ISR Institute for Software Research
University of California, Irvine

A Tagging-Based Approach for Eliciting Engineering Requirements in Established Domains

Leyna C. Cotran
Lockheed Martin Space Systems Company
leyna.c.cotran@lmco.com

Eric M. Dashofy
The Aerospace Corporation
edashofy@antconcepts.com

Richard N. Taylor
University of California, Irvine
taylor@uci.edu

December 2012
ISR Technical Report # UCI-ISR-12-12

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

A Tagging-Based Approach for Eliciting Engineering Requirements in Established Domains

Leyna C. Cotran
Lockheed Martin Space Systems Company
1111 Lockheed Martin Way
Sunnyvale, CA 94089 U.S.A.
leyna.c.cotran@lmco.com

Eric M. Dashofy
Affiliated with The Aerospace Corporation
2310 E. El Segundo Boulevard
El Segundo, CA 92045 U.S.A.
eric.m.dashofy@aero.org

Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697 U.S.A.
taylor@ics.uci.edu

***Abstract*— Most traditional requirements engineering and elicitation techniques assume that the requirements being developed are high-level requirements for a new, "green-field" system. However, this is not always the case. In established domains, systems are often developed as new members of existing product lines. In these cases, architectural design knowledge already exists within the product line, and the need is for detailed engineering requirements rather than high-level requirements. To address this situation, we propose the Co-Evolving Traceable Requirements and Architecture Network (COTRAN) technique, a novel collaborative requirements engineering technique that elicits detailed engineering requirements for new product line members through tagging architecture artifacts. We then discuss improvements in requirements generated using COTRAN on an industrial aerospace project.**

***Index Terms*—Requirements engineering, software architecture, aerospace, tagging.**

I. INTRODUCTION

Problems with software requirements are a leading cause of unsuccessful projects. If requirements are ambiguous, unverifiable, or do not accurately reflect stakeholder needs, they can be the source of costly downstream defects. If problems with the requirements go unrepaired, those costs multiply as each subsequent phase of the software development lifecycle proceeds [23]. Frederick Brooks, in [2], emphasizes that the biggest problem in developing a product is defining the requirements.

This is a long-recognized problem in software engineering, and many requirements elicitation techniques have been developed to address it [3][7][17][29][30]. The vast majority of these techniques make an assumption common in software engineering: that there is one canonical set of requirements, and that these requirements should be developed entirely within the "problem space"—divorced from notions of how the software will be later designed and implemented—issues considered part of the "solution space."

In many real-world domains these assumptions do not hold true. The aerospace domain is an excellent example of this for two reasons. First, in aerospace systems there are multiple kinds of requirements. High-level, mission-oriented requirements (that define what the system must do) are developed early in the systems acquisition process. These requirements most closely resemble traditional problem-oriented requirements. Interestingly, they may not even be captured in a formal requirements document, instead existing in less-formal concept of operations (CONOPS) documents or memoranda of understanding between customer and contractor. These high-level requirements are not used by engineers directly. Rather, a more detailed set of engineering requirements is developed, agreed upon, and used by engineers to design and implement the system.

These engineering requirements do not exist solely in the problem space. In the aerospace domain, systems are not designed and built from scratch. Though these systems are complex, much of the complexity has already been addressed in previous system designs—and thus much of these previous designs can be reused. This previous design experience represents a wealth of domain knowledge common to established domains like aerospace. Engineering requirements for aerospace systems are defined and interpreted with these reference architectures and designs in mind. They are also directed and informed by the high-level requirements, CONOPS, and other objectives and assumptions developed in early parts of the system acquisition process.

Surprisingly, even with so much information available, engineering requirements for aerospace systems tend to suffer from the same problems as any other kind of requirements: ambiguity, non-verifiability, inaccuracy, and so on. There are

a number of reasons for this: they are often developed by teams of engineers collaborating in an ad hoc, undirected way, relying primarily on their experience and judgment to invent requirements. Also, because new systems are often variants of old systems, these engineers often "copy and paste" poor requirements from old systems into new ones, perpetuating defects.

Consequently, there is a strong and obvious need for a more rigorous requirements elicitation technique to develop engineering requirements in the presence of rich domain and design data including previous system architectures, domain reference architectures, and high-level requirements. Unfortunately, as noted above, existing techniques are not targeted for this situation, and so they cannot be used.

We present the Co-Evolving Traceable Requirements and Architecture Network (COTRAN) technique [4] to fill this gap. COTRAN is a requirements engineering technique that focuses on eliciting engineering requirements from data found in a network of architecture artifacts. It employs a well-defined process for tagging these artifacts with metadata, a process by which engineers identify specific functional aspects of the design. This metadata, tempered by discussion and engineering judgment, is used by engineers to inform their process of defining specific engineering requirements for the system under development.

COTRAN has been piloted successfully in real-world, "live" aerospace projects. We discuss one of these projects and the lessons learned from it in Section IV.

This paper is organized as follows. Section II provides background on the co-evolution of architecture and requirements as well as the use of tagging. Section III discusses the COTRAN technique in greater detail. Section IV discusses an application of COTRAN in a live aerospace project. Section V provides discussion of the experience and findings about COTRAN. We conclude in Section VI and VII with related work and future directions for COTRAN.

II. BACKGROUND

The two most novel aspects of COTRAN are also the aspects where it diverges from traditional software engineering canon regarding requirements elicitation. The first is that COTRAN assumes that architectural software design will occur *before and during* the development of engineering requirements—as such, requirements and architecture are expected to co-evolve in COTRAN.

