A Formal Scheme for Systematic Translation of Software Requirements to Source Code

ARBİ GHAZARIAN
Arizona State University
Department of Engineering
7171 E. Sonoran Arroyo Mall, Peralta Hall, Mesa, AZ 85212
USA
Arbi.Ghazarian@asu.edu

Abstract: Problem Decomposition Scheme (PDS) is a requirements specification framework that allows for consistent and objective decomposition and classification of software problems into categories of software requirements or problem dimensions. PDS-based problem dimensions represent recurring development problem types and as such, when paired with their optimal solutions, form reusable development patterns known as traceability patterns. Traceability patterns allow us to systematically transition from software requirements to source code. In this paper, we introduce an extension to the PDS formalism, called Systematic Translation Scheme (STS), which incorporates a sequence of traceable and objective transformations from software requirements to source code. We demonstrate that STS is capable of providing a formal foundation that allows for a rigorous description, analysis, and automation of traceability patterns.

Key–Words: Formal Development Model, Requirements Traceability, Software Patterns, Software Requirements Transformation, Traceability Patterns

1 Introduction

The discovery, documentation, and application of software patterns have become an important component of modern software development. Patterns play two major roles in software engineering:

- Patterns capture expert development knowledge in a reusable form thus helping novice practitioners to quickly come up with the same solutions as seasoned professionals.
- Patterns serve as a means of technical communication among software engineers. They become a part of the technical vocabulary of software practitioners.

In general, the capturing of software patterns involves two steps:

- Identification of recurring problem types in a context, and
- Provision of a common solution to the identified problem types.

In a previous work [3], we introduced a requirements specification framework called Problem Decomposition Scheme (PDS) that allows for consistent and objective decomposition and classification of software problems into categories of software requirements (a.k.a. problem dimensions). PDS-based problem dimensions represent frequently occurring functional software requirement types and therefore can be viewed as the realization of the first step in the capturing of patterns mentioned above.

What makes PDS-based problem dimensions excellent candidates for the formation of software patterns is their distinguishing property of application objectivity; in contrast to conventional patterns, where multiple patterns might be applicable to the same situation, PDS-based patterns are unique solutions to their corresponding problem dimensions. This determinism, which is an inherent property of PDS-based patterns allows us to view patterns as objective transformations from software requirements to source code components. A detailed description of the PDS framework can be found in [3].

PDS-based software patterns provide direct traceability from software requirements to source code components and as such they are commonly referred to as traceability patterns. Traceability patterns are a significant improvement over other forms of traceability such as the approaches that are based on building traceability matrices because they offer system-designed constructive traceability. That
Recent Researches in Mathematical Methods in Electrical Engineering and Computer Science

is, they help software engineers to design traceability into the software systems. The importance of traceability in software evolution and maintenance in well emphasized in software engineering (for example see [11] [8] [9] [10] [12] [13] [11]). A previous study [4] has demonstrated a strong relation between design regularity, traceability based on it, and software maintainability.

Traceability patterns are described informally or semi-formally through Unified Modeling Language (UML) diagrams or unified templates that use a combination of structured English, free text comments, and code snippets written in a programming language. Although such informal or semi-formal descriptions of traceability patterns are necessary in order to facilitate their application in industrial software development settings, the lack of a more rigorous formal foundation can impede their further advancement. The provision of a formal foundation for traceability patterns will allow for a more rigorous description, analysis, and automation of traceability patterns. This paper is a first step in providing such a formal foundation for traceability patterns.

The contributions of this paper is twofold: first, we introduce a formalism for software development called Systematic Translation Scheme (STS), which is capable of transforming software requirements into source code components. STS is an extension to the PDS formalism. Second, We demonstrate that STS provides a formalism for traceability patterns.

The rest of this paper is organized as follows: Section 2 is an overview of the PDS framework. In Section 3, we introduce an extension to PDS framework called STS that bridges the gap between the PDS-based software requirements specifications and their corresponding source code elements. Section 4 presents a discussion of how the formal elements in STS relate to traceability patterns. Conclusion and future work follow in Section 5.

