959.

[19] R. Pulavarthi, and A. Ghazarian, An Interactive Network of Events with Geographic Perspective, *WSEAS Transactions on Information Science and Applications*, No. 12, Vol. 9, pp. 369-378, World Scientific and Engineering Academy and Society, December 2012.

[20] D. A. Schon, The Reflective Practitioner: How Professionals Think in Action, Basic Books, New York, 1983.

[21] L. Shuang, W. Zhixin, W. Guoqiang, A Feedback Linearization Based Control Strategy for VSC-HVDC Transmission Converters, WSEAS Transactions on Systems, Issue 2, Volume 10, pp. 49-58, February 2011.

[22] H. A. Simon, The Sciences of the Artificial, 3rd Edition, MIT Press, Cambridge, MA, 1996.

[23] D.I.K. Sjoberg, T. Dyba, B.C.D. Anda, and J.E. Hannay,Building Theories in SOftware Engineering, Guide to Advanced Empirical Software engineering, F. Shull et al. (eds.), Springer, 2008, pp. 312-336.

[24] T-S Tsay, Intelligent Guidance and Control Laws for an Autonomous Underwater Vehicle, WSEAS Transactions on Systems, Issue 5, Volume 9, pp. 463-475, May 2010.

[25] S-G Yoo, K-Y Park, J. Kim,Software Architecture of JTAG Security System, WSEAS Transactions on Systems, Issue 8, Volume 11, pp. 398-408, August 2012.

[26] R. Weber, Editor's Comments:Still Desperately Seeking the IT Artifact, MIS Quarterly (27:2), June 2003, pp. iii-xi.