The second is that COTRAN uses a novel tagging-based approach to help engineers systematically explore, understand, and derive requirements from existing architectures. Tagging in COTRAN is inspired by the success of tagging in social media, a lightweight metadata elicitation technique. The simplicity of tagging makes it easy to learn and apply across heterogeneous kinds of architecture artifacts.

To provide context for a more extensive description of COTRAN in Section III, we describe these two aspects of the technique here.

A. Co-evolution of Architecture and Requirements

The co-evolution of requirements and design/architecture in software engineering is not a new concept [8][23][24]. It does, however, conflict with idealized software engineering notions (embodied in, e.g., the waterfall model [28]) that the problem and solution spaces can and should be cleanly separated in software development. An alternative to the waterfall approach that emphasizes requirements/ architecture co-evolution is the Twin Peaks model [7][15][24].

The general rationale for intertwining requirements and architecture development is well-described by Nuseibeh [7][15] and is too extensive to fully recapitulate here. At a high level, requirements inform the designs we will consider, but design knowledge can also constrain and shape requirements. In established domains, well-known architectures and styles to solve existing problems provide information that can be used to guide and simplify requirements.

Beyond this general rationale, however, there are excellent reasons to use a Twin Peaks-like approach in our particular circumstance: the development of engineering requirements in an established domain (aerospace). First, unlike high-level system requirements, engineering requirements tend to be functionally-oriented. They describe what the system must do, generally in enough detail that subsets of them can be "handed off" to engineers for direct design, implementation, and verification. Engineering requirements tend to be the functional realization of more abstract or non-functional higher-level requirements. Because of their functional nature, it makes intuitive sense to connect these requirements closely with the design elements that will be used to implement them.

Secondly, as we have discussed, strong designs (in the form of previous-system and reference architectures) already exist, and new designs will follow these patterns closely. Ideally, we would be able to directly reuse the engineering requirements that were developed along with these earlier designs—but often, these requirements were poor or problematic to begin with (and never fixed as the design evolved).

Extensive design reuse, whether intentionally or not, creates product-lines [5]: systems that share many points of commonality with well-defined points of variation. We believe that the development of product-line requirements, focusing on these variations, is far different from green-field requirements development.

An additional benefit of requirements-architecture co-evolution is that the resulting requirements and architecture are more closely connected than they would be if they were developed in isolation. The result is that both requirements and architecture "carry" more information into further development phases. For example, despite engineers' best efforts, requirements captured as "shall" statements can be ambiguous or hard to interpret. When a requirement is explicitly connected to architecture, however, the architecture provides additional contextual information that an implementer can use to understand the intent of the requirement. Complementarily, an engineer looking at an

element of the design can use a connected requirement to understand the "why" behind the design decision.

B. Tagging

Architecture artifacts are concrete expressions of the principal design decisions about a system. These design decisions may govern any aspect of the system—structure, behavior, timing, physical deployment, development organization, and so on. They may be expressed in many forms—usually in a combination of diagrams (whether informal or in a more rigorous notation such as UML) and prose text.

A challenge for engineers working on a new aerospace system is to identify, understand, and interpret the design decisions expressed in architecture artifacts—whether those artifacts exist already or are being developed/refined as requirements elicitation proceeds.

We believe that engineers can best understand and interpret these artifacts through a systematic, directed process. We have chosen a lightweight metadata development process, tagging, to stand in this role.

Tagging is a key part of many social media systems [9][14], wherein users examine data artifacts (such as photos, videos, and articles) and annotate them with simple keywords, or 'tags.' Depending on the system, the entire artifact (e.g., a photo) or only a part of the artifact (e.g., a specific person in the photo) may be tagged. Tags serve as metadata [19], and are generally rendered by social media software as clickable links. Clicking on the tag executes a search for other artifacts with the same tag applied. This creates a connection between artifacts (or artifact parts) that are tagged with the same keyword.

The use of tags is not limited to social media—it has also become increasingly popular in software development approaches [1][6][12][13][22][25]. [25] and [27] discuss the use of tags in organizing specific software tasks for ease of collaborative efforts amongst the team. The Jazz study [26] found that tags were used in aspects of software requirements development, specifically across functional and non-functional requirements. The tags focused on all aspects of software development.

In social media tagging (and in the above software engineering approaches), there is no pre-defined taxonomy of tags. Users can apply any tag they want to an artifact. The process of tagging creates an informal and evolving taxonomy [20], reinforced by subtle cues about how other users are applying tags [21].

In social media, where thousands of users act independently and incorrect or missed tags have few negative consequences, this strategy (using an emergent rather than well-defined taxonomy) is generally successful. However, the COTRAN context is different. Here, a relatively small group of engineers participates in tagging, so the phenomenon of many people establishing tagging trends does not occur. Also, the consequences of mis-tagging an artifact are higher (e.g., a requirement may be missed).

We use social media tagging as an inspiration, but apply a small amount of additional formality. As a first step, engineers develop a dictionary of tag types. This process is made easier by a rich, shared vocabulary drawn from engineers' knowledge about the target domain and the system under development. The engineers then tag artifacts with tags that are instances of the tag types defined in the dictionary. This process is discussed in more detail in Section III.D.

III. THE COTRAN TECHNIQUE

COTRAN is a technique to systematize how stakeholders develop engineering requirements for new systems in a product-line, in the context of an established domain, with the goal of developing more correct, precise, and verifiable requirements. In this paper and associated studies, we have focused on aerospace as the target domain, but we believe that the technique could be applied in similarly mature domains with product-line-based development.