2 The PDS Formalism

A Problem Decomposition Scheme (PDS) is a tuple $S = \langle R, T \rangle$ where $R$ is a set of requirements (i.e., a requirements specification for a development problem), and $T$ is a set of requirement types. $R$ is decomposed into classes of requirements based on the membership of its individual requirements in the categories of requirement types defined by $T$.

In the discussion that follows $R$ is the domain of discourse for $r$, and $T$ is the domain of discourse for $t_i$ and $t_j$. Below, we consider four properties of a PDS, namely the single-dimensionality, atomicity, completeness, and the well-definedness.

Single-Dimensionality of PDS

Let $S = \langle R, T \rangle$ be a PDS. $S$ is single-dimensional, if and only if every requirement $r$ of $R$ is a member of at most one category of requirements $t$ in $T$. Each element $t$ of $T$ represents a single dimension of the problem under study. This can be formally expressed as:

$$\exists r \in T_i \land r \in T_j \rightarrow T_i = T_j$$

(1)

Note that Formulae (3) below further constrains a PDS by requiring each requirement to have exactly one requirement type.

Atomicity of PDS

Let $S = \langle R, T \rangle$ be a PDS. $S$ is called atomic, if and only if no requirement $r$ of $R$ is decomposable into two or more requirements. In other words, $S$ is atomic if and only if every element $r$ of $R$ is atomic. Given that the atomicity of a requirement $r$ is denoted as atomic$(r)$, this can be formally expressed as:

$$\forall r \{\text{atomic}(r)\}$$

(2)

An atomic software requirement statement is a single indivisible statement about a software problem; they cannot be further broken down into two or more simpler statements. Atomic requirement cannot be implemented partially; they are either fully implemented, or not. Similarly, atomic requirements cannot be partially tested. The testing of an atomic requirement should result in either a clean pass or fail. Atomic requirements are the result of splitting the higher-level requirements into elementary, or indivisible, requirements.

Completeness of PDS

Let $S = \langle R, T \rangle$ be a PDS. $S$ is complete if and only if the set of requirements categories $T$ covers every single requirement in the set $R$ of requirements. In other words, every requirement $r$ of $R$ should qualify for membership in a category in $T$. This can be formally expressed as:

$$\forall r \exists t_i [r \in t_i]$$

(3)
Well-Definedness of PDS
A PDS is well-defined if and only if it is single-dimensional, atomic, and complete.

PDS is a simple yet powerful formal framework for the decomposition and classification of the problem space in software engineering endeavors. Defining software problems as PDS-based specifications can have a significant impact in the success of pattern-oriented approaches to software development. As stated in [3], "the motivation behind pattern-oriented software development is to decompose complex problems into recognizable sub-problems with predefined solutions, hence promoting both the quality of the resulting software product and the efficiency of the development process through the reuse of optimal solutions (e.g., best practices)."

In order to realize the full potential of a PDS-based pattern-oriented software development approach, we need to link categories of software problems or requirements types as defined in PDS to their corresponding solution elements in the solution space. In the next section, we will achieve this objective by introducing an extension to the PDS formalism that allows for the definition and enforcement of the architectural, design, and implementation rules that are intended to govern the design and source code components that form optimal solutions to the PDS-based problem categories. Throughout this paper, we use the term design rules or simply rules to refer to all kinds of development rules including the high-level architectural rules, detailed design rules, and implementation rules.

3 The STS Formalism

Before introducing the STS framework, we first need to consider the set-theoretic notion of a one-to-one correspondence.

Definition (One-to-one Correspondence)
Two sets $A$ and $B$ are said to be in one-to-one correspondence when we have a pairing of the elements of $A$ with the elements of $B$ such that each element of $A$ corresponds to one and only one element of $B$ and each element of $B$ corresponds to one and only one element of $A$ [2].

The source code components that make up a software system can be grouped into two broad categories: the application domain components and the infrastructural components. Domain components represent concepts from the problem domain of the software, whereas the infrastructural components represent concepts from the technological domain of the solution space. Domain components rely on the services provided by infrastructural components to achieve their domain, application, or business objectives. Modern software development heavily relies on the infrastructural services provided by open-source and commercial application frameworks, Application Programming Interfaces (APIs), and libraries. The incorporation of each such technology into a software system introduces a number of infrastructural components along with a set of development rules and naming conventions that must be conformed to. STS makes the distinction between domain and infrastructural components explicit.

In the discussion that follows $C_1$ is the set of design rules that define and govern domain software components in the solution space of a given software system. $c$, $c_1$, and $c_2$, which are themselves sets of design rules, are elements from the set $C_1$. Each member of set $C_1$ represent the set of design rules corresponding to one dimension or category of problems as defined by a corresponding PDS. $C_1$ is the domain of discourse for $c$, $c_1$, and $c_2$.

Some of the design rules in $C_1$ result in the creation of domain software components. The set of such software components is represented by $P_1$. $P_1$ is the domain of discourse for $d$, $d_1$, and $d_2$. The set of naming conventions and rules defined for the domain components in $P_1$ is represented by $N_1$. $N_1$ is the domain of discourse for $n$, $n_1$, and $n_2$.

$C_2$ is the set of design rules that define and govern the infrastructural components of the system. $C_2$ is the domain of discourse for $k$, $k_1$ and $k_2$. Some of the design rules in $C_2$ result in the creation of infrastructural software components. The set of such infrastructural software components is represented by $P_2$. $P_2$ is the domain of discourse for $e$, $e_1$, and $e_2$. The set of naming conventions and rules defined for the components of $P_2$ is represented by $N_2$. $N_2$ is the domain of discourse for $l$, $l_1$, and $l_2$. Examples of domain and infrastructure design and naming rules can be found in [5].

Satisfies(x,y) denotes that component rule x satisfies the problem category or requirements type y. Names(x,y) denotes that x names elements of component y.

With this definitions and assumptions in mind, we can now introduce a formalism for STS.

Systematic Translation Scheme
A Systematic Translation Scheme (STS) is a tuple
Recent Researches in Mathematical Methods in Electrical Engineering and Computer Science

$S = < R, T, C_1, P_1, N_1, C_2, P_2, N_2 >$ where $R$ is a set of requirements; $T$ is a set of requirement types (a.k.a. a requirements taxonomic scheme); $C_1$ and $C_2$ are sets of domain and infrastructure component rules, respectively. $P_1$ and $P_2$ are set of components created as a result of the application of the component rules $C_1$ and $C_2$. $N_1$ and $N_2$ are sets of naming rules for domain and infrastructure components (a.k.a. a naming scheme). An STS satisfies the well-definedness properties of a PDS along with the following conditions:

**Type-Component Rules Correspondence Condition**

\[ \forall c \exists t_i[Satisfies(c, t_i)] \quad (4) \]
\[ \forall t_i \exists c[Satisfies(c, t_i)] \quad (5) \]
\[ \exists c[Satisfies(c, t_i) \land Satisfies(c, t_j)] \rightarrow t_i = t_j \quad (6) \]
\[ \exists t_i[Satisfies(c_i, t_i) \land Satisfies(c_j, t_i)] \rightarrow c_i = c_j \quad (7) \]

This means that sets $T$ and $C_1$ are in one-to-one correspondence.

**Domain Component-Naming Correspondence Condition**

\[ \forall n \exists d[names(n, d)] \quad (8) \]
\[ \forall d \exists n[names(n, d)] \quad (9) \]
\[ \exists n[names(n, d_i) \land names(n, d_j)] \rightarrow d_i = d_j \quad (10) \]
\[ \exists d[names(n_i, d) \land names(n_j, d)] \rightarrow n_i = n_j \quad (11) \]

This means that sets $P_1$ and $N_1$ are in one-to-one correspondence.

**Infrastructure Component-Naming Correspondence Condition**

\[ \forall l \exists e[names(l, e)] \quad (12) \]
\[ \forall e \exists l[names(l, e)] \quad (13) \]
\[ \exists l[names(l, e_i) \land names(l, e_j)] \rightarrow e_i = e_j \quad (14) \]
\[ \exists e[names(l_i, e) \land names(l_j, e)] \rightarrow l_i = l_j \quad (15) \]

This means that sets $P_2$ and $N_2$ are in one-to-one correspondence.

Note that STS does not require a one-to-one correspondence between problem dimension and the infrastructural component rules. This is primarily because infrastructural components most often provide common services to the domain components and are not necessarily tied to specific domain objects.