We do not intend for COTRAN to be a strict process; rather, it is a set of ordered activities that are executed iteratively. The five activities that comprise COTRAN are:

1. Developing a dictionary of tag types;
2. Developing architecture artifacts;
3. Tagging architecture artifacts;
4. Developing requirements; and
5. Organizing data.

Each iteration of the technique may apply different amounts of emphasis to each individual activity, much like the Rational Unified Process [11]. As the system-to-be evolves, it is expected that these activities are revisited as necessary. Architecture decisions are likely to change during development. As artifacts become more concrete, all elements produced by the technique will be revisited over the project's lifetime.

A. Prerequisites and Inputs

Application of COTRAN has several prerequisites. First, we assume that the high-level requirements for the system have been defined [18]. These requirements may not be in formal requirements documents (e.g., one consisting of 'shall' statements); rather, they may be in informal concept-of-operations (CONOPS) documents or memoranda of understanding. These documents describe, at a high level, what functions the system must perform. Key performance parameters (KPPs) may also be established, identifying critical functions to be implemented or performance/timing targets the system must meet.

Objectives and assumptions may also be defined. These provide additional context for interpreting the high-level requirements, or may indirectly further define high-level system requirements.

In the aerospace domain, these steps generally occur as part of the bidding phase of the initial acquisition process. As such, when COTRAN is used in an aerospace context, it is generally used after this phase is completed. Other domains may not have an analogue to this phase providing a clear boundary when these activities have been completed.

Before COTRAN begins, a team must be established to execute the technique. The team should consist of one or more architects, requirement engineers, and knowledgeable domain

experts. This team must be well-versed in the domain. Although they will be focused on requirements elicitation and development, they will also make architectural design decisions.

B. Activity 1: Developing a dictionary of tag types

The goal of this activity is for the COTRAN team to develop or refine a shared dictionary of tag types. As noted above in Section II.B, tagging in COTRAN (Activity 3) resembles social media tagging, but tags are drawn from a pre-defined taxonomy. This activity involves the COTRAN team collaborating to define that taxonomy.

In COTRAN, tag types and tags have a types-and-instances relationship, similar to classes and objects in an object-oriented programming language. A tag type identifies a semantically meaningful concept from the domain or system under development. A tag instance (or just 'tag') of that type annotates a specific part of the architecture that embodies (or is otherwise closely related to) that concept.

For example, an important concept in the aerospace domain is the concept of 'telemetry.' Many parts of an aerospace system's architecture are involved in the generation, transmission, analysis, and storage of telemetry. These architectural elements would be annotated with tags of the tag type 'telemetry.' How the tagging activity occurs is detailed below in Section III.D.

In this activity, all members of the team collaboratively identify and define tag types. They do so primarily by enumerating important concepts from the domain or the system under development. The concepts enumerated can come from:

- Their own experiences as domain experts;
- Canonical references in the domain;
- The high-level system requirements, CONOPS, objectives, and assumptions; and
- The existing architecture artifacts, or architectures of analogous/similar systems.

After concepts are enumerated, they must be further elaborated. A tag type consists of more than just the concept name. Each tag type is elaborated with the following data:

TABLE I. TAG TYPE DATA.

Element	Rationale
Name	A short word or phrase identifying the concept (e.g., 'telemetry').
Definition	Text or formal statements defining the concept—explaining what it is and its role in the domain or target system.
Guidance	Documentation describing when to tag something with this tag type.

The definition helps establish the scope of the tag type—what it means, and what it does not mean. The guidance is documentation of agreements between the team members on when the tag type should be used.

An obvious question is raised: how do engineers develop guidance for their own use later, especially in early iterations? This question is apt, and we believe the iterative nature of COTRAN provides opportunities for substantial refinement of this guidance as engineers gain more experience with the system architecture.

However, one technique that can be used in early iterations is to identify a "conceptual cluster" of nouns, verbs, and noun-verb combinations related to the tag type's core concept. For example, if the core concept is 'telemetry,' the domain provides a rich set of related nouns and verbs. Nouns include 'telemetry data,' 'telemetry database,' 'telemetry format,' 'telemetry source,' 'telemetry sink' and so on. Verbs include telemetry creation, storage, transmission, reception, conversion, and so on. In our experience, instances of any of these nouns or verbs in the architecture artifacts should all be tagged with tags of the type 'telemetry.'

Tag types are collected and written down in an artifact we call the Tag Type Dictionary. This dictionary becomes an evolving, shared reference used in Activity 3 (tagging).

C. Activity 2: Developing Architecture Artifacts

The goal of this activity is for the team to select, develop, and refine architecture artifacts. As noted in Section II, architecture artifacts are concrete expressions of the principal design decisions about a system, usually in the form of diagrams and text.

COTRAN is targeted at eliciting engineering requirements for new systems in an existing product line [5]. As such, we assume that many architecture artifacts already exist—from previous versions of the system, other product-line members, or reference architectures.

Throughout system development, these architecture artifacts will evolve—existing artifacts will be modified and new artifacts will be created. If high-quality architecture artifacts exist at the start of development, these can be largely reused with judicious modification to take into account new or changed high-level requirements. However, it is also possible that pre-existing artifacts are low-quality (inadequate, unmaintained/divergent from implementations, incorrect, too ambiguous, too informal). In these cases, an effort to "clean up" or develop architecture artifacts should be undertaken before the COTRAN technique begins.

D. Activity 3: Tagging Architecture Artifacts

The goal of this activity is for the team to tag the architecture artifacts with tags that are instances of the tag types defined in the tag type dictionary.

Tagging is fundamentally a mapping activity: taking the concepts from the tag type dictionary and mapping those to instances of those concepts in the architecture artifacts. There is no single required way to perform the tagging activity. Individual COTRAN team members can create this mapping in the manner they find most effective.

There are two obvious strategies. The first is to iterate through the tag types in the dictionary—selecting a tag type, then looking for instances of that concept in all the architecture artifacts. The second is to iterate through the elements (diagrammatic elements, sentences/paragraphs) in the architecture artifacts, and then select appropriate tags from the tag type dictionary.

Which strategy is most effective depends somewhat on how the work of tagging is partitioned among the team members. Here again, there are two obvious strategies: individual team members may be assigned a subset of tags or a subset of architecture artifacts. We have found that when team members are subject-matter experts in narrow aspects of the domain, assigning them tag types related to their expertise is effective. This would, for example, mean assigning the 'telemetry' tag type to the engineer most familiar with telemetry, and then asking this engineer to find all elements/design decisions in the architecture artifacts to tag with 'telemetry' tags.

The guidance associated with the tag type in the dictionary is used extensively during tagging. Team members engaged in the tagging activity use this guidance to determine whether to tag a particular architecture element with a given tag type. Consider our earlier example of the 'telemetry' tag type and the associated noun/verb-based guidance. A team member assigned to use this tag might look at a structural (components-and-connectors) diagram of a system, find any component or connector that creates, transmits, or stores telemetry, and then tag it.

One additional aspect of tagging unique is that tag instances are distinguished from each other. This is an important and subtle aspect of the technique. In our running example, different elements in the architecture may be tagged with tags of the type 'telemetry.' We noted that components and connectors that create, transmit, or store telemetry would be tagged this way. However, we also want to distinguish these distinct aspects/uses of telemetry in the architecture from each other. So, each instance of the 'telemetry' tag type is annotated with a unique ID number. As a hypothetical example, consider three tag instances that might occur, all of the tag type 'telemetry:'

- **Telemetry.01:** The creation of position telemetry for the rocket.
- **Telemetry.02:** The transmission of position telemetry for the rocket.
- **Telemetry.03:** The storage of position telemetry for the rocket.

Distinguishing different tag instances from one another becomes useful later when using the tags to help formulate requirements, as we discuss below.

Note, however, the same tag occurrence (e.g., Telemetry.01) may appear in multiple architecture artifacts. One technique used frequently in architectural modeling is the use of multiple *views*—descriptions of the same aspects of the architecture from multiple perspectives [10]. So, a logical view may describe the software component that creates position telemetry data (Telemetry.01). A physical view may describe the physical sensors used to create that position telemetry data. These elements are *also* tagged Telemetry.01, because they capture the same aspect of the architecture—just from a different perspective.

Tagging can be done by individuals, or collaboratively as a team. Ultimately, the tagging process has two aims. The first is to generate tagged architecture artifacts, but this is the lesser aim. The greater aim is to inspire collaboration and negotiations between the team members, such that they come to a *shared* understanding of the architecture—and from that, the requirements. As described below, the system requirements cannot be directly derived from tagged architecture artifacts. Rather, they come from the engineering judgment of the individual engineers on the COTRAN team. As such, the tagging *collaboration process* is just as responsible for their developing a set of consistent and complete requirements as the actual artifacts themselves.

Because of this, in our experience, a combination of both individual and collaborative tagging works best. Many team members will want to spend time alone understanding and interpreting the artifacts, but it is critical that team members explain their tagging rationale to each other and ensure that the team as a whole is using tags consistently.

E. Activity 4: Developing Requirements

Requirements are not derived directly from tags. Rather, the tagging process is an intellectual exercise that generates discussion among the team developing the requirements. The requirements are generated by the team through a creative process informed by the tagged architecture artifacts.

Different requirements elicitation techniques can result in different requirement formats. The format for requirements may also be driven by external factors: for example, in aerospace and defense-related projects, requirements are often captured in English text sentences for contractual reasons. These sentences are colloquially called 'shall' statements because they are of the form "The X (part of the system) shall Y (expected behavior)." The project stakeholders must agree on the format for requirements independently; COTRAN prescribes no specific format.

The tags, and associated discussion, inspire engineers to develop requirements. One systematic strategy for requirements development is to look at each tag and attempt to elicit one or more requirements related to it. Earlier, we defined a hypothetical tag instance "Telemetry.02: The transmission of position telemetry for the rocket." This tag instance may occur in multiple places. Each annotation may inspire a new requirement statement, as shown in Table II.

TABLE II. HYPOTHETICAL REQUIREMENTS.

Instance	'Shall' Statement
On a software connector	Connector *Conn1* shall transmit rocket telemetry from Component *Comp1* to *Comp2*.
On a physical connection	The *Network1* network shall transmit rocket telemetry from the Sensor component and the Antenna component.
On a timing diagram	Rocket telemetry shall be transmitted between the sensor and antenna components in no more than 20ms.
On an activity diagram	Transmission of a rocket-telemetry packet shall always be followed by a rocket-telemetry-complete packet.

The noun/verb guidance associated with the underlying tag type can also be used to elicit potential requirement statements—many 'shall' statements, for example, are in the form "The (noun) shall (verb) the (noun)."

One characteristic of good requirements is that they are verifiable, especially for functional requirements. This is true no matter which elicitation technique is used. COTRAN prescribes that each proposed requirement must be associated with verification success criteria (VSC). The team defines VSCs using statements that explain what must be done in order for the requirement to successfully be verified.

F. Organizing Data

The goal of this activity is to collect, organize, and link data items and artifacts created during COTRAN. The overall collection of data and artifacts in COTRAN is called a *repository*. The repository contains:

- The dictionary of tag types
- The (tagged) architecture artifacts
- The correlated requirements
- The VSCs for each requirement.

One key aspect of assembling the repository is associating requirements with tag instances in the architecture artifacts. As noted earlier, there is not necessarily a direct correspondence between tag instances and requirements. However, in many cases (including our earlier examples) a correlation can be established. This correlation should be captured along with the requirement. In this way, stakeholders examining the requirements and architecture artifacts later can understand where the requirements came from, and how they are reflected in the architecture.

IV. Industrial Experience

The COTRAN technique was applied on an industrial project in the aerospace domain. The goal of this experience was to apply the COTRAN technique to develop detailed engineering requirements in a relevant target field on an actual, "live" project and gain insight as to its effectiveness.

The project was to develop a new system in a product line, where each member of the product line is born from a common architecture. Each new member varies from the others based on customer and mission needs. The common architecture is a plug-and-play testbed that can be used to test components under different conditions for potential survivability in space. These components include avionic components and guidance components that are software-intensive.

Many product-line members (and their associated detailed engineering requirements) had been developed without COTRAN. The status quo procedure for doing so involved direct reuse of existing requirements and architecture artifacts from the common architecture, modifying them for the new product-line member's needs. No rigorous process was used to do so—engineers and other stakeholders made these modifications in the way that they see fit. A single team persists across development of product-line members.

A critical problem for the continued expansion and health of this product line is that, while previous requirements specifications and architecture artifacts exist, they have issues that make their direct reuse a liability.

We analyzed the existing product-line requirements specification and found that 60% of the stated requirements were not testable. These included requirements that pointed to other reference documents or government standards, were copied and pasted from a parent source that was retired off the specification list of the project, or were focused on following a process (e.g., "shall obtain a manager's signature") rather than being technical requirements. The team felt that 17% of the requirements could be carried over to the current project's specification, but this 17% needed architectural context to be fully interpreted.

We found that the existing architecture artifacts needed to be elaborated and the set of artifacts needed expansion to capture the details necessary to fully describe a new member of the product line.

Twenty people were assigned to this project to create a new product-line member. Seventeen of these people worked on several previous members of the same product line. Fourteen of them worked on ten or more product-line members overall. The team was composed of senior people with deep domain knowledge. It included architects from various aspects of the project: two electrical architects, one software architect, and two system architects.

Team members had access to ample domain knowledge for building new members of this product line. In addition to numerous publications and articles, the team members had a deep understanding of the domain from their experience in building many previous product-line members. The new components integrated into new product-line members are largely commercial or public government products that were also heavily documented.

A. Project Settings

The kickoff for this project started in bidding discussions, and the project team was in place and funded three months later. During this period, several technical discussions were held identifying potential objectives, assumptions, and risks involved with the project. Objectives and assumptions were documented and revised from the project's kickoff. They underwent two significant revisions before the customer requirements review ten months into the project. The team was fully involved in these discussions and contributed to these revisions.

The use of COTRAN was focused on the functional and performance requirements related to the avionics software system, the telemetry system, and the electrical power system that was controlled by the avionics software. These three subsystems were deeply interdependent.

This paper's first author participated in this project as the requirements engineer. The COTRAN technique was introduced to the team over several meetings. Team members that had questions regarding the technique required individual meetings.

B. Tag Types and the Tag Type Dictionary

Our previous work [4] was provided to team members in order to understand tag types, their creation, and the structure

of the tag type dictionary. Examples of tagging approaches from social media were also discussed.

As noted earlier, there is no strict process by which the team must collaboratively develop tag types. In this particular case, the team members preferred to begin with independent sessions, defining tag types individually before working together. Initial development of tag types came primarily from examination of existing architecture artifacts (rather than, e.g., asking each team member to enumerate important domain concepts). This provided the main jumping-off point for discussions about tag types with team members (both individually and as a group) facilitated by the requirements engineer.

In the tag type discussions, the requirements engineer asked each team member to describe what he or she believed his or her portion of the design entailed. Each team member described concepts that were either based on domain understanding (e.g., "We need to follow these pre-defined trajectories"), architectural decisions (e.g., "the avionics software must pass location data through the bus interface, and the data must be encrypted"), or on design principles (e.g., "the software must use the navigation coordinates that get uploaded by a command"). Based on these responses, the requirements engineer directed the team members to find the information in the architecture artifacts that reflected these statements. In some cases, team members would sketch additional context diagrams to help the discussion and to explain and defend potential tag types.

As these early statements and the associated architectural information were enumerated and documented, team members began to pick key words from the responses as their tag types, and then create definitions for the tag types. As the tag type dictionary grew, additional team members became involved in collaborating on tag types and their definitions. Team members also started to define tag types for other team members.

Some team members noticed similarities and overlap between tag types and began consolidating tag types they had defined with those defined by other team members. Often, this required a semantic expansion of existing tag types to take into account the perspectives of multiple stakeholders. A specific example occurred with the *telemetry* tag type. Telemetry had different architectural implications for the software and electrical systems of the project, so the definition and associated guidance were expanded to include telemetry as both a software concept and an electrical concept. Ultimately, as intended, the tag types served as a shared vocabulary for the team.

C. Creating and Refining the Architecture Artifacts

Architecture artifacts from previous product-line members and the common architecture existed and were part of the discussions throughout the development of the requirements. During the application of COTRAN, 8 architecture artifacts were reused and updated, and 10 were created anew.

The team expanded the set of architecture artifacts as they developed tag types. For example, the electrical group decided to separate their electrical data flow diagram to deal with the different configurations of the project. An N-squared diagram was created and shared among the electrical and avionics software team. The avionics experts added a context diagram, a modes diagram, and several message sequence diagrams. The software experts added several software context diagrams, largely to illustrate facilitated discussions about tag types with the requirements engineer.

TABLE III. INDUSTRY STUDY ARTIFACT SUMMARY.

Architecture Artifact	Responsible Group
System Context Diagram	Electrical, Software, Telemetry
Data Block Diagram per configuration (4 total)	Electrical, Software
Electrical Block Diagram per configuration (4 total)	Electrical
Message Sequence Chart	Software
Modes Diagram	Software
Avionics Dataflow Diagram	Software
N-squared Diagram per configuration (4 total)	Electrical, Telemetry, Software
Software Context Diagram	Software, Telemetry
Software Time Line	Software, Telemetry
Total Artifacts	18

Table III enumerates the resulting 18 architecture artifacts for this project. These artifacts were shared by the team, meaning that team members with different technical backgrounds referred to the same artifacts during discussions. This was helpful in enabling a consistent understanding of the project among the team members.

D. Tagging Architecture Artifacts

As expected, the tagging activity was extensive. The avionics software group iterated through their tag types and tagged artifacts over a two-month period. During this period, they made several changes to their tag types, some of which resulted in additional sharing of tag types with other groups. The group found that the software architecture artifacts were useful for overall system discussions, and other team members leveraged these artifacts for their own respective tag types and tagging approaches.

The electrical group spent several weeks tagging their artifacts. From their electrical data diagrams, the group created additional N-squared diagrams capturing associations between elements. This was especially helpful for tagging, as the electrical group's tag types correlated with the data interfaces that occurred between components.

As expected in the COTRAN technique, the tagging activities inspired many discussions among the group, particularly about where to place tags and what aspects of the architecture the tags represented. The team as a whole found that the architecture artifacts were useful for overall system discussions, and different team members adapted their own tagging approaches and use of tag types while watching each other tag.

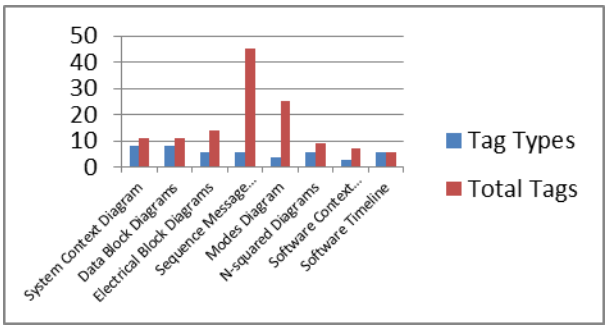

Fig. 1. Tags Summary.

Error! Reference source not found. shows the correlation of tag types and tags among the different artifacts. We observed that the N-squared diagrams were the most difficult to tag. Each N-squared diagram documented roughly 40-50 interactions, multiplied over four N-squared diagrams for different configurations (so, nearly 200 total interactions). Not all these interactions were tagged. Some of them were internal to particular components or subsystems. For example, the telemetry subsystem used a standard commercially-available processor. Interactions within the processor were already documented and not something that would be implemented or changeable by engineers on this project. Because of this, there was no way the interactions could influence system requirements, and so they were not tagged.

The context diagrams were useful for broad discussions about the system, but the details needed for tagging existed in more detailed artifacts. Often, in discussions with team members, the system context diagram was one of the last artifacts to be reviewed for potential tags.

The avionics software group's modes diagram, message sequence diagrams, and electrical block diagrams were the most decorated with tags. These diagrams had particularly dense information about system functionality and behavior, so they contained a substantial amount of taggable information. The modes diagrams captured 9 modes and 4 submodes with 14 transitional behaviors and 4-6 sub-transitions. The sequence diagrams captured 4 major activities and 36 individual events. However, not all detailed diagrams were densely tagged—the four electronic block diagrams described 30-36 blocks and 42-47 interactions, but had relatively few tags.

E. Development of Requirements

Several meetings were held to discuss requirements statements. Tags in the different artifacts helped facilitate these discussions and guide the team members to develop specific requirements statements.

In tag types that were closely related, the participants would suggest potential requirement statements for other team members, even when they were not present at meetings. Team members that shared tags appeared to have a shared understanding of tag type terms, and felt that they could contribute to other team member requirement statements.

F. Development of Outputs

There were several outputs produced for the project. These outputs included two requirement specifications: a system-level specification and an avionics software-specific specification. The system-level specification contained performance and behavior requirements that emerged from the previously-described use of COTRAN, as well as some additional logistical requirements mandated by the government (which were not technical and were not developed as part of the use of COTRAN). The avionics software specification was produced from the requirements statements developed through the COTRAN technique.

Once requirements were codified, the team developed verification success criteria for each requirement. These were then mapped to a set of standard verification methods (e.g., analysis, demonstration, inspection, test) that are customarily used in the aerospace domain. The specific verification success criteria, however, were far more specific than these high-level methods.

These outputs were presented at a formal customer requirements review meeting during the first year of the project. The customer was given a copy of both requirements specifications prior to the review. At the review, the participants presented their respective requirements and discussed the verification approaches for each requirement. The presentations contained snapshots of the requirements. Upon completion of the requirements review, the customer signed off on the requirements—that is, the review was successful.

V. DISCUSSION

The COTRAN technique resulted in demonstrable improvements in the requirements for this new product-line member when compared to previously-developed members.

As discussed earlier, the existing common requirements were 60% untestable. Of the requirements that resulted from the use of COTRAN, 66% were verifiable as written. 11% needed to be slightly reworded for clarity. 7% remained untestable as written because they were carried over from the previous specification and could not be negotiated out. The remaining 16% were requirements reused from the previous specification discussed earlier in this section and were testable and mapped to the architecture artifacts. Table IV summarizes the previous project requirements and the new project requirements.

TABLE IV. REQUIREMENTS METRICS.

Category	Before COTRAN	After COTRAN
Verifiable requirements	23%	66%
Verifiable requirements, slight clarity needed in wording	17%	11%
Unverifiable requirements	60%	7%
Reused from previous projects	--	16%

In addition to improvements in the requirements, this experience also resulted in substantial improvements to architecture artifacts. As noted earlier, many new architecture

artifacts were developed, providing additional information for engineers that will eventually be tasked to implement this system, as well as providing additional project documentation. Furthermore, all architecture artifacts—both reused and new—are tagged, and individual tagged data elements are mapped to requirements. This provides a wealth of new information for stakeholders trying to interpret the requirements and the architecture artifacts themselves.

As a final note, the experience also resulted in events that demonstrated areas where COTRAN would not be useful. Other groups on the same project focused on logistics and facility operations were primarily interested in requirements regarding compliance with government standards rather than technical development or engineering to satisfying customer needs. The COTRAN technique was not a good fit for these particular groups because they did not generate requirements related to function or behaviors of the product, and did not generate architecture artifacts to tag.

VI. Related Work

Various existing techniques are related or similar to the COTRAN technique. As we have described, COTRAN differs from most requirements engineering techniques because of differing assumptions: that architecture artifacts co-evolve with the requirements and that the goal is to develop detailed, rather than high-level, requirements for a system. A few requirements engineering techniques are more closely related and deserve special mention.

SEI's SQUARE [15] is a requirements engineering technique targeted for security requirements in software-intensive systems. SQUARE and COTRAN are similar in that both approaches create common terms for the project and establish artifacts. COTRAN differs significantly from SQUARE in that the participants establish specific project terms based on the architecture artifacts that pre-exist, the vocabulary is born from tag types in the architecture artifacts, and specific guidance is created to apply tags across artifacts and build requirements statements. COTRAN is more explicit about linking to artifacts through the use of tags and promoting specific guidance to creating correlating requirements statements. SQUARE does not explicitly tie requirements and architectures, and assumes that artifacts are created first, and requirements created second.

An interesting question is COTRAN's relationship to agile methods. COTRAN relies heavily on the existence of architecture documentation to facilitate tagging. Agile approaches eschew much of this documentation, focusing on lighter-weight approaches for requirements elicitation such as user stories [3]. In a way, user stories and COTRAN tags are analogous in that they represent a small bit of information that guides later implementation, but there is no direct analogy between the two.

UML and SysML offer alternative approaches to requirements capture in the form of use case diagrams and requirements diagrams, respectively. Both are simply notations and do not prescribe a process for eliciting requirements. These diagrams could be used as alternatives to lists of 'shall' statements as requirements capture techniques as part of COTRAN.

VII. Conclusions and Future Work

The COTRAN technique leverages a collaborative tagging process of architecture artifacts to elicit detailed engineering requirements for new members of an existing product line in a mature domain. It was applied in a live aerospace project. It is quite simple, collaborative, largely informal, but amazingly effective in credible industrial practice

This application resulted in a number of benefits. First and foremost, it resulted in a measurable improvement in the testability of the generated requirements compared to previous product line member requirements developed without COTRAN. Second, it fostered the development of additional architecture artifacts that would not have been otherwise developed, providing additional project documentation as well as design information that can help stakeholders to interpret the developed requirements.

Both existing and new architecture artifacts were tagged with domain- and system-specific terms drawn from a shared dictionary developed by the project team. The process of collaboratively developing the dictionary and tagging the artifacts created dialogue between the team members. Participants discussed, rationalized, and debated tags and tag types. In several cases, tag types were shared among stakeholders that had common concerns, and participants placed tags for others and created correlating requirements statements. The resulting tag type dictionary provided a rigorous and agreed-upon vocabulary for discussing the system that did not previously exist. Positive feedback was received from the avionics software lead via email, describing COTRAN as "very powerful" for promoting a detailed understanding of requirements.

Going forward, this research will focus on further refining the guidance for team members in defining tag types and applying tags. We also want to investigate how to integrate more rigorous and formal requirements verification techniques for developing effective verification success criteria (VSCs) such as those described by Whalen et al. [31].

Acknowledgments

This work is supported in part by the National Science Foundation under grants CCF-0917129 and CCF-0820222.

References

[1] M. Ames and M. Naaman. 2007. Why we tag: motivations for annotation in mobile and online media. In Proc. SIGCHI Conf. Human Factors in Comp. Sys. (San Jose, California, USA, April 28 - May 03, 2007). CHI '07. ACM, New York, NY, 971-980.

[2] F. P. Brooks, Jr. 1995. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Mike Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[4] L. C. Cotran, R. N. Taylor. Developing requirements in an established domain using tags and metadata. Requirements Engineering Sys. of Sys. Workshop, 2011. pp.24-27

[5] P. Clements, L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2005.

[6] S. Golder, B. Huberman. The structure of collaborative tagging systems. HP Labs technical report, 2005.

[7] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. 2002. Relating Software Requirements and Architectures Using Problem Frames. In *Proc. 10th Ann. IEEE Joint Intl. Conf. Requirements Engineering (RE'02).* IEEE Comp. Soc., Washington, DC, USA. 137-144.

[8] A. Jansen, J. Bosch, P. Avgeriou, Documenting after the fact: Recovering architectural design decisions, Journal of Sys. and Software, Volume 81, Issue 4, April 2008, Pages 536-557.

[9] L. Kennedy, M. Naaman, S. Ahern, R. Nair, and T. Rattenbury. 2007. How Flickr helps us make sense of the world: Context and content in community-contributed media collections. In *Proc. 15th Intl. Conf. Multimedia* (MULTIMEDIA '07). ACM, New York, NY, USA, 631-640.

[10] P. Kruchten, "The 4+1 View Model of Architecture," IEEE Software, 12 (6), pp. 45-50, 1995.

[11] P. Kruchten. 2003. The Rational Unified Process: An Introduction. 3rd ed. Boston: Addison-Wesley Professional

[12] G. MacGregor, E. McCulloch. Collaborative Tagging as a Knowledge Organisation and Resource Discovery Tool. Library Rev., 55 (5) (27 February 2006)

[13] A. F. Marvasti and D. B. Skillicorn. 2010. Structures in collaborative tagging: an empirical analysis. In *Proc. 33rd Australiasian Conf. Comp. Sci. – Vol. 102* (ACSC '10), Australian Comp. Soc., Inc., Darlinghurst, Australia 109-116.

[14] C. Marlow, M. Naaman, D. Boyd, and M. Davis, 2006. HT06, tagging paper, taxonomy, Flickr, academic article, ToRead. In *Proc. 17th Conf. Hypertext and Hypermedia (HYPERTEXT '06).* ACM, New York, NY, USA. 31-40.

[15] N. Meade, E. Hough, and T. Stehney II. "Security Quality Requirements Engineering." Tech. Rep. CMU/SEI-2005-TR-009. Carnegie Mellon University. November 2005.

[16] B. Nuseibeh, "Weaving the Software Development Process between Requirements and Architectures", in *Proc. ICSE 2001 STRAW Workshop*, Toronto, 1996.

[17] B. Nuseibeh and S. Easterbrook. 2000. Requirements engineering: a roadmap. In *Proc. Conf. Future of Softw. Eng.* (ICSE '00). ACM, New York, NY, USA, 35-46.

[18] H. Ossher, D. Amid, A. Anaby-Tavor, R. Bellamy, M. Callery, M. Desmond, J. De Vries, A. Fisher, S. Krasikov, I. Simmonds, and C. Swart. 2009. Using tagging to identify and organize concerns during pre-requirements analysis. In *Proc. 2009 ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design (EA'09)*. IEEE Comp. Soc., Washington, DC, USA, 25-30.

[19] R. Sarvas, E. Herrarte, A. Wilhel, and M. Davis. 2004. Metadata creation system for mobile images. In *Proc. 2nd Intl. Conf. Mobile Systems, Applications, and Services* (MobiSys '04). ACM, New York, NY, USA, 36-48.

[20] S. Sen, S. K. Lam, Al-M. Rashid, D. Cosly, D. Frankowski, J. Osterhouse, F. M. Harper, and J. Riedl. 2006. Tagging, communities, vocabulary, evolution. In *Proc. 2006 20th Ann. Conf. Computer Suppoerted Cooperative Work (CSCW '06).* ACM, New York, NY, USA 181-190.

[21] C. Shirky. *Here comes everybody: The power of organizing without organizations*. Penguin Books, 2008.

[22] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. 2006. Shared waypoints and social tagging to support collaboration in software development. In *Proc. 2006 20th Ann. Conf. Computer Supported Cooperative Work* (CSCW '06). ACM, New York, NY, USA, 195-198.

[23] R. N. Taylor and A. van der Hoek. 2007. Software design and Architecture: The once and future focus of software engineering. In *2007 Future of Softw. Eng.* (FOSE '07). IEEE Comp. Soc., Washington, DC, USA, 226-243.

[24] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architectures: Foundations, Theory, and Practice*. John Wiley & Sons, 2008.

[25] C. Treude and M.-A. Storey. 2009. How tagging helps bridge the gap between social and technical aspects in software development. In *Proc. 31st Intl. Conf. Softw. Eng.* (ICSE '09). IEEE Comp. Soc., Washington, DC, USA, 12-22.

[26] C. Treude and M.-A. Storey. 2012. Work item tagging: Communicating concerns in collaborative software development. *IEEE Trans. Softw. Eng.* 38, 1 (January 2012), 19-34.

[27] C. Treude and M.-A. Storey. 2010. The implications of how we tag software artifacts: exploring different schemata and metadata for tags. In *Proc. 1st Workshop on Web 2.0. for Softw. Eng. (Web2SE'10).* ACM, New York, NY, USA, 12-13.

[28] W. Royce, Managing the development of large software systems. In *IEEE WESCON*, pages 1-9. IEEE, Aug. 1970.

[29] A. Van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley & Sons, 2007.

[30] A. Van Lamsweerde. 2001. Goal-oriented requirements engineering: A guided tour. In *Proc. Fifth IEEE Intl. Symp. Reqts. Eng.* (RE '01). IEEE Comp. Soc., Washington, DC, USA, 249.

[31] M. Whalen, A. Rajan, M. Heimdahl, S. Miller. Coverage Metrics for Requriements-based Testing. In *Proc. 2006 Intl. Symp. Softw. Testing and Analysis*, Portland, ME. pp. 25-36