Figure 1 depicts a schematic view of the layers of the STS framework. Each layer takes information from its previous layers and maps it into the next layer. With this in mind, STS can be viewed as a sequence of transformations that starts with a set of software requirements and moves toward a source code that satisfies these requirements. What is important here is that intermediate transformations can be objectively computed based on the information provided by the previous layers. In other words, the information generated in each layer is traceable to its predecessor and successor layers.

<table>
<thead>
<tr>
<th>N1, N2</th>
<th>Naming Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1, C2</td>
<td>Component Scheme</td>
</tr>
<tr>
<td>T</td>
<td>Taxonomic Scheme</td>
</tr>
<tr>
<td>R</td>
<td>Software Requirements</td>
</tr>
</tbody>
</table>

Figure 1: Layers of the Systematic Translation Scheme.

As depicted in Figure 1, the first two layers of the framework correspond to sets $R$ and $T$ of the formal model, respectively. The output of the second layer is a set of atomic and single-dimensional software requirements, which become the input to the third layer. The third layer corresponds to sets $C_1$ and $C_2$ of the formal model. In the third layer, these atomic and single-dimensional requirements are mapped to their component design rules. The dimension of the requirement is the key to this mapping. Layer three generates a set of software components. The fourth layer corresponds to sets $N_1$ and $N_2$ of the formal model. Layer four imposes naming rules on the components generated at level three. Each layer is directly traceable to its predecessor and successor layer.

The formal model specifies the layers of the framework, the contents (i.e., entities) of each layer specified in terms of the mathematical notion of sets, and the properties of individual entities (i.e., set elements) in each layer, as well as the relationships between entities in different layers specified in terms of mathematical logic.
4 Traceability Patterns and the STS Formalism

Figure 2 shows a conceptual model of the traceability patterns. This model was introduced in [6] and [7]. [6] defines Requirement-component traceability patterns as: “a mapping from a category of software requirements, classified under a requirement type, to a generic component structure that addresses that category of requirements”.

In Figure 2, the three concepts of "requirements taxonomic scheme", "requirement type", and "requirement" are formally described by the PDS framework. These three concepts relate to the first two layers in Figure 1. The Requirements taxonomic scheme that is used to classify software requirements is the set $T$ in the PDS formalism. Requirement types are members of set $T$. Requirements are described by set $R$. PDS requires that requirements be atomic and single-dimensional. The completeness condition in PDS guarantees that all the software requirements are classified by the categories of requirements in the taxonomic scheme. A well-defined PDS ensures that all the three conditions of a PDS are met and as such the problem space of the software problem to be solved has been properly taken care of.

The other concepts in Figure 2 are defined in the STS formalism. The generic components, which provide a solution to their corresponding requirement types are defined by sets $P_1$ and $P_2$ of STS. Components represented by $P_1$ are directly addressing the application or business domain of the software problem, whereas the components represented by $P_2$ are infrastructural components that come from the software technological domain, and are necessary to support the business or application components.

The architectural, design, and implementation rules that define and govern the business and infrastructural components are defined as sets $C_1$ and $C_2$ of STS; $C_1$ defines the rules for the domain components, whereas $C_2$ defines the rules for the infrastructural components. As depicted in Figure 2, these component rules can constrain various aspects of the design of the components including, the location of the components in the system, the content and structure of the components, and their naming conventions. Naming rules for the domain and infrastructural components are specified by sets $N_1$ and $N_2$ of STS, respectively.

The type-component rule correspondence condition of STS maps each category of problems (i.e., PDS requirements types) to a set of component rules. The domain component-naming and infrastructure component-naming rules maps domain and infrastruc-
tural components to their naming rules.

5 Evaluation

A catalog of traceability patterns was used to develop a proof-of-concept system. The catalog and the source code for this system can be obtained from [14]. While the resulting source code serves as an early evaluation for the traceability patterns, the STS framework provides a formal foundation for them.

6 Conclusion and Future Work

In this paper, we introduced an extension to the PDS formalism that is capable of mapping software requirements to source code components. We further demonstrated how this formalism provides a formal description for traceability patterns, which was informally introduced in a previous work [6]. As future work, we plan to further evaluate our framework by applying it to larger industrial software systems.

References: