TORUS: TRACING COMPLEX REQUIREMENTS FOR LARGE CYBER-PHYSICAL SYSTEMS

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER AND INFORMATION SCIENCES

Supervisors
Dr. Roopak Sinha
Professor Enrico Haemmerle

July 2016

By
Barry Dowdeswell
School of Engineering, Computer and Mathematical Sciences

Copyright

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

[illegible]

Signature of candidate

Acknowledgements

A thesis is one of the few visible outcomes of the journey of a research student. Buried deep within one of the papers I read while researching my topic was this quotation from the French impressionist painter Henri Matisse (1869-1954):

"I have always tried to hide my own efforts and wished my works to have the lightness and joyousness of a springtime which never lets anyone suspect the work that went into them. But I fear young painters seeing them will ignore the slow and painful work on drawing that is indispensable."

Research allowed me to spend so much precious, enjoyable time alone, wrestling with concepts that often frustrated but always fascinated me. I grew tired, but never *tired of*, this work. Yet in the midst of that solitude, I found myself collaborating with other students and staff who have become firm friends.

Figure 1: Woman Reading (Matisse, 1894)

My wife, Jane, has always been my most precious confidante and best-friend. Thank you so much for sharing this new journey with me. We spent twenty-seven good years working side-by-side in the same company. Now, our new careers seem to be opening up so many exhilarating opportunities to share together.

Dr. Roopak Sinha, thank you for being my thesis supervisor. Your teaching and mentoring is what first led me to begin studying these tiny computers that fascinate us both. Your guidance, inspiration and friendship has made this time and research so much richer.

My son Benjamin, his fiancé Emily, my daughter Michaela and her fiancé Luke; thank you all so much for being fellow students with me and for all your proof-reading. You are the best friends and study-buddies I could ever wish for.

My son Philip, my late-night study companion. Some of my most productive times were at 2:30 am in the morning in my laboratory, while you were working late in the city. I loved those times with you (especially the early-morning hamburgers) far more than you realize.

Professor Enrico Haemmerle also kindly stepped in to become my second supervisor after Dr. James Skene. Thank you so both for your input into this work and your kindness.

Dr. Jennifer Gibb wrote part of her PhD about the company Jane and I founded. Jenny then stayed on as a precious friend and mentor after that time. Thank you for all the hours drinking coffee together as you helped me make sense of academic life. I am so grateful for your friendship and kindness.

I could not leave out Ramon Lewis and Saide Lo. You both always seem to know where *everything is* here, which has been immensely comforting whenever I was feeling lost. Long may our shared hamburgers, biscuits and coffee continue.

In research establishments such as Auckland University of Technology, you quickly grow to treasure that you have found yourself alongside the most faithful companions. All of you seemed to be able to sense when I needed company and when I needed space.

Thank you so much for being so discerning and I sincerely hope I have been able to help you on your own journey.

Barry Dowdeswell, July 27th, 2016.

Abstract

Cyber-Physical Systems are embedded computers that control complex, physical processes via autonomous peripherals while cooperating as agents in distributed networks. Due to the scale and complexity of the interactions that occur within cyber-physical systems, tracing system requirements accurately and appropriately is extremely hard. The literature confirms that they are even harder to maintain and keep up-to-date during the life of the project.

However, the information that requirements traceability provides is a crucial part of determining the completeness of an application. Existing requirements management systems do not scale well and traceability is difficult in such highly heterogeneous environments.

This research presents TORUS (Traceability Of Requirements Using Splices), a novel traceability framework that operates outside of, yet connects to, diverse requirements and development environments. Our approach introduces *Splices*, autonomous traceability data structures that persist trace information through the inevitable changes that occur during system design and development.

A Design Science research methodology was adopted to show how the TORUS framework can be applied to cyber-physical systems that employ the IEC 61499 Function Block Architecture. A mechanical item sorting machine is modeled, the requirements of which are described initially using CESAR (Cost-efficient Methods and

Processes for Safety-relevant Embedded Systems) requirement templates. These templates help to formalize the pre-Requirement Specification's free-form text into less ambiguous requirements statements. A domain ontology is defined before modeling the requirements further within the Sparx Enterprise Architect Requirements Management system. Enterprise Architect uses SysML diagrams to capture each requirement in context with its acceptance tests, non-functional and safety requirements while the model can be persisted for later use.

Formal mathematical models of requirements, function blocks and splices are presented to show how this trace information can be mined, delivering important project metrics to stakeholders. By capturing not only the current state of the system but also by preserving historic traces, TORUS allows project teams to see a much richer view of their system's artifacts.

In parallel with the creation of these models, prototypes of TORUS were created in Java to explore the proposed splice metadata model. These demonstrated that it is possible to extract trace information directly from both Enterprise Architect models and the nxtStudio IEC 61499 object repository.

Using the relationships expressed by these formalisms, the resulting metadata information model for splices is extended to demonstrate how these entities can capture the status of each requirement. We define a set of splices as being the *Skein* of the system; the set of traces that connect the model and application artifacts together like warp and weft of the threads in a tapestry. Information aggregated in this way is important since it provides quantifiable metrics that can be used to provide an empirically-determined overall state of the system under examination.

The results indicate that the TORUS framework scales well and that the skein and splices can provide metrics that should allow us to perform code-level validation and completeness checking in the future.

Publications

TORUS: Tracing Complex Requirements for Large Cyber-Physical Systems.	Dowdeswell, B., Sinha, R. (2016). Submitted to the ICECCS 2016 21st International Conference on Engineering of Complex Computer Systems.
Slicing the Pi: Device-specific IEC 61499 design.	Sinha, R., Dowdeswell, B., Vyatkin, V. (2015). In IEEE 13th International Conference on Industrial Informatics (INDIN) (pp. 1257-1262).
Requirements engineering of industrial automation systems: Adapting the CESAR requirements meta model for safety-critical smart grid software.	Sinha, R., Patil, S., Pang, C., Vyatkin, V., & Dowdeswell, B. (2015). In Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE (pp. 2172-2177).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

How can we be sure that the computer systems that control our elevators, medical devices, automobiles and aeroplanes actually satisfy all the requirements we specify for them? It is often easy to manually trace how requirements are implemented for simple systems such as electric coffee brewing machines. However, the majority of real-world systems are far more complex and must deal with huge numbers of requirements, a problem that has been studied deeply in projects such as CESAR (Cost-efficient Methods and Processes for Safety-relevant Embedded Systems) (Rajan & Wahl, 2013; Mitschke, Andreas and Loughran Neil and Josko Bernhard and Häusler Stefan and Dierks Henning, 2010). The depth of analysis such systems require is only one aspect of their complexity; managing a large requirements set to ensure that each aspect of the design is implemented correctly demands sophisticated requirement traceability techniques (Cleland-Huang, Gotel & Zisman, 2012).

Requirements Traceability is defined as the degree to which a relationship between a piece of code in an application and its stated, parent requirement can be established (IEEE, 1990). Traceability also helps to establish the reason why a particular software algorithm should exist in the product. However, traces alone do not provide ways of ascertaining *completeness*; the degree to which the requirements have been implemented in the system so far (Barnat et al., 2015). Estimating completeness and other metrics can be complex and time-consuming, yet they are crucial in order to successfully deliver a product that meets the expectations of all of its stakeholders (M. J. Ryan, Wheatcraft, Dick & Zinni, 2015).

The focus of this research is requirements traceability for *Cyber-Physical Systems*. These are typically distributed networks of cooperating computers that control intelligent peripheral devices (Khaitan & McCalley, 2014). They control peripherals directly and their computational elements can often perform complex data processing autonomously on the raw telemetry they capture before passing refined information onto other computing elements. In industrial environments, cyber-physical systems contain distributed software which control mechatronic components that interact with complex physical processes. Applications developed for such environments are usually highly standardized.

IEC 61499 is one of several well-known architectures for writing software controllers for industrial cyber-physical systems (IEC, 2013). IEC 61499 is particularly applicable since it provides a formal, event-driven execution model. Each IEC 61499 application is constructed from basic elements called *Function Blocks*, whose inputs and outputs are connected as a collaborative network. Through encapsulation and component re-use, complex and highly modular software for distributed control systems can be constructed.

While traceability is now a well-established and mature software engineering discipline (O. Gotel et al., 2012), cyber-physical systems present unique challenges. Finkelstein comments that traceability in environments where complex and safety-critical requirements predominate is still an emerging discipline; advances in traceability techniques that have emerged within other sectors are not always applicable in this environment (Finkelstein, 2012, p. vi). Distributed processing usually implies decentralized control so tracing an individual requirement through to all the components that work together to fulfill it is difficult (Penzenstadler & Eckhardt, 2012). Large-scale systems can involve thousands of requirements, covering both internal system needs as well as mandated contract requirements (Wnuk, Regnell & Schrewelius, 2009). Storing requirements artifacts in an appropriate framework often requires the use of hierarchical relationships if large numbers of traceability linkages are going to be created. There is little evidence that traditional elicitation and management frameworks will scale when tens of thousands of requirements are involved in cyber-physical environments (Broy & Schmidt, 2014).

This research proposes a novel and generalized framework for the traceability of requirements within large-scale cyber-physical systems. The framework, called *Traceability Of Requirements Using Splices* or *TORUS* enables an automated yet formal approach to tracing requirements. Central to TORUS, is the concept of a *Splice*, which is a link between software requirements, their Use Cases and Unit Tests on one side and system elements such as components, algorithms and their lines of code on the other. By connecting or *splicing together* these artifacts, we can follow the traces like threads in a tapestry. We have named this web of traces the *Skein* of the system. The metadata stored within each splice is used to extract useful traceability-related information such as historical linkages to previous versions, relationships between requirements and system elements as well as information about code coverage. The generalized TORUS framework is presented in Chapter 6.

One of the focuses of the research is scalability. An ideal traceability solution for a complex application that has a large number of requirements has to scale well. It must demonstrate ways of detecting ambiguity and differentiate between correct and incorrect trace information. The literature clearly shows that there is no simple, all-encompassing solution for this class of systems.

1.1 The Workpiece Color Sorter Case Study

While TORUS is not bound to any specific Requirements Model (RM) or system architecture, we demonstrate the applicability of TORUS via a case study using the Cost-efficient Methods and Processes for Safety-Relevant Embedded Systems (CESAR) requirements metamodel (Rajan & Wahl, 2013). IEC 61499 was used for system development. Figure 1.1 presents a *Workpiece Color Sorter* machine which classifies and segregates widgets based on their color. Its architecture is similar to those of real-world cyber-physical systems such as automotive cruise-control systems, pacemakers, elevators, high-speed sorting machines and baggage-handling equipment.

A solid-state camera detects the color of each spherical widget, referred to as a *Workpiece*, to be sorted and a software controller uses this information to actuate one of two pistons to push the sphere in an appropriate direction. We intentionally kept this case study simple and allowed only red and black workpieces. It can easily be scaled to multiple colors where pistons move different distances depending on their color and we can just as easily add more pistons for a two or three dimensional sorting of workpieces. This case study was developed using IEC 61499 in the nxtStudio development environment (nxtControl GmbH, 2016a) and is presented in more detail in Chapter 5. We also use it to illustrate the problem at hand in the following sections of this chapter.

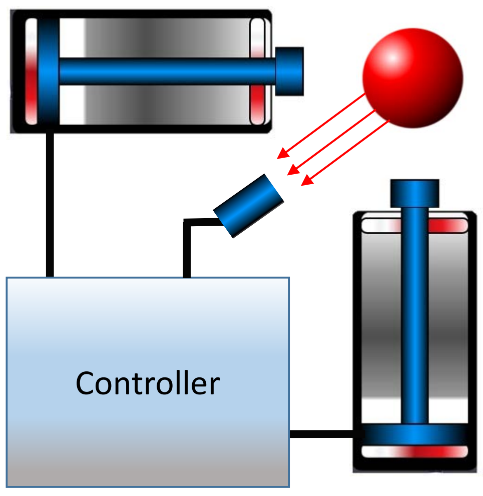


Figure 1.1: The Workpiece Color Sorter.

1.2 Early-Stage Requirements and Specifications

Pre-Requirements Specifications (pre-RS) are the early-stage informal explanations of what a system must do, often elicited directly from project stakeholders (Cleland-Huang, Gotel & Zisman, 2012). Since these documents are typically created very early in the project, they are often free-form, semi-structured, incomplete and ambiguous. A representative pre-RS specification for the workpiece color sorter is shown below:

"The sphere moves into the workspace. The system checks the color of the workpiece. If the sphere is black, the horizontal piston moves it horizontally off of the workspace. If the workpiece is red, the vertical piston moves it vertically. If it is any other color, the direction the thing is moved in is exactly the same direction the last one was moved in."

Current IEC 61499 development tools such as nxtStudio do not directly support traceability so in Chapter 6 we propose how easily TORUS can be adapted for this case study to facilitate traceability. The experimental results showing the applicability of TORUS from a prototype implementation appear in Chapter 7.

1.3 Putting the Research Question into Context

The primary research question this work examines is *"How do we use formal methods to facilitate the traceability of large, complex requirement sets for safety-critical Cyber-Physical Systems?"* Each of the aspects of this question, alluded to in the previous section, need to be put into context in a literature search. Chapter 2 looks at the current state of traceability research and its applicability to cyber-physical systems. We examine the way the automotive, aerospace and industrial automation sectors implement software traceability in real-world environments and the issues they face.

A requirements management framework for a complex, software-intensive system must provide ways of capturing and presenting requirements that do not become cumbersome as the number of requirements increases. Since Object-Orientated Encapsulation is a core feature of cyber-physical system architectures such as IEC 61499, the ability to partition and manage subsets of common requirements is one way of addressing complexity and scalability that is examined.

Change is a constant challenge throughout the design and build phases of a solution's life-cycle. How do we maintain the integrity of the traceability links we create and the code we write? How do we re-factor code safely so that we are not continually breaking the relationships between these entities? Further, how can we refine the requirements in our models in such a way that we can quickly identify which parts of our code need to be re-visited? How do we elucidate non-functional requirements such as reliability and safety from pre-RS and later stage documentation? Hänninen identifies non-functional requirements as being the main source of concern amongst developers (Hänninen, Maki-Turja & Nolin, 2006).

At the same time, cost and time are relentless drivers in real-world projects. While it is always desirable to build thorough, detailed and complete requirements models, the cost of doing so cannot always be justified. Rapid development cycles are often mandatory since the time-to-market for many products is very short. Throughout this research, we examine the trade-offs needed to facilitate traceability that really does provide valuable feedback to the teams who are creating these products.

Chapter 2

Literature Review

The research work to design the TORUS traceability framework has been driven primarily by seeking to understand the needs of representative industry groups. The automotive, aerospace and industrial automation communities are typical sectors where cyber-physical systems are widely used. Hence, there are numerous examples in the academic literature which outline the results of collaborations within these groups and the challenges they face.

This section looks more deeply at the current thinking on each of the key themes outlined in the introduction section. It also explores how we have arrived at our current understanding of traceability based on historical research in this area. Finally, it attempts to clarify the gaps in the research that our work attempts to bridge, which also helps to establish the novelty of this work.

2.1 Cyber-Physical Systems

Khaitan (2014) explains that the computer systems we now refer to as cyber-physical systems have emerged due to a natural and obvious convergence of different computing technologies. For example, the advances in TCP/IP networking topologies from other sectors have been progressively adopted to replace proprietary network protocols that were common in industrial automation environments. These and other complementary technologies have developed independently but where advances in one area have been shown to supersede current methodologies, cross-pollination occurs to take advantage of opportunities. Lee emphasizes that cyber-physical systems are more likely to be designed as networks of interacting elements rather than discrete, stand-alone devices (E. A. Lee, 2008).

Lee and Bagheri (2015) describe the way cyber-physical systems are different from traditional embedded systems using their multi-level 5C model. Figure 2.1 was adapted from their work by including additional comments to illustrate their concepts:

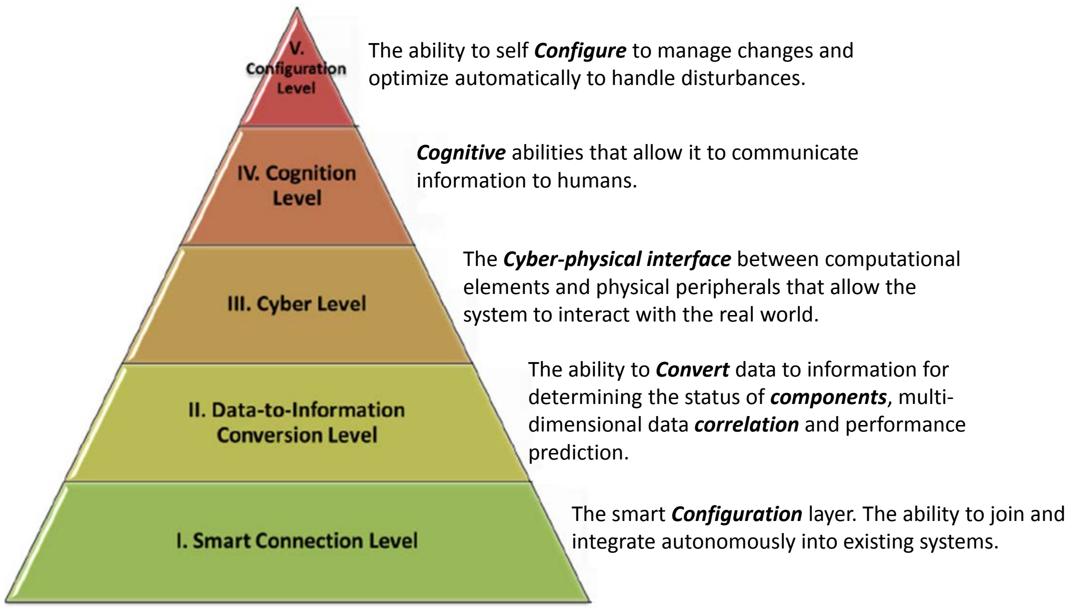

V. Configuration Level
The ability to self *Configure* to manage changes and optimize automatically to handle disturbances.
IV. Cognition Level
Cognitive abilities that allow it to communicate information to humans.
III. Cyber Level
The *Cyber-physical interface* between computational elements and physical peripherals that allow the system to interact with the real world.
II. Data-to-Information Conversion Level
The ability to *Convert* data to information for determining the status of *components*, multi-dimensional data *correlation* and performance prediction.
I. Smart Connection Level
The smart *Configuration* layer. The ability to join and integrate autonomously into existing systems.

Figure 2.1: The 5C Cyber-Physical System Architecture. (adapted from Lee et al,. 2015)

The Smart Connection Level is the lowest layer, closest to the sensors, actuators and motors used to control or interact with the physical parts of the system. Cyber-Physical system elements are responsible for gathering accurate data and disseminating processed information to other, connected parts of the system. To do that, they have to be able to negotiate and manage connections, often to dissimilar units within the same network.

The Data to Information Conversation Level. Cyber-physical systems have powerful computational elements, capable of executing high-level languages, managing large amounts of data and processing algorithms rapidly. They usually execute on high-level architectures (Alippi, 2014; Wolf, 2014; Marwedel, 2010). Cyber-Physical system networks are often used where the capture and processing of large amounts of high-speed telemetry is required (Stojanovic, Stojanovic & Stuehmer, 2013).

The Smart Connection Level is the lowest layer, closest to the sensors, actuators and motors used to control or interact with the physical parts of the system. Cyber-Physical system elements are responsible for gathering accurate data and disseminating processed information to other, connected parts of the system. To do that, they have to be able to negotiate and manage connections, often to dissimilar units within the same network. This creates challenges when verifying and tracing requirements because the system configuration can be dynamic. Since tasks can be shared across distributed components, verifying the fulfillment of a requirement demands traceability strategies that can cope with heterogeneous systems.

The Data to Information Conversation Level. Cyber-physical systems have powerful computational elements, capable of executing high-level languages, managing large amounts of data and processing algorithms rapidly. They usually execute on high-level architectures (Alippi, 2014; Wolf, 2014; Marwedel, 2010). Cyber-Physical system networks are often used where the capture and processing of large amounts of high-speed telemetry is required (Stojanovic et al., 2013).

The Cyber Level lies one layer below the Cognition layer. The Cyber Level is defined as being the point where all data that has been made available to this unit from external sources is correlated. Self-testing and analytics to extract information and make decisions within the Cognition level are processed here.

The Cognition Level acts on the available data, making decisions and managing its presentation to humans as needed.

The architecture outlined shows that cyber-physical systems are much more complex than traditional embedded systems. Hence the need for traceability that traverses these levels is critical.

2.2 Function Blocks as Cyber-Physical Systems

The concept of *programmable* industrial controllers evolved from the purely mechanical cams and relays that had predominated in industrial environments for the last hundred years. The software for the earliest industrial control systems was usually created in low-level assembly language (Osswald, Matz & Lienkamp, 2014). Since on-board memory was expensive, compact, optimized code was the most important characteristic of these systems. As systems became more complex and the microprocessors available to build them with became more cost-effective, software architectures designed specifically for industrial control began to emerge. The IEC 61131 standard, first published in 1992, was designed for industrial controllers, also known as *Programmable Logic Controllers* or *PLCs* (TC65, 1993). This standard consolidated techniques such as Ladder Logic Diagram Programming while offering additional implementation languages that supported the architecture (Kamen, 1999). It adopted the traditional block diagrams commonly used at the time and introduced the concept of these being discrete units of common functionality or *function blocks*.

The IEC 61131 definition of a function block specifies both an interface and an implementation so that within each controller, logical units of functionality can be created and interconnected. While IEC 61131 laid important foundations, it did not support built-in communications layers to allow functionality to be distributed easily across separate controllers (Gerber, Hanisch & Ebbinghaus, 2008). Communications architectures such as PROFInet (Kleines, Detert, Drochner & Suxdorf, 2008) and Modbus-IDA (Modbus, 2004) that were implemented to exchange information between distributed nodes evolved from earlier proprietary work by vendors. The IEC 61499 Standard was released in 2005 (IEC, 2013) and addressed these communications needs as well as deeper architectural issues that were intrinsic to the then current industrial control paradigms.

2.2.1 Event-Driven Object-Orientation in IEC 61499

IEC 61131 functions blocks always operate in a cyclic, polled-mode. In contrast, IEC 61499 is entirely *event-driven*; the function block only responds when new data appears at its *event inputs*. The interface it presents contains event inputs and outputs in addition to data inputs and outputs. Each event represents a finite state in a state machine that is described by its *Execution Control Chart* or ECC. This architectural model associates pre-defined discrete states with code algorithms to implement constraint logic and data processing. Hence IEC 61499 function blocks remain idle until new data is received through their event inputs. This abstraction is a key factor in enabling IEC 61499 function blocks to be described formally since they are deterministic finite state machines (Yoong, Roop, Vyatkin & Salcic, 2009).

Vyatkin (2009) explains that this was a novel concept to PLC engineers when it was introduced. They were more familiar with the timed cyclic approaches PLCs of the time used and were comfortable with their ability to share variables globally across the whole application. The strong, object-orientated encapsulation of IEC 61499 encouraged component re-use since global, and hence hidden, cross-module variable dependencies were not possible. IEC 61499 only allows data to be exchanged between function blocks using message passing (Graefe & Basson, 2014). Hence, implementing distributed systems using function blocks is no more complex for the designer than creating local applications; all communication requirements are provisioned automatically at design and compile time by the development systems themselves.

2.2.2 Real-world system implementation using IEC 61499

IEC 61499 addresses well each of the cyber-physical system architectural layers shown in Figure 2.1. Table 2.1 illustrates the diversity of the usage of IEC 61499 in typical industry sectors where cyber-physical system approaches are having an impact:

Table 2.1: Application of IEC 61499 by industry sector

Industry Sector	*Focus of IEC 61499 implementation*
Industrial and manufacturing systems.	- Agent-orientated techniques for manufacturing systems (Hoffman et al., 2014). - Software agents for real-time control of manufacturing systems (Hegny et al., 2008). - Integration with Enterprise Resource Planning (ERP) systems for flexible manufacturing (Morel et al., 2003).
Intelligent buildings.	- Energy-efficiency and the use of ontological agents (Mousavi et al., 2014). - Smart agents in cyber-physical building control (Leitao et al., 2016). - Safety-critical issues in smart buildings (Saldivar et al., 2015).
Smart Power Grids.	- Multi-agent techniques for intelligent automation and regulation (Zhabelova & Vyatkin, 2012). - Distributed intelligent control of power systems (Strasser et al., 2011). - Decentralised architectures for voltage regulation (Loia & Vaccaro, 2011).
Smart Power Grids.	- Multi-agent techniques for intelligent automation and regulation (Zhabelova & Vyatkin, 2012). - Distributed intelligent control of power systems (Strasser et al., 2011). - Decentralized architectures for voltage regulation (Loia & Vaccaro, 2011).
Traffic Control Systems.	- Intelligent, component-based approaches (Black & Vyatkin, 2010).
Baggage Handling Systems.	- Cyber-Physical design approaches (Yan & Vyatkin, 2011).

Yan and Vyatkin (2011) describe the design of baggage handling systems built using IEC 61499 in terms of cyber-physical concepts. Both Young and Roop (2012) and Riedl and Zipper (2014) explain how IEC 61499 addresses the challenges of scale, dynamic adaptability and complexity needed to build large cyber-physical systems.

Vyatkin (2011) cites the development of IEC-61499-specific integrated development environments such as nxtControl (nxtControl GmbH, 2016a), IsaGRAF (ISaGRAF, 2010) and 4DIAC (Strasser et al., 2008) as facilitators of the acceptance of IEC 61499 by industry. Gerber (2008) observes that IEC 61131 had become a classical and well-accepted programming methodology for PLCs, effectively slowing the uptake of newer standards. The advantages of IEC 61499, including encapsulation of intellectual property, are being taken advantage of only when systems are re-designed as well as when larger systems are contemplated. However, given the size of the user base for IEC 61131, changes like this take time.

2.3 Software Requirements for Cyber-Physical Systems

We have proposed traceability as a way of managing *software requirements*, which are statements that describe a discrete, atomic capability that a computer system must be able to perform to satisfy a client's needs. Hamilton and Zeldin explain that requirements are "*those items that are desired*" while the resultant specifications are *"..the results that realize those requirement"* (Hamilton & Zeldin, 1980, page 29).

IEEE Standard 610.12-1990 (IEEE, 1990, page 62) defines software requirements more formally as:

1. *"A condition or capability needed by a user to solve a problem or achieve an objective.*

2. *A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.*
3. *A documented representation of a condition or capability as in 1 or 2."*

The earliest references to software requirements engineering in a distributed, embedded system's environment is Alford and Lawson's work at TRW, a U.S. Department of Defense provider in the early 1970's (Alford & Lawson, 1979). They echo Hamilton and Zeldin's reflections on the difficulties encountered in the design and verification of software used on the Apollo program during the 1960's (Hamilton & Zeldin, 1976). Both emphasize that the task of managing requirements is a time-consuming, error-prone and on-going process, one that should not stop when the initial specification is thought to be complete. This was in direct contrast to the widely-used forward-only approaches of the 1950's that would later come to be known as the Waterfall Method (Royce, 1970).

Alford and Lawson's work was a year-long study of the problems encountered in the development of complex weapon systems and embedded real-time software. By the late 1970's, the control systems for rockets designed by TRW could contain over a million requirements (page 114). As the missiles they were creating became faster and were required to have more precise targeting, their data processing and control needs became significantly more difficult to fulfill. Their study sought to uncover ways to solve problems through an integrated requirements engineering approach. They proposed the use of automated tools as well as formal mathematical foundations for front-end requirements engineering and design. Key to their recommendations was the need for requirements traceability. While this might initially seem to be an obvious conclusion, implementing the methodologies, processes and frameworks to capture software requirements properly and facilitate traceability was and still is fraught with controversy and open problems. Alford and Lawson's preface to their 1979 report bears repeating in the current context of cyber-physical systems:

"..however, the major cause of inadequate software, poor requirements definition and design, had been relatively neglected by the R&D community. The few efforts which did address the more tangible pre-coding and pre-design activities, yielded prototype developments for specific and limited applications... A broad and comprehensive view of the initial user-developer interactions was needed; one which portrayed goals and alternative solutions, in spite of complexity, being successively defined and refined within a framework of effective common understanding." (Alford & Lawson, 1979, page xiii).

Further than this, Finkelstein (2012) re-iterates that the requirements elicitation process must be a *relationship-driven* one. On-going involvement with stakeholders is more important than Royce's uni-directional Waterfall approach, which often isolated clients and end-users from the engineering process. The resulting catastrophic mismatches between what was requested and what was delivered are well documented; the survey by Emam and Koru (2008) reported a combined rate of failure and cancellation of IT projects as high as 34% while the Standish Report of 1994 continues to be debated to this day (Standish, 1994).

2.3.1 Functional and Non-Functional Requirements

It is useful to categorize requirements into sub-groups dependent on their differing characteristics. *Non-functional requirements* often capture the qualitative aspects of a system including scalability, capacity, performance and reliability (A. J. Ryan, 2000). *Functional requirements* are quantitative, more concrete descriptions of needs that describe more implicitly discrete tasks that a system has to be able to perform. In the specification for the workpiece color sorter, requirements which are functional describe how the workpiece should be moved under well-defined, concrete conditions. The intended motion of the red and black workpieces is clearly specified so when testing such a system, functional requirements such as these usually pass or fail in more clearly defined ways than a non-functional requirement would.

Hänninen et al. (2006) identify non-functional requirements as being the main source of concern amongst developers. A non-functional requirement such as performance, perhaps measured in transactions processed per second, can fail by almost reaching its target. Ryan (2000) comments that most studies that focus on inadequate requirements only discuss functional requirements. In practice, non-functional requirements are most often fulfilled by successfully implementing a related set of functional requirements. Ryan notes that it is significant that the software engineering standards EIA 632 (SAE, 2016b) and ISO 15228 (ISO, 2015) do not address non-functional requirements at all.

Chung (2012) notes that there is a need to describe non-functional requirements such as performance or throughput in quantitative terms such as baggage items or fruit sizes sorted per second. Since non-function requirements are often thought of in qualitative terms, there is the temptation to think of them as somehow immeasurable. Non-function aspects of cyber-physical systems are also observed in the Cognitive Level of Figure 2.1 where aspects including ease-of-use and understandability are requirements for the human-machine cyber interface.

The ability to trace non-functional requirements is important since safety-critical requirements are most often non-functional in nature (Cleland-Huang, Gotel & Zisman, 2012). Forward-traces assist with project planning and scale estimations (Ramesh, Powers, Stubbs & Edwards, 1995). Backward-traces to original design decisions help to reinforce acceptance testing of load capabilities and transactional throughput by providing justification for baseline metrics. Siewiorek and Narasimhan (2005) illustrate this by examining the fault-tolerant characteristics of satellite avionics systems for whom the unattended, autonomous mode is the most common operational environment. Tracing both functional and non-functional requirements from a fault-tolerant perspective leads to more resilient cyber-physical systems which usually have to take rapid corrective action autonomously without operator intervention.

The concept of requirements existing also as run-time entities to support multiple System-of-Systems instances and monitor their performance is now emerging (Vierhauser, Rabiser, Grunbacher & Aumayr, 2015; Bencomo, Whittle, Sawyer, Finkelstein & Letier, 2010). Boeing pioneered the ability for the requirements set of an airliner to be used to drive real-time predictive diagnostics or *prognostics* (Stephenson, 2006). It is used by flight and maintenance engineers to monitor both functional and non-functional performance of components and predict imminent or future failures.

2.3.2 Pre-Requirement and Post-Requirement Specifications

One of the core problems of requirements engineering is that pre-RS stakeholder specifications are notoriously informal, vague, ambiguous, and often unattainable (Li et al., 2015, page 164). They must be refined through the application of systematic processes to transform them into requirement statements that are consistent, formal and measurable.

Gotel and Finkelstein (1994) define the *Post-Requirements Specification* (post-RS) as those aspects of a requirement's life after it has been included in the requirements specification. Figure 2.2 illustrates the demarcation point that separates the scope of traceability initiatives in the pre-RS phase from the subsequent post-RS phase. Pre-RS traceability is most often useful during root-cause analysis of later requirements issues and for determining who the original owner of that particular need was. Post-RS traceability reaches from the requirements specification itself through to the code in the application and can be bi-directional (O. C. Gotel & Finkelstein, 1994). This has implications for the resultant scope of traceability initiatives; defining where the traces will stop and start can make a significant impact on the cost of implementing traceability (O. Gotel & Mäder, 2012). Knowing *what* to trace as well as *how far* to trace is a crucial decision in any project.

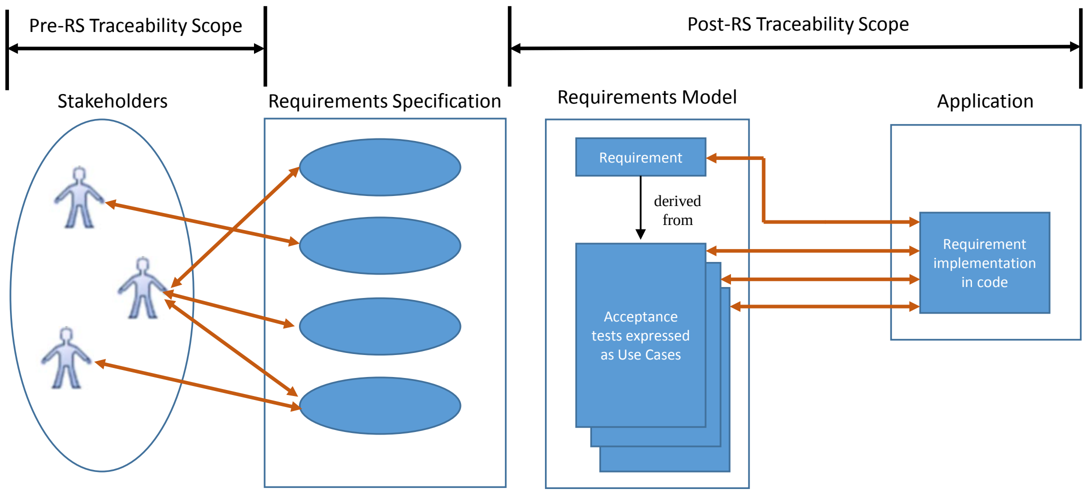

Pre-RS Traceability Scope
Post-RS Traceability Scope
Stakeholders
Requirements Specification
Requirements Model
Application
Requirement
derived from
Acceptance tests expressed as Use Cases
Requirement implementation in code

Figure 2.2: The scope of pre-Requirements and post-Requirements Traceability (after Gotel & Finkelstein, 1994)

Not all traces are useful and defining the criteria to determine what is important is challenging. Vierhauser (2015) discusses what to monitor in real-time, linking it the concept of the requirements scope. Scope is defined in terms of what System-of-System instances need to be monitored and what they connect or *reach* to. When cyber-physical components protect their intellectual property from inspection, it is not possible to trace beyond the public interface of a resource. This impacts the granularity of the traceability; in IEC 61499 systems, it is often desirable to be able to reach down to the level of individual algorithms in safety-critical environments to ensure that all specified scenarios are covered.

2.4 Traceability Initiatives for Cyber-Physical Systems.

The problems Alford and Lawson identified are still being explored in the current literature. Finkelstein (2012) provides a perspective on the current maturity of traceability in this sector in his introduction to Cleland-Huang, Gotel and Zisman (2012). He states that general traceability has reached a level of maturity where the problem has been characterized and tools exist to manage traces between different document types. However, while he believes that traceability works well in most environments including Agile ones, he states that *"..there remains a hard core of large systems development characterized by strong safety and other constraints. Defense and other mission-critical systems exemplify this. There is a continuing need to address traceability in this setting and in particular to support navigation of the complex relationships that arise. This still remains at the edge of what can be practically accomplished and will require further research."* (page vi).

In parallel with Alford and Lawson's early work on cyber-physical systems, the NATO working conference in 1968 included one of the earliest references to traceability. It recommends making traceability an implicit part of the design process (Naur & Randell, 1969). Boehm writing on software quality (1976) also explored the concept of *external consistency*, an attribute of the code that implies that its content was traceable to the original requirements. He also cited the need for code to be self-descriptive in the sense that it contains enough information for the person reading it to be able to verify its *"objectives, assumptions, constraints, inputs, outputs, components and revision status"* (page 604). He suggests that one way of realizing this was to create easily traceable paths to previous changes and code comments. By doing so, he foreshadows the later use of metadata from requirements models to support the traceability of algorithms back to their requirements (Fernández, Penzenstadler, Kuhrmann & Broy, 2010).

The following sections examine the techniques used to facilitate traceability and the challenges that are still being reported in the literature.

2.4.1 Facilitating Traceability via Requirements Models

A *Software Requirements Model* is a collection of alternate visualizations or views of an application's text-based requirements using diagrams and formal expressions (Van Lamsweerde, 2009). Through modeling, individual atomic requirements can be inter-related to expose different aspects of them to designers and users. Patterns and issues emerge when we see information presented in different ways that would not be discernible if requirements were expressed purely as lists of needs (Lindland, Sindre & Solvberg, 1994). Dubois & Perialsi-Franti (2010) argue that requirements models are the foundation for validation and verification strategies.

Dorfman and Flynn (1984) discuss the management of thousands of requirements within large aerospace systems. They cite Brook's familiar *Software Tar Pit* metaphor to explain the need for better requirements management if high-quality software products are to be delivered (Brooks, 1975). Analysts and developers become progressively more lost in a sticky tar pit of ever-changing specifications and code if requirements are not managed in a systematic, structured way. Inadequate requirements are cited as the most significant contributers to project failures (Dorfman & Flynn, 1984; O. C. Gotel & Finkelstein, 1994; Wiegers & Beatty, 2013; Bell & Thayer, 1976).

Traceability relies on having structures that trace-creation systems can navigate through and nodes within these structures that they can attach to. Hence requirements models are a framework for organizing requirement statements. Requirements are inherently hierarchical and software elements attach to nodes within this hierarchy at differing depths. Tracing strategies have to be able to navigate through nested levels but there is no single, established technique for doing so. Conversely, there is no single, pervasive requirements model that fits all cases (O. C. Gotel & Finkelstein, 1994, page 1). Table 2.2 details requirements modeling systems and architectural description languages currently in use.

Table 2.2: Traceability within Cyber-Physical System Modeling Solutions.

Name	*Focus and Features*	*Traceability-related Issues*
Sparx Enterprise Architect (Architect, 2010).	- Application - Uses SysML. - UML model generation. - Integration with multiple IDEs	- Limited to links between internal models.
IBM Rational Rhapsody (IBM, 2016).	- Application. - Uses SysML - C/C++/C# code. - Multiple industry-specific versions.	- Limited to links between internal models. - Does not easily trace externally-generated code.
AUTOSAR (AUTOSAR, 2016).	- Automotive Framework - Open industry standard - UML metamodel - Focused on standardizing interfaces - ISO 26262 compliant	- Limited to safety-critical systems - Requires compatible industry-specific specifications.
AMALTHEA (Wolff et al., 2015)	- Automotive Framework - Open source Toolchain.	- Limited support for traceability. - Focused on timing.
SPICE (Automotive, 2010).	- Framework. - Automotive, Aerospace and Avionics. - Focused on software process improvement. - Assessment via the Capability Maturity Model.	- No dedicated support or assessment process.

2.4.2 Traceability Tools for Cyber-Physical Systems.

The previous sections have examined what a requirements model is as well as defining the characteristics of the target cyber-physical system that we wish to trace; an application that implements those requirements by distributing its functionality across multiple, connected sub-systems. Such architectures present unique traceability challenges that are explored in more detail in section 2.5. The current literature presents numerous tool-based approaches to implementing traceability strategies. All of these tools can be classified into one of three broad categories:

1. ***Workbenches.*** These are integrated environments for either design or development that often include traceability functionality. Examples of requirements modeling tools that have this capability are Sparx Enterprise Architect and IBM Rational Rhapsody. Examples of development system add-ons include Eclipse ReqCycle (Eclipse, 2016b), Eclipse ProR Requirements Engineering Platform (Eclipse, 2016a) and the Visual Studio Team Foundation Server (Microsoft, 2016). None of these integrated workbenches can reach out and mine data from an external development system. Some of their websites discuss external import and mechanisms for linking-in information but these require the data to first be re-formatted externally before being imported (Architect, 2010; IBM, 2016).
2. ***Specialised Traceability Systems*** are tools that sit outside of the development or requirements modeling environment. Shalid and Suhaimi (2011) discuss systems such as Retro (Hayes et al., 2007), Borland CalibreRM (Borland, 2016) and Cradle (Corporation, 2016). None of the systems they cite support automatic link detection and creation but some are able to detect when a link is broken or has changed. However, studies by both Hansen (2005) and Braun (2014) suggest that there is strong evidence that both workbenches and specialized systems are underutilized and not well-integrated.
3. ***General-purpose tools*** include unenhanced word processing or spreadsheet systems, paper-based systems with or without customized forms and collaboration tools such as Google Docs or Dropbox. These often evolve over time to address particular characteristics of the company that implements them (O. C. Gotel & Finkelstein, 1994). However, they are often ad-hoc solutions with limited integration or ability to mine traceability linkages automatically. Gartner (2014) reported that forty to fifty percent of companies manage the creation, iteration, testing and launch of new products with only Microsoft Office, Google Docs and other all-purpose documentation tools. Gotel and Finkelstein comment that

these systems often cannot guarantee the long-term viability or re-usability of the models they generate.

2.5 Design Challenges for Cyber-Physical System

The previous sections have discussed the foundation principles of requirements modeling and traceability as they apply to cyber-physical systems. Figure 2.3 emerged from time spent considering the ways in which these entities relate to each other in the context of how they are derived, what they validate against and where function blocks sit within this hierarchy. It was adapted from Ortel and Malot's (2013) original diagram:

This diagram again emphasizes the point that has emerged continually within the literature; to successfully trace a requirement through to its implementation and back demands a robust, formal structure to host the requirements and all their related entities within. Any successful traceability initiative that has to track a large heterogeneous solution has to relate each entity within the model in a navigable way.

However, cyber-physical systems push the limits of the current abstractions that we use to describe systems (E. A. Lee, 2008; Rajkumar, Lee, Sha & Stankovic, 2010). This section examines some of those challenges in the context of how they impact our ability to trace requirements.

2.5.1 The Role of Abstraction in Cyber-Physical System Design

Lee (2008) comments that industrial control systems are always expected to meet a higher standard of reliability, safety and fault-tolerance than business computing systems. With the proliferation of autonomous functionality within cyber-physical systems, those expectations will only increase. However, unlike desktop business systems, cyber-physical systems inhabit a world that is highly unpredictable. While the reliability of computer hardware has increased exponentially, what our systems

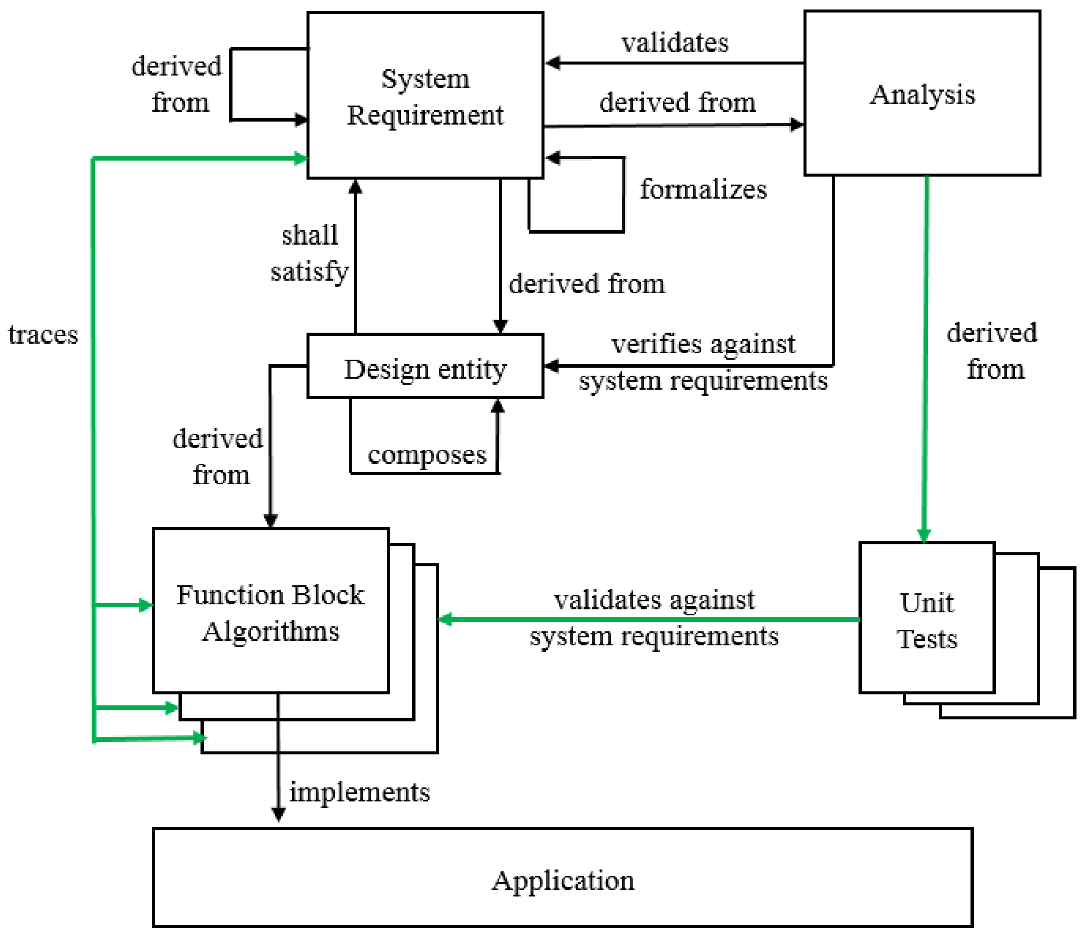


Figure 2.3: Relationships Between the Entities Within a Cyber-Physical System Model. (adapted from Ortel et. al., 2013)

are expected to control in the wild has become extremely complex. Lee asks if it is even technically feasible to make such control systems predictable and reliable (page 364). Our current programming language abstractions, especially those embodied in the C language, cannot guarantee timing since they do not manage concurrency in ways that are applicable to heterogeneous systems. While the *semantics* of a particular programming language may be impeccable, they cannot guarantee it will meet timing deadlines; current language abstractions do not capture such characteristics.

Rajkumar et al., (2010) propose that we need abstractions with a much broader scope to define cyber-physical systems. They must address not only the interactions between computing elements and time-critical external processes but also how the system interacts with humans. In particular, how we compose architectures for heterogeneous, hierarchical distributed systems is not well defined at present. Security approaches for cyber-physical, which rely heavily on agent-orientated techniques, are not the same as those required for homogeneous business systems (J. Lee et al., 2015). This remains an open question which is discussed further in Section 2.6.

2.5.2 Cross-functional Outsourced Development Challenges

Large cyber-physical systems are usually Systems-of-Systems and are highly heterogeneous. Particularly in the aerospace and automotive industries, the life-time of a design can be extensive as it is adapted to support multiple product lines (Broy, Kruger, Pretschner & Salzmann, 2007; Gaeta & Czarnecki, 2015). Within these environments, what does a large-scale cyber-physical systems development project look like? Figure 2.4 illustrates the cross-functional activity silos that would be needed to manage the entities detailed previously in Figure 2.3. What are the challenges that face managers and developers when trying to trace information across these zones of activity?

Outsourced development of specialized sub-systems often requires that they be maintained and put into operation by independent teams from multiple companies. The assumption that the system under design is under the control of a homogeneous group of stakeholders, who determines a consistent set of requirements, no longer holds (Vierhauser et al., 2015, page 97). They can be refined by diverse teams over a period of years, making long-term management of requirements problematic. This observation is supported by Charette (2009) who explains that automotive manufacturers are now more like systems integrators than Original Equipment Manufacturers (OEM's). Broy (2007) suggests that while functionality may change as little as 10% from vehicle-to-vehicle with each new generation, co-ordination and version control between suppliers and

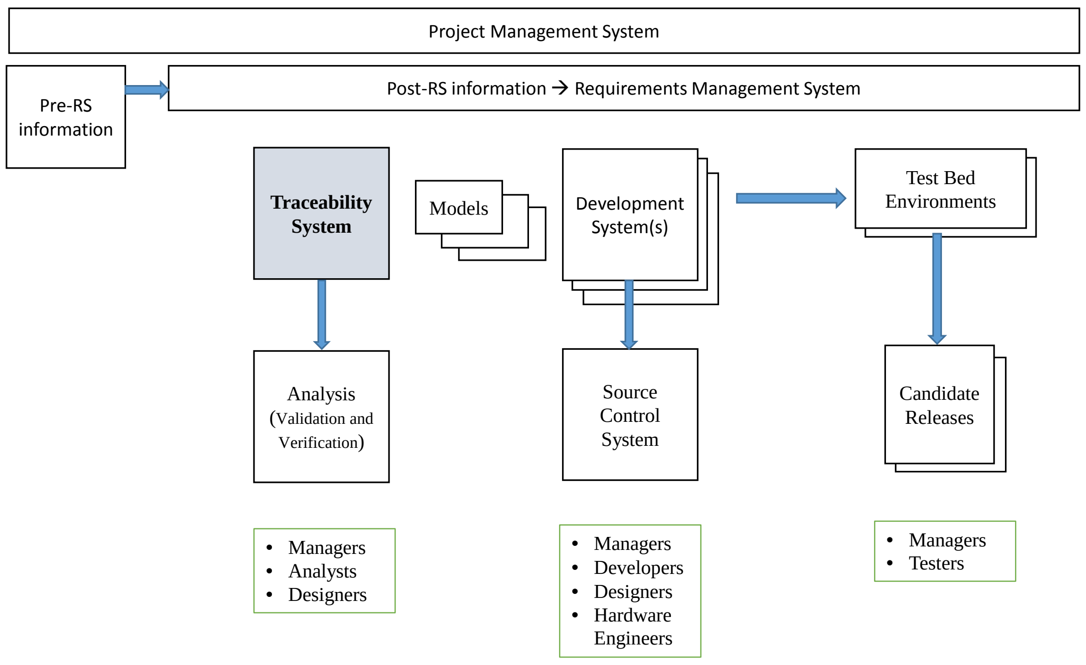


Figure 2.4: Cross-Functional Activity Silos for Cyber-Physical Systems Development.

OEM's is difficult.

2.5.3 Safety-Critical Aspects of Cyber-Physical Systems

Safety is a qualitative property of a system that is often a key justification for implementing traceability. If a requirement of the system is to be recognized as being *safety-critical* during the early elicitation and analysis phases, preexisting metrics for identifying such behaviors must be available. Safety-Critical Requirements are non-functional requirements, fulfilled by other functional requirements (Lutz, 1993). For example, a safety-critical requirement to stop the pistons of the workpiece color sorter colliding is fulfilled by the other functional requirements that stop the pistons moving at the wrong time. In this section, we examine what safety-critical requirements are and what standards exist to define, mandate and prove compliance.

Achieving compliance is primarily a way of establishing confidence in the design of a system. In that sense confidence is a derived property founded upon the premise that a standard provides a metric to measure how completely a solution meets specified criteria. Traceability provides a measure of confidence however, mandatory compliance alone is not a way of identifying all possible dangerous characteristics or failure modes. Ho gives an obvious example of the dangers of computer tape handling robots (Ho, 2015). It is significant that no international safety standards for the use of industrial robots exist (USDOL, 2015). The 2015 accident at Volkswagen that resulted in a worker being killed during the commissioning of an industrial robot was not necessarily a failure in design that would have been identified via a compliance audit (Docrill, 2015). Hänninen proposes that the software in such systems should be able to be overridden by the cyber-physical mechanisms in cases where the software malfunctions and a safety-critical requirement is violated (Hänninen et al., 2006).

Safety is closely coupled with the concept of risk. Charette (1991) defines risk as that which has a possibility of loss associated with it where chance or uncertainty is involved. The first risk-averse real-time computing system is believed to be the Whirlwind Project, created by MIT in 1944. It was used to evaluate alternatives for building military air traffic control systems, the first of which was deployed in the early 1950's (J. Bowen & Stavridou, 1993). Since the mean time between failure in Whirlwind's vacuum tubes was twenty minutes, fault-tolerance was established by identifying weak tubes and re-directing signals to redundant backup components to maintain operations. This echoes Charette's definition; establishing empirical probabilities of failure in such systems is an important part of designing appropriate risk mitigation strategies. You cannot manage what you cannot measure (Breen et al., 2012) and traceability is one mechanism to help evaluate compliance. However, a poor integration between requirements engineering and safety engineering has been identified as an open challenge (Martins & Gorschek, 2016).

Hence safety-critical standards and guidelines exist to define acceptable behavior and compliance activities to manage risk appropriately. All of the following examples mandate traceability as a key component of compliance:

DO-178C, Software Considerations in Airborne Systems and Equipment Certification is a standard for the design of safety-critical avionic software, including embedded and cyber-physical devices (FAA, 2011). It mandates traceability from system requirements to all source code. Traceability tools used as part of the process must form part of the compliance declaration and certification process.

ARP-4754, Certification Considerations for Highly Integrated or Complex Aircraft Systems is an Aerospace Recommended Practice (ARP) document that details the compliance levels and practices for DO-178C standard compliance when applied to large-scale or complex avionics subsystems (SAE, 2016a). It includes traceability practice guidelines.

ISO 26262 Standard for Road vehicles - Functional safety is the 2011 standard that defines compliance for electronic and electrical systems in production automobiles (ISO, 2016). It embodies the Automotive Safety Integrity Level (ASIL) risk classification scheme which is a common guideline used across a set of similar automotive standards. It is complementary to other standards including D0-178C and the railway standard CENELEC 50126/128/129. Part six of the standard is concerned with MBSE software development practices. It outlines fault-injection techniques using a V-Model approach (Rana et al., 2013). Born and Favaro criticize the heavily document-orientated compliance requirements of ISO 26262, pointing out that manual compliance documents are difficult to integrate into automated traceability frameworks (Born, Favaro & Kath, 2010). Often the trace linkages to the primary compliance documents have to be created and managed manually.

ANSI/AAMI/IEC 62304:2006 defines appropriate software life-cycle procedures for medical device software (ANSI/AAMI, 2006). It was harmonized with other European standards in 2008 and is based on the ISO 12207 generic software guidelines (ISO/IEC, 1997). Section 5.2 of IEC 62304 states that the manufacturer must establish a software development process that addresses *"traceability between system requirements, software requirements, software system test, and risk control measures implemented in the software"*. Section 7.3 details risk-control measures that include the ability to *"..document traceability of software hazards as appropriate: From the hazardous situation to the software item. From the software item to the specific software cause and from the software cause to the risk control measure and from the risk control measure to the verification of the risk control measure"*.

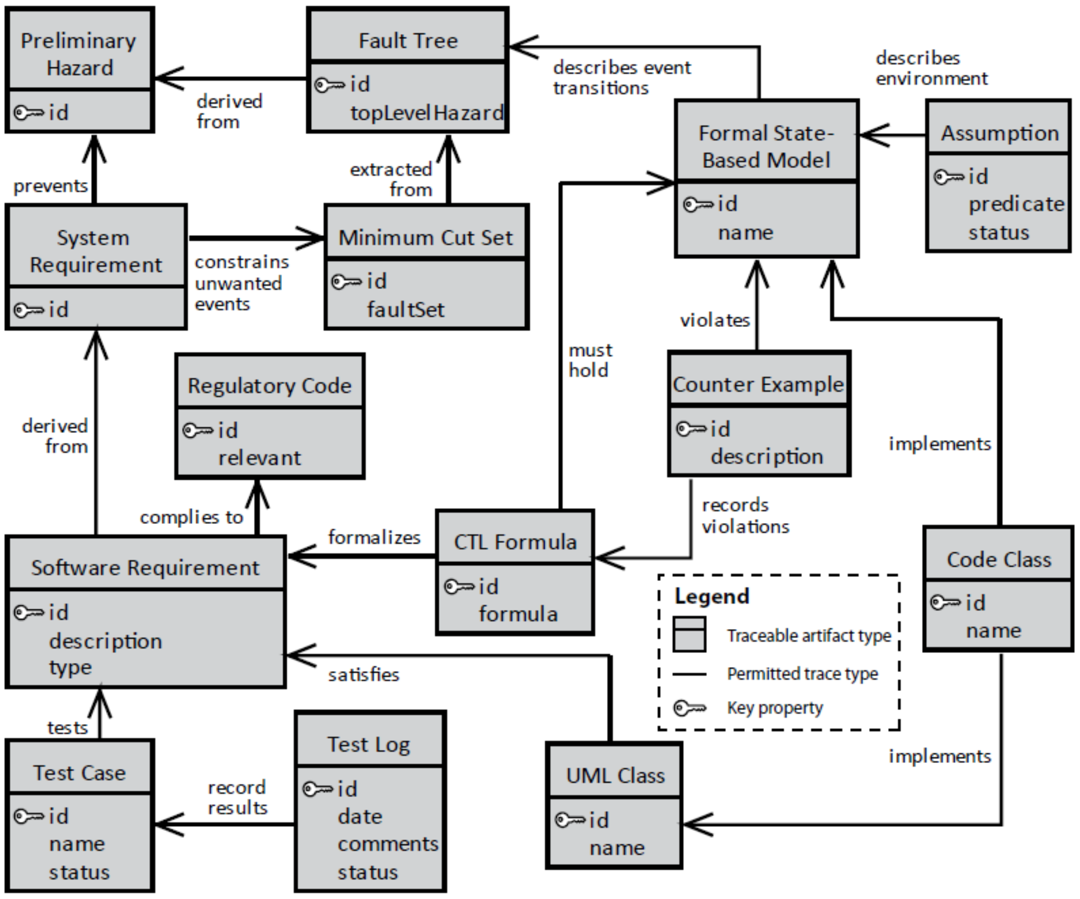


Figure 2.5: A Traceability Information Model for a Safety Critical System (Cleland-Huang, Heimdahl et al., 2012).

However, there is evidence from both government and industry case studies of an almost universal failure to implement traceability successfully for safety-critical systems. Those studies suggest that this is primarily due to the *"difficulty of constructing* *useful traceability queries using the existing tools"* (Cleland-Huang, Heimdahl et al., 2012, page 180). Figure 2.5 presents examples they give of the complexity of typical *Traceability Information Models* (TIMs) that are needed to capture safety-critical requirements. They argue that since regulatory agencies require full-cycle traceability in both directions, models need to be comprehensive enough to capture all the relevant safety requirements. Models that are too generic do not yield well to formal analysis.

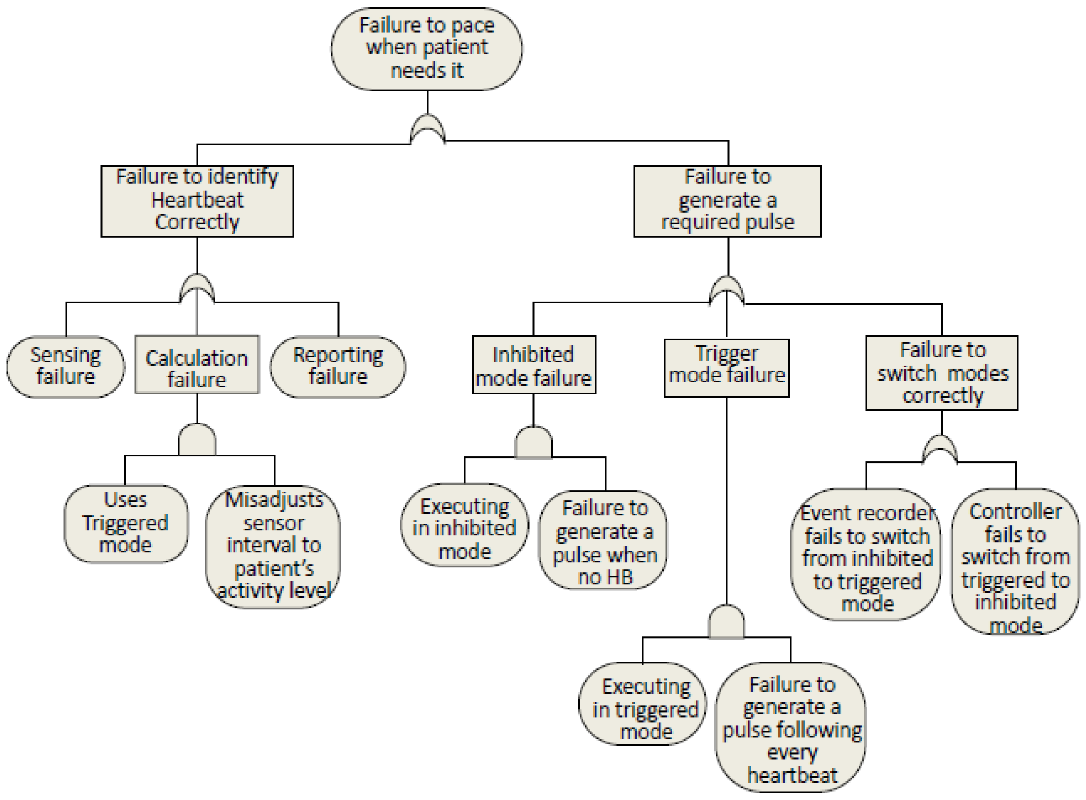

Failure to pace when patient needs it
Failure to identify Heartbeat Correctly
Failure to generate a required pulse
Sensing failure
Calculation failure
Reporting failure
Inhibited mode failure
Trigger mode failure
Failure to switch modes correctly
Uses Triggered mode
Misadjusts sensor interval to patient's activity level
Executing in inhibited mode
Failure to generate a pulse when no HB
Event recorder fails to switch from inhibited to triggered mode
Controller fails to switch from triggered to inhibited mode
Executing in triggered mode
Failure to generate a pulse following every heartbeat

Figure 2.6: The Fault Tree for a Surgical Pacemaker (Dehlinger & Lutz, 2004).

Building complementary *Fault-Trees* allows formal methods to identify counter-examples that show that a particular state can be entered in a way that violates a safety criteria. Fault trees are often used in these scenarios to facilitate Primary Hazard Analysis (PHA). These model constraints are mandated by the safety standards in a way that allows them to be used in formal methods. They do this by allowing the high-level fault description to be modeled as a series of connected intermediary states with their complementary transitions. Dehlinger and Lutz illustrate this with a fault tree for a surgical pacemaker as shown in Figure 2.6 (Dehlinger & Lutz, 2004).

2.5.4 The Complexity of Large Cyber-Physical Systems

Cyber-physical systems have become complex because the scope of what we are able to do with current technology makes feature-rich systems possible. Economies of scale, reliability and lower costs mean that it is now more cost-effective to innovate via software than it is via mechanical features. Why is that and what are the problems it causes?

The automotive sector is a typical example of how manufacturers have become increasingly dependent on cyber-physical control systems. Volkswagen introduced the first computer-controlled fuel injection into their 1600 Type 2 E models in 1968 (Osswald et al., 2014). Note that their discrete component system pre-dates the release of the first integrated circuit based microprocessor, the Intel 4004, in 1971 (Bohr, 2009). By 2007, typical high-end vehicles such as the BMW series 7 relied on 67 different embedded micro-controllers (Pretschner, Broy, Kruger & Stauner, 2007). The 2011 Chevrolet Volt hybrid relies upon ten million lines of code, four million more than is required by the F-35 jet fighter (Pesce, 2011; Charette, 2009). By 2016, vehicles are predicted to contain up to 100 separate Embedded Control Units (ECUs) requiring 100 million lines of source code (Charette, 2009).

Automotive control systems interact with their users and each other in a myriad of ways, requiring the exchange of data between engine management, electronic stability controls, security and multimedia systems. McKinsey estimates that the ECU's in 2014 vehicles can exchange over 25GB of data per hour (McKinsey, 2014). This explosion of capabilities and the resultant complexity is driven in part by the economic value that these systems return. By 2015, the value of the software generated by these systems is estimated to rise to 133 billion EUR (Braun et al., 2014). Extrapolating beyond this to 2018 suggests that the total value will rise to 179 billion EUR by 2018 (Arpinen, Hämäläinen & Hännikäinen, 2011) for total sales of 1 trillion EUR. Total expenditure on research in the automotive sector is estimated to rise to 100 billion EUR by 2020 (Hill, Menk & Swiecki, 2016).

Figure 2.7 was adapted from Braun (2014), Broy (2007) and additional data from Hill and Menk (2016). It shows that in 2000, the percentage value of software and computing hardware contributing value to the entire automobile was 22%. By 2020, it is expected that software will constitute 80% of the value of the vehicle. The percentages in the boxes show the proportion of the software value to the hardware value that make up each solution. By 2020, the proportional value of the hardware will have dropped to 20%. While hardware becomes increasingly more cost-effective, the expenditure on software development will continue to rise.

Braun notes that this is an increase in market value of the software of nearly 530% since 2003. Hardung states that electronics in automobiles now account for 90% of the innovations. Of that, 80% are attributable to software. (Hardung, Kölzow & Krüger, 2004). Broy et al. note that, compared to the cost of the mechanical parts of the car, the cost of replicating software is nearly zero (Broy et al., 2007). Coupled with the low cost of the processing hardware, the most cost-effective way to realize innovation such as this is through software. The increasing reliability and sophistication of the hardware has meant that realizing ever-more complex, software-intensive systems is not infeasible; it is just difficult.

However, metrics such as the lines of code quoted previously can be misleading; the Boeing 787 Dreamliner requires only 6.5 million lines of code while the latest F-35 Joint Strike Fighter uses 5.7 million lines of code (Charette, 2009). Surely a system which can take off and land by itself, managing multiple engines is more complex than an automobile?

It is not that simple. McCabe explains that code complexity is independent of its physical size. He explains that this *Cyclomatic Complexity* depends only on the intricacy of its branching structure (McCabe, 1976). McCabe also discusses good and bad coding practices. The IEC 61499 Function Block architecture creates systems that consist of large numbers of discrete algorithms, encapsulated as C++ or Java functions. These *Event-Processing* code blocks are responsible for decision making while interacting with the peripherals that the system is controlling. Having a clear understanding of what makes good, well-structured and reliable code that is applicable to this architecture is an important contributor to the problem of creating unambiguous traceability linkages.

In cyber-physical systems, complexity is also evidenced when the number of different functional building blocks used increases while the number of non-trivial relationships and interactions between them is growing larger (Regnell, Svensson & Wnuk,

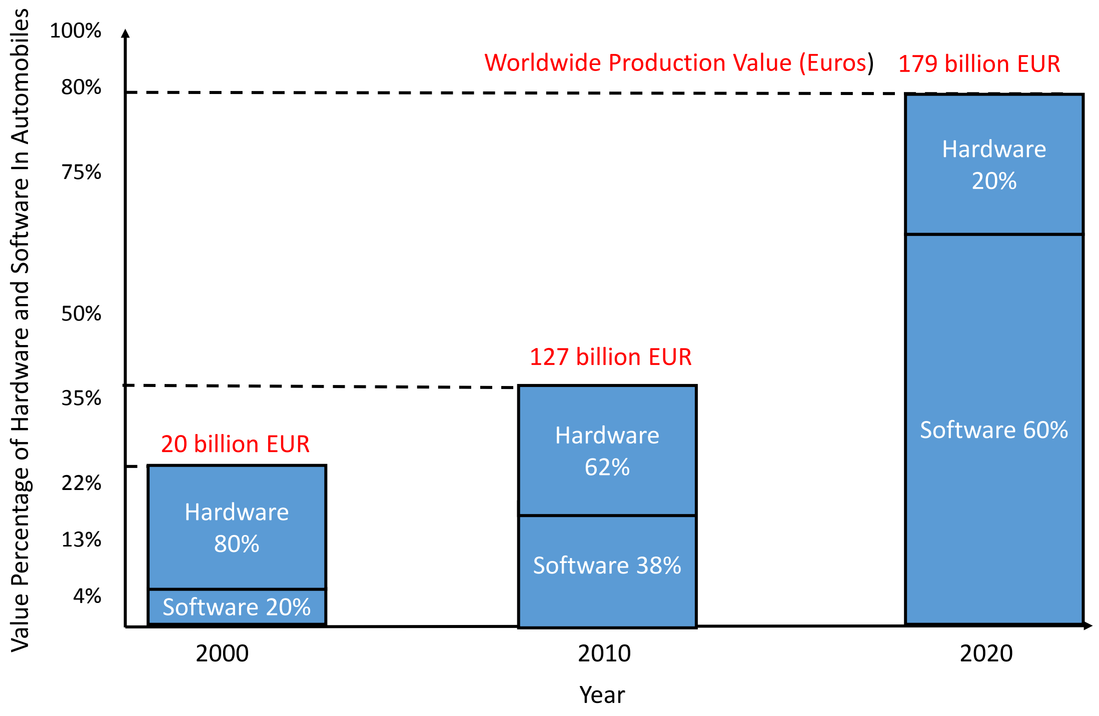


Figure 2.7: Rise of importance of Software in Automobiles (adapted from Braun et al. 2014).

2008). Documenting, tracing and verifying the requirements of such software-intensive systems becomes exponentially more time-consuming and error-prone as the overall system code base expands.

Complexity manifests itself through two distinct characteristics of cyber-physical systems:

A. ***They are often Systems-of-Systems.*** To maximize the return on investment for their development, the automotive and aerospace industries increasingly rely on the re-use of standard sub-assemblies across a range of different vehicles. These are typically crafted in-conjunction with both internal developers and external suppliers (Hardung et al., 2004). While car manufacturers would prefer just to be Original Equipment Manufacturers (OEM's), they operate today far more like systems integrators. The design of reliable and re-usable internal software interfaces becomes critical to amortize costs, increase reliability and standardize inventories. Hence, requirements frameworks have to be able to cope with black-box interfaces and hierarchies of requirements. Where safety-critical certification is required, deep requirements traceability and verification become mandatory. However, they operate in a world of black-box componentry, the internal workings of which may be closed and proprietary.

B. ***They rely on System-to-System interactions.*** Engine management systems work in tandem with not only data from the engine hardware itself and its myriad of sensors but also with the data they exchange with other autonomous sub-systems. Electronic Stability Controls, Hybrid Fuel Management and driver interfaces rely on critical data interchanges to facilitate both efficient fuel management and share co-operative tasks. As the number of systems that rely on each other increases, specifying and validating requirements that span multiple systems becomes problematic.

Pretschner cites deeper drivers of complexity. Since automotive and aerospace systems often rely on sub-systems sourced from multiple vendors, they are highly *heterogeneous* (Pretschner et al., 2007). In contrast, *homogeneous* systems are more common in companies that use closed-architecture's such as Apple. Automobile manufacturers seek to have common systems that handle variants across their range. Such variations, referred *product lines*, are part of their drive for differentiation within mass markets. Pretschner (2007) gives the example of a Crysler power-train control application that can have 3,488 possible component variations by using different code algorithm configurations. Coupled with this, base software systems are designed to have a life-span of at least 15 years. They must evolve as each year's model changes whilst providing backwards-compatibility for vehicles which are already in the field. In the aerospace environment, systems can exhibit lifetimes of 30 years or more. Suppliers have to guarantee the supply of compatible componentry for that period due to FAA regulations. In contrast, the automotive industry has to cope with the changes that occur when electronic components become obsolete within three to five years (Condra, 2015).

There are also deep interactions between separate sub-systems. Krüger and Nelson give an example of door control systems used in Daimler cars (Krüger, Nelson & Prasad, 2004). Up to eighteen separate electronic control units can communicate with the door locking systems. The Engine Management System ensures that it informs the Door Control Systems that the vehicle is moving and that the doors should be closed. However, in the event of a collision, the systems that co-ordinate the air bags also interact with the Door Control Systems to ensure that they are unlocked at the right time.

2.5.5 Using Formal Methods in both design and traceability

Formal methods are those techniques that are by definition, structured or *formalized* rather than being arbitrary or ad-hoc (Monin, 2012). While many formal methods involve mathematical approaches, in their simplest expression they have three primary characteristics:

1. They are alternative ways of specifying how we build our requirements models.
2. They are processes, often involving mathematical techniques, that we run against our requirements model to explore it.
3. Formal methods help to provide rigor and structure to validation, verification and traceability techniques.

Holloway asserts that, just as in any other engineering discipline, performing mathematical analysis in software engineering contributes to the robustness and reliability of a systems' design (Holloway, 1997). Key to Alford and Lawson's recommendations were the use of formal mathematical methods to support requirements engineering processes (Alford & Lawson, 1979, page xiv). Hamilton and Zeldin's earlier work on design and verification of spacecraft guidance systems emphasizes the same need by relying on formal mathematics for verification. Hamilton's later work inspired the concepts of the *Correct By Construction* methodology that utilized requirements frameworks that can be analyzed by mathematical techniques (Hamilton & Zeldin, 1976). In a related work of theirs, the formal mathematical verifications they describe rely on locating code to analyse via traceability methods (Hamilton & Zeldin, 1980).

Within this context, Meyer (1995) stated that it is clear that a more rigorous mathematical approach was needed if software quality was to improve. This section explores what the current role of formal methods is within cyber-physical system development and how traceability supports these approaches.

Formal approaches verify and validate models using specific properties and quality characteristics (Guessi, Oliveira, Garcés & Oquendo, 2015). Software architecture models are necessarily abstract but they are constructed from concrete statements or representations of the atomic requirements. Architectural Description Languages (ASLs) are used to compose such models. While UML and SysML are semi-formal languages, formal languages including MARTE, which focuses on performance and scheduling, EAST-ADL2 and AUTOSAR are widely used in automotive software design (Guessi, Cavalcante & Oliveira, 2015).

Bowen and Hinchley emphasize the following aspects of formal methods (J. P. Bowen & Hinchey, 1995):

The use of appropriate notations. Architectural Description Languages are formal in the sense that they use technically-precise notations, which often include mathematically-based syntaxes. Pre-defined and agreed language constructs help to promote consistency.

Formal methods should be used appropriately. There is a limit to value returned in creating formal methods for every part of a system. However, specifying requirements using formal notations makes them more concise.

Formal Methods do not replace testing however, formal models can be used to create executable specifications (Marwedel, 2010).

There are a number of studies that explore the impact of formal methods on the effectiveness of traceability. Easterbrook and Singer (2008) explain how formalizing requirements in models for NASA embedded software systems allowed traces to be established to verify Functional Concept Diagrams (FCDs), which are flowchart-like diagrams used to model algorithms. They report that finding little traceability in some parts of their system was also useful since it identified issues that were not well handled in their models. Bozzano et al., (2014) discuss model-based checking of satellite systems using formal methods that rely on system-generated traces and well-defined properties. Their focus is on early stage design validation where formal methods are used to address complexity issues within avionic systems. They stress that manual methods of traceability are too cumbersome while the advantages of probabilistic model checking allow them to perform more dependable risk assessment. Spanoudakis and Zisman detail rule-based creation of traces using formal constructs to analyze free-form text requirements using Natural Language Processing (NLP) to assist with the analysis of their model's integrity (Spanoudakis, Zisman, Pérez-Minana & Krause, 2004). Their work is interesting since they model formal state relationships that capture overlapping, incomplete and partially-fulfilled requirements. This mirrors the work of Vierhauser and Rabiser that detected false trace monitoring violations that cross into incorrect areas. This is a problem of context; the traceability strategy cannot determine if it is looking in the correct part of the system (Vierhauser et al., 2015).

Formal methods also help to facilitate safety-critical compliance. Breaux and Gordon (2013) discuss ways of data mining and interacting with regulatory requirements, including examples from Legal Requirements Specification Languages (LRSL). These are formalisms which requirements management systems ought to be able to extract information from automatically to ensure that legal or compliance requirements are being considered and satisfied.

Hence within the scope of our research question, formal methods are by definition not just mathematical operations. Rather, they are all those techniques that facilitate the validation and verification of the system requirements by performing traces within a well-defined requirements model.

2.6 Gaps, Conclusions and Opportunities

Traceability is a key enabler of validation and verification strategies, especially when applications demand safety-critical compliance. Good models and reliable traces yield quantifiable benefits, as suggested by Mäder et al., in their study of the design of software maintenance tasks that were supported by traceability. On average, subjects performed 24% faster on given tasks and produced 50% more correct solutions (Mäder, Gotel & Philippow, 2009b).

However, without adequate models and the adoption of appropriate traceability tools, the task of manual traceability is onerous. The high cost of creating traces and maintaining them is only perceived to be beneficial if it leads to measurable product improvements, even with the use of automated traceability strategies (Ramesh et al., 1995). This problem is exacerbated by standalone tools that do not easily integrate with external development systems, requiring traces to be created and updated manually. It is significant that Microsoft Visual Studio, Eclipse and nxtStudio do not include traceability tools and resources as standard core modules. In reality, the tools that do support these environments are independent enhancement products created by other vendors.

These shortcomings are also reflected by the predominance of general-purpose tools in practice (O. C. Gotel & Finkelstein, 1994). Gartner's statistics quoted earlier support this assertion; less than 50% of the participants in their survey were using specialized traceability or requirements management tools.

Given this trend, there is a clear need for tools that take a different approach. We could find little evidence of traceability tools which were able to reach both into requirements management systems as well as being able to mine data from development tools. A neutral traceability architecture such as this would present opportunities to capture historical information as well as perform formal analysis on the entities it has created. Subsequent chapters examine modeling strategies and techniques in more detail. Chapter 4 examines how we can refine requirement statements and express them more concretely. Techniques examined include the applicability of the CESAR requirement boilerplates (Mitschke, Andreas and Loughran Neil and Josko Bernhard and Häusler Stefan and Dierks Henning, 2010) as well as the Object Management Group (OMG) SysML Requirements Diagrams (OMG, 2016a).

Scope is also a challenge. Mistakes when deciding on the complexity, depth and intensity of requirements models can be costly; Mäder et al. (2009b) and Leffingwell (1997) both explain that the wrong granularity of traces leads to overly complex or inadequate traceability graphs. During the design of TORUS, the scope of the traceability between the requirements model and the example nxtStudio application needs to be defined clearly and justified. As stated earlier, knowing *what* to trace as well as *how far* to trace is a crucial decision in a project.

Complexity and scalability are a core focus of the research. An ideal traceability solution for a complex application that has a large number of requirements has to scale well. It must demonstrate ways of detecting ambiguity and differentiate between correct and incorrect trace information. The literature clearly shows that there is no simple, all-encompassing solution that addresses cyber-physical systems.

The most appropriate abstractions to use for modeling cyber-physical systems also remains an open question. Cyber-physical systems push the limits of the current abstractions we use to describe applications (E. A. Lee, 2008; Rajkumar et al., 2010). Later in Section 8.1, during the discussion following the creation of the workpiece color sorter in nxtStudio, the value of the SysML model created in Enterprise Architect is brought into question. Is the nxtStudio application itself a more appropriate requirements modeling environment? IEC 61499 is arguably an excellent abstraction for representing cyber-physical systems. If we are to encourage practitioners to adopt Model-Based Systems Engineering approaches, we have to be able to justify the cost and effort that will go into creating models by facilitating excellent traceability, validation and verification.

Chapter 3

Research Methodology

This research seeks to identify techniques that deliver reliable requirements traceability for large cyber-physical systems. TORUS splices are discrete structures that have been proposed as a way of identifying and correlating the code that fulfills a requirement, contained within an appropriate requirements model. Rather than being qualitative, we have sought to build experimental frameworks and verify them with evidence from example systems. Formal methods were used to better understand and attempt to verify our models of requirements, function blocks and splices.

Sjøberg (2007) asserts that no matter what the form, the essence of empirical research is to acquire knowledge by empirical methods. This section examines research methodologies for running and evaluating controlled experiments on our example requirement sets and the function blocks that satisfy them. It contrasts the two most promising-looking approaches while considering some aspects of the methodological pluralism and diversity that mixing these methods might achieve.

3.1 Factors in choosing a research methodology

Selecting an appropriate research methodology is hard. Easterbrooke (2008) suggests that the pros and cons of any particular method are not always well-documented in the literature. Wohlin and Aurum explain that this is partially because the underlying assumptions of the methods used are often not well understood by those using them (Wohlin & Aurum, 2015). Since researchers often have little knowledge of alternatives outside their field, there is little cross-pollination of ideas and techniques.

Both Easterbrooke and Sjøberg et al. (2007) provide broad guidelines for what constitutes high-quality empirical research. Core to their thinking is the need to increase the shared understanding of how to conduct empirical research because:

- Practitioners need to understand how the individual components of a research methodology work together. For example, how do literature reviews influence the design of later experimental work?
- It is important to report and understand the implications of assumptions that have been made.
- Research results need to be presented appropriately and confidently. Tables, figures and graphs are different approaches to presenting results that need careful consideration to determine what works best in any particular situation. *Infographics* is an emerging, cognitive discipline that seeks to understand how graphical representations of information can present difficult concepts quickly and unambiguously (Schoffelen et al., 2015; Siricharoen, 2013; Borkin et al., 2013).
- Original research needs to also be able to contrast and build on other researchers work in context. Newton's compliment to Robert Hooke in February, 1676 came at the end of a long period of both conflict and collaboration between them:

"What Descartes did was a good step. You have added much in several ways, and especially in taking the colours of thin plates into philosophical consideration. If I have seen further it is by standing on the shoulders of giants" (Maury, 1992; Brewster, 1860).

- There is also a case for using more than one methodology. Purely empirical research that generates quantitative results can be enhanced in some situations by qualitative input from surveys and interviews later (Wohlin & Aurum, 2015).

3.2 Alternative Research Methodologies

One way of verifying the TORUS architecture and designing the metadata requirements for the splices would be by the creation of representative cyber-physical system models in a software tool. A prototype of TORUS would allow the different splice scenarios and types to be examined in realistic scenarios. For example, the resultant splice metadata that describes the *state* of the requirement that is not yet instantiated in code is expected to be different from the scenarios of orphan code. Hence the scope of the methodology exploration was narrowed to exclude those approaches or methodologies that did not directly support experimentation via construction.

Easterbrooke et al. (2008) identify six research approaches that were examined while considering how to design and evaluate the TORUS framework:

Action Research is characterized by the combination of theory and practice being explored in a real-world practitioner scenario. Aviston & Lau (1999) state that it is a quantitative method that is useful in explaining the behavior of software in a particular organizational setting. In experimental settings, it can often help to format and present evidence in ways that augment expert opinions (Dos Santos & Travassos, 2011). Runeson and Höst contrast it with other methodologies stating that it is primarily qualitative but offers flexible research design options (Runeson & Höst, 2009).

Ethnographies are similiar to Action Research in that they co-locate researchers in the field with their subjects and examines all available data (Hutchins, 1995). Many conclusions arise from observations and outlier data that might not have been captured or seen as relevant in other methodologies. Ethnographic and Action Research would be more applicable to our situation if the research was evaluating existing traceability regimes that were already being used in the field.

Case Studies and Surveys were considered as a way of later determining what would encourage practitioners to adopt TORUS as a traceability strategy. However, surveys can be problematic if there are insufficient tangible results to propose. Runeson and Höst caution that over-simplistic *"toy studies"* do not yield results that contribute meaningful findings (Runeson & Höst, 2009, page 132). Client-focused case studies and surveys will be considered for future work once TORUS is ready for field trials.

Controlled Experiments and Quasi-Experiments have more applicability in social science or medical randomized trials. A quasi-experiment is an empirical study used to estimate the causal impact of an intervention on it's target. It is not always possible to demonstrate a causal link between a condition and observed outcomes. This is particularly true if there are confounding variables that cannot be controlled or accounted for.

Constructive Methods and Design Science are research approaches that embrace aspects of behavioral science, the search for theories that predict or explain organizational or human behavior (Kukafka, Johnson, Linfante & Allegrante, 2003). However, behavioral science in this field has seldom focused on evaluating iterative models; its focus is primarily on the use or acceptance of software artifacts" (A. Hevner & Chatterjee, 2010). In contrast, Constructive Methods and Design Science are methodologies that *"seek to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts"* (A. R. Hevner, March, Park & Ram, 2004, page 75).

Piaget argued that humans generate knowledge and meaning from the interaction between their experiences and their ideas (Wadsworth, 1996). Within Information Science, Piaget's constructivist and exploratory approaches often include the creation of software and hardware systems that attempt to model one or more aspects of a research problem.

Nunamaker et al. (1990) presented some of the earliest work done in the area of design science. They proposed a multi-methodological approach to building information systems where experiments, case studies, simulation and mathematical methods were used iteratively:

1. In the *Theory-Building Stage*, new ideas and concepts are proposed and possible frameworks are evaluated. Formal methods are encouraged since they allow constraints to be modeled and assumptions to be tested more rigorously. Hevner and March (2004) explain that formal methods allow quantitative rather than just qualitative data to be gathered, allowing activities such as *"optimization, analytical simulations and quantitative comparisons to be performed"* (page 77).

2. *Experimentation* can be performed not only under laboratory conditions but also through limited and controlled user-participation trials.

3. *Case Studies, Surveys* and *Field trials* allow for a more extensive evaluation of prototype artifacts, sometimes for extended periods. Nunamaker et al. describe this as an *observational* phase whose analysis and outputs feed back into repeated cycles as the design is refined iteratively.

Hevner and Chatterjee (2010) propose that Behavioral Science and Design Science are actually complementary and effectively inseparable. They essentially draw from a pragmatic philosophy which proposes that *truth* is theory that has been justified while *utility* is evidenced when artifacts are deemed to be effective (Figure 3.1). The

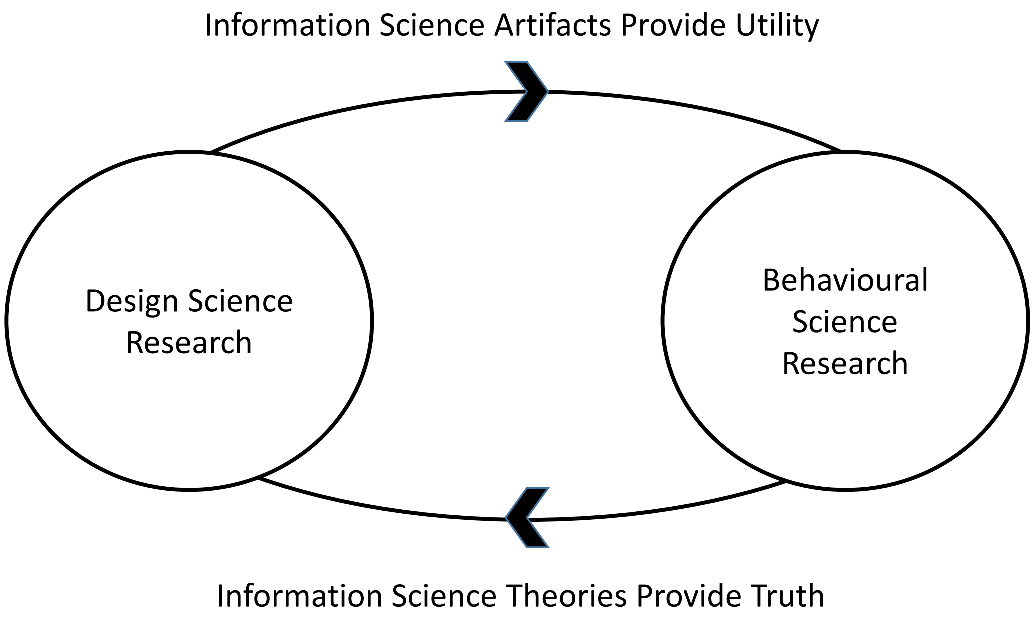

Information Science Artifacts Provide Utility
Design Science Research
Behavioural Science Research
Information Science Theories Provide Truth

Figure 3.1: The Complementary Nature of Design Science and Behavioral Science Research (A. Hevner & Chatterjee, 2010, page 11).

implication is that information science research should be evaluated in the light of its practical outcomes. Easterbrook and Singer et al. take this further; *"pragmatism adopts an engineering approach to research - it values practical knowledge over abstract knowledge, and uses whatever methods are appropriate to obtain it"* (Easterbrook et al., 2008, page 292).

3.3 A Framework for Design Science Research

Hevner and March (2004) present a framework for conducting design science research shown in Figure 3.1. It proposes seven key aspects to focus processes and protocols around.

Each of these recommendations were used to draw up guidelines and propose tasks and activities for modeling the TORUS architecture. Prototypes were developed iteratively in phases where creating part of a TORUS implementation would be relevant. These guidelines support an iterative approach that was proposed and discussed in the planning stages of the research that is shown in Figure 3.2:

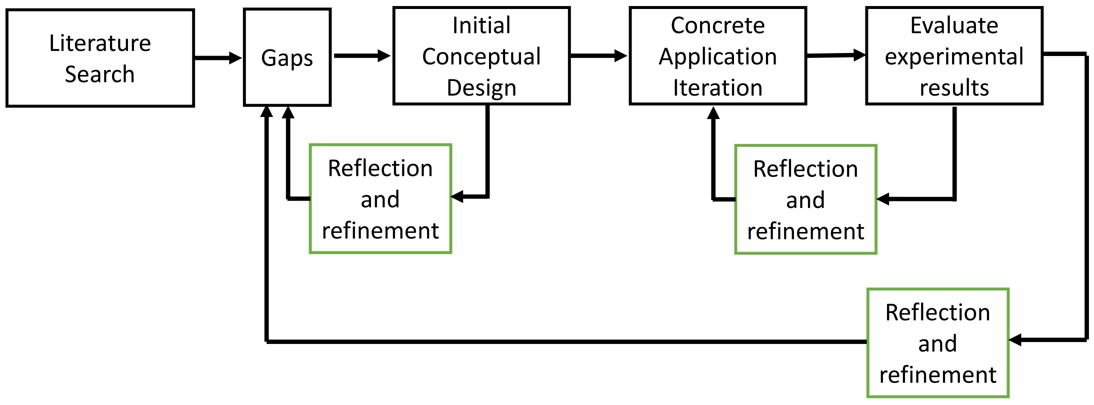

Literature Search
Gaps
Initial Conceptual Design
Concrete Application Iteration
Evaluate experimental results
Reflection and refinement
Reflection and refinement
Reflection and refinement

Figure 3.2: The TORUS Design Science Research Protocol.

Table 3.1: Design Science Research Guidelines

Guideline	Description
Guideline 1: Design as an Artifact.	Design Science research must produce a viable artifact in the form of a construct, a model, a method or an instantiation.
Guideline 2: Problem relevance.	The objective of Design Science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design evaluation.	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research contributions.	Effective Design Science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research rigor.	Design Science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a search process.	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of research.	Design Science research must be presented effectively to both technology-oriented and management-oriented audiences.

3.3.1 Guideline 1: Design as an Artifact

Hevner and and March's first guideline states that design science research must produce a viable artifact. Deeper than that, design is concerned not only with the form of the IT artifact but also with the creative processes, modeling techniques and formal mathematics that underlie each instantiation. Research and development management strategies are also a core component that evolves as new techniques and methodologies are introduced as needed. Hence, design is manifested not just in the quality of the resulting system artifact but also in the quality of the justified theory and processes that produced it (Norman, 2013).

Figure 3.2 shows the research cycle steps, where the thinking and gaps uncovered in the literature search result in an initial conceptual design. Our first iteration of this loop is detailed in Chapter 4 where the requirements for the Workpiece Color Sorter are refined and modeled. The resultant requirement's set provides examples of the data that need to be captured in our traceability structures as well as helping to identity potential splice types and relationships.

Hollan defines this aspect of *culture* in the context of software development by viewing culture as the process that *"accumulates partial solutions to frequently encountered problems"* (Hollan, Hutchins & Kirsh, 2000, page 178). Without this residue of previous activity, we would all have to find solutions from scratch, re-inventing them each time we encounter the same problem. In this context, Hollan and Hutchens both see culture as a process rather than as a collection of things (Hutchins, 1995).

3.3.2 Guideline 2: Problem relevance

Hevner proposes that "*A justified theory that is not useful for the environment contributes as little to the IS literature as an artifact that solves a non-existent problem*" (A. R. Hevner et al., 2004, page 81). The literature search uncovered evidence that traceability in CPS's is not a mature discipline yet. TORUS is also designed to investigate aspects of complexity and scale that were identified as being important. Applegate and King caution that there is a balance between the need for rigor discussed later in Guideline 5 and the desire to only include experiments that are deemed relevant (Applegate & King, 1999). They argue that there is a need for both.

It is to be hoped that the TORUS framework and the concept of splices will be seen by practitioners as a viable traceability approach. However, if TORUS is not supported by a strong underlying design framework, it will be difficult for other researchers to extend it with their own contributions. Appropriate design documentation was developed in addition to the material that was created specifically for this thesis. The modeling and design of the classes and data structures in the prototypes reflects iterative coding practices that are typical in Test-Driven Design environments (Diepenbeck, Kühne, Soeken & Drechsler, 2014).

3.3.3 Guideline 3: Design evaluation

The quality and applicability of a design artifact must be demonstrated by well-defined and rigorous methods. Our requirements set was used to create a formal Requirements Model in Sparx Enterprise Architect (Architect, 2010). This process is documented in Chapter 4 and includes a formal definition of what a requirement is. This led to the creation of a prototype of the Workpiece Color Sorter in the nxtStudio IEC 61499 development environment (nxtControl GmbH, 2016a). This process was documented in Chapter 5 and presents a formal definition of a function block. During this phase, possible ways of tagging algorithms were investigated to determine if they could be used to automatically identify traceability endpoints. Deliberate gaps exist in our requirements model that were seeded there to see if evidence of their presence emerged as particular classes of splices.

Design Science result evaluations rely on the identification of metrics that demonstrate the efficacy of the artifacts produced in an iteration (A. R. Hevner et al., 2004). Our approach was to model both ends of the solution and determine how TORUS could bridge the gap between our requirements model and our application solution.

Both Norman (2013) and Hevner (2004) assert that efficacy of design is not enough; good designers incorporate both style and aesthetic's in their artifacts. Gelernter describes this as *"machine beauty"*, the marriage between simplicity and efficacy that drives innovation in science as much as it does in technology (Gelernter, 1999, page 3). The quality of the underlying architecture of TORUS will contribute to supporting the planned future research into cognition and visualization of large requirement sets. Part of that will involve determining how easy it is for practitioners to make sense of the results that traceability solutions such as TORUS present.

3.3.4 Guideline 4: Research contributions

This guideline was considered to be an extension of guideline two. Weber considers the lifetimes of IT artifacts and their resultant cycles. Valuable research contributions include software artifacts that inspire and motivate research by other practitioners as well as wide-spread acceptance of novel solutions that extend the field (Weber et al., 1997). Our contribution lies in both the instantiations of TORUS as well as the introduction of the concepts of the Splice and the Skein of the application.

3.3.5 Guideline 5: Research rigor

Rigor in design science research is maintained by the careful design of both artifacts and evaluation strategies. However, there is a fine balance to maintain since some problems require serendipitous solutions. Buchanan and Brooks both argue that while some technological advances are the result of creative and perhaps innovative design science processes, some solutions are arrived at via capricious or sometimes arbitrary routes (Buchanan, 1992; Brooks, 1987). Those are the *"ah ha!"* moments that make this sort of research so rewarding.

3.3.6 Guideline 6: Design as a search process

Design is an iterative search to find an effective solution for a problem. Simon (1996) describes this as a cyclic process of generating design alternatives followed by testing each iteration against known requirements and constraints. However, when the range of possible solutions or routes that could be followed is large, we cross into the space described by Buchanan as *"wicked problems"* (Buchanan, 1992, page 15). These are problems whose descriptions are extremely difficult to formulate and where the data that accompanies them is confusing or indeterminate. While design science processes through iterative phases, it is hard to predict the number of iterations that would be needed to produce a solution. In such circumstances, the solution may emerge unexpectedly.

Hevner and March (2004) suggest that one way of keeping such scenarios under control is to continually attempt to determine how close the artifact, developed in an iteration, is to being an optimal solution. This task is supported by the application of Guideline 3 that addresses the methods for evaluating designs.

Following the first iterations that created the requirements model, a TORUS prototype was created. The formal definition of splices and skeins is presented in Chapter 6 in the context of the TORUS architecture. This prototype was first used to explore the requirements model of the workpiece color sorter. A later iteration extracted the splices from the IEC 61499 function blocks used to create the workpiece color sorter application and linked them to the requirement model splices found in the first iteration.

3.3.7 Guideline 7: Communication of research

TORUS was never intended to be an application that only existed within the confines of a research environment. Such solutions mature when they are continually presented with real-world scenarios that describe contemporary problems. When research is published, there is a juxtaposition of fresh ideas from other researchers that can be used to approach the problem in new ways. Hence, in future work, collaboration with other researchers and examining TORUS in the wild remain as key aims.

3.4 Conclusions, Limitations and Assessing the Threats to Validity

This section has explored alternative research methodologies to determine what would best support the design and evaluation of TORUS. An experimental design-science approach that first built the requirements model, then built a matching IEC 61499 application was chosen as the best way of getting to the point where evaluating splice scenarios was possible.

While some aspects of the expected metadata structure of a splice had already been worked out during the early research planning phase, a number of unanswered questions remained about what the scope of possible splice types might be. It was determined that some basic splice types were trivial to conceptualize; when there are no algorithms in the code, then all splice statuses would show that the requirement existed but was unfulfilled. As representative scenarios are explored more deeply, it is expected that new splice classes that are more fine-grained and descriptive will emerge from the first basic cases.

When evaluating the research methodology chosen, three threats to the validity of our results and the limitations of this study were considered more deeply:

1. **The Limitations of Single-Outcome Studies.** The objective of most case studies is to discover something about the broader population of cases. While single-outcome studies are often viable in medical research (Gerring, 2006), they can pose problems during Design Science studies. There is a risk that the act of design and construction can become an end in itself. Hence exemplars and cases need to be convincing and representative, but not contrived. One approach is to ensure that while the artifact can have a single, well-defined deliverable goal in each phase, the aim of each iteration should be to realize different, evolving aspects of the overarching need. The Workpiece Color Sorter is a simple example yet is nonetheless valuable since it also embodies safety and performance requirements which have only emerged during the subsequent requirement's elucidation. These were able to be identified by representative splice types and states. As such, though we are examining only one application in this study, the splice concepts it has led us to are arguably novel constructs.
2. **Understanding the scope of prototypes.** The prototypes produced from this research are not fully-featured implementations. That is an expected condition of typical design science projects; artifacts are categorized as innovations that help to define ideas, theory and practices (Denning, 1997). The true value of the TORUS framework would only be realized once it has been exercised in a number of real-world scenarios. This maturing time for a prototype builds application resilience by stressing it within a production environment and ensures that it addresses real problems, not just academic ones. However, before such field trials can begin, the prototype must reach a level of maturity in a laboratory environment that takes it beyond the "toy" stage.
3. **IEC 61499 alternatives within the context of the Internet of Things.** IEC 61499 is not the only architecture currently being used to build modern cyber-physical systems. Hence, the first iterations of TORUS have to be developed while

remaining cognizant of the need to allow alternative requirements management and development systems to be accommodated later. Function Blocks were chosen since they lend themselves well to analysis by formal methods (Dubinin & Vyatkin, 2008; Drechsler & Kühne, 2015). Within the IEC 61499 community, there are also multiple competing development environments that store their application metadata in different formats. The TORUS architecture is modular and was designed to allow the system to be extended through interfaces to support more function block development systems as well as alternative requirements management systems. Extending TORUS to work with other cyber-physical system architectures could be a worthwhile path to investigate in future research.

Chapter 4

Requirements Modeling

Requirements need to be organized in a systematic manner, especially when they represent large requirements sets. To study the nature of traceability within complex cyber-physical systems, we need to gain an understanding not only of the needs expressed in the requirement statements but also of the context that the application will be operating in.

This section explores the raw specification further and refines it through a series of iterations to build a robust requirements model. The Sparx Enterprise Architect Requirements Management system (Architect, 2010) was used to create a view of the system using the SysML modeling language. The data held in this model was mined, showing how trace footholds or hooks could be established for the splices to use later.

The Workpiece Color Sorter is a practical example of a cyber-physical system that illustrates how the trace linkages from the requirements through to the implementation could be constructed. While this device initially appears to be a trivial example, it quickly becomes clear that the pre-RS descriptions of its anticipated behavior fail to capture significant non-functional and safety-critical requirements. These classes of requirements often only emerge as the specification is refined and analyzed in greater depth.

4.1 Gathering the initial pre-Requirements information

The first information about what the Workpiece Color Sorter was expected to be able to do was gathered during an initial informal discussion. One of the supervisors for this research was asked to act as a client and present a very rough outline of the device. The hand-drawn sketch shown in Figure 4.1 was created during this discussion as well as a simple, verbal description of the functionality. The client used the diagrams as a way of both explaining and exploring their thinking rather than as concrete design objects that they had prepared earlier:

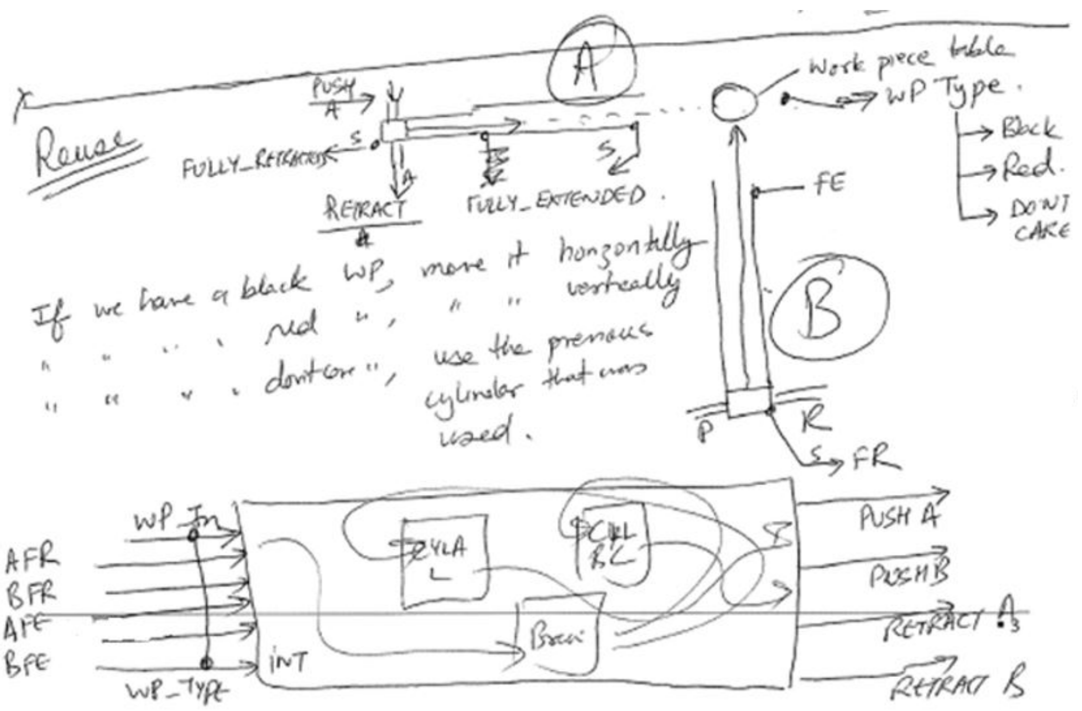


Figure 4.1: The First Diagram Drawn by the Client Illustrating Some of the Functionality.

The second sketch shown in Figure 4.2 was created during the same session. At the time, the client was giving a free-form, natural-language explanation of the device's operations:

"The sphere moves into the workspace. The system checks the color of the workpiece. If

the sphere is black, the horizontal piston moves it horizontally off of the workspace. If the workpiece is red, the vertical piston moves it vertically. If it is any other color, the direction the thing is moved in is exactly the same direction the last one was moved in."

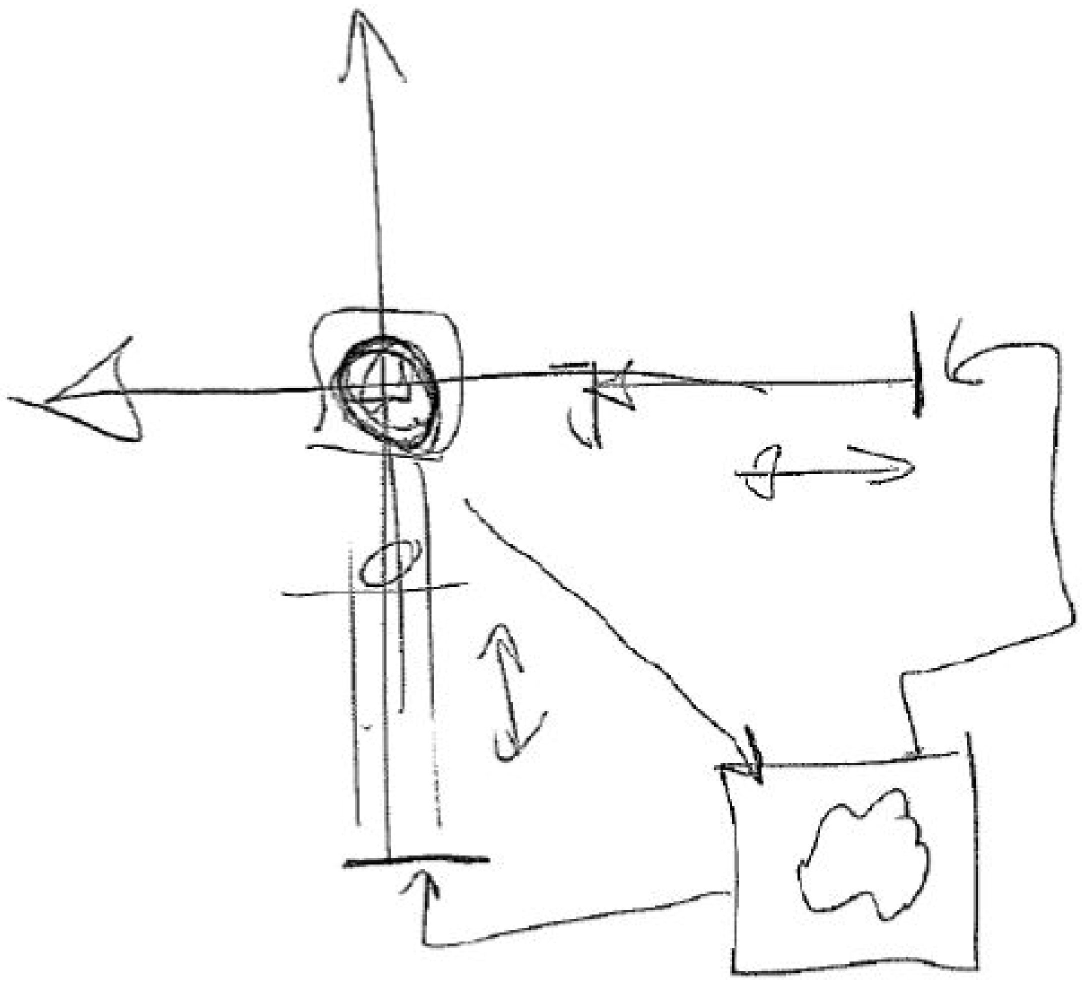


Figure 4.2: The Second Diagram Drawn by the Client to Further Explain the Functionality.

The format of these early-stage requirements is typical of those found in real-world projects. In this phase, the job of the requirements engineer is to translate the view that the stakeholders have of their world, colored by their implicit domain knowledge, into a view that the developers can comprehend (Sandhu, 2015). These statements express intent without explaining how the piston or software is intended to implement the functionality. For example, no mention is made of safety constraints. Other Non-Functional Requirements such as the number of spheres the device was expected to handle per second or the physical size and weight of the spheres were not provided. These are typically details that the requirements analyst would be expected to elicit as their shared understanding of the client's needs is built. Terminology such as *workpiece* and *workspace* may later become elements of a glossary of shared terms or *ontology* that describes the solutions knowledge environment. Note the ambiguity in sentences: the terms *sphere*, *thing* and *workpiece* are used interchangeably. We can summarize these abstractions as:

- *Workpiece*. The colored sphere that is going to be shifted or moved according to the system's rules.
- *Workspace*. The physical region within the bounds of the devices activities where the Workpiece can be controlled.
- *Piston*. A device for moving a Workpiece horizontally or vertically within and out of the Workspace.

These formalizations help to better define objects and remove ambiguity from the descriptions given by the stakeholder. Gervasi and Zowghi (2010) cite such ambiguity as the most significant cause of the later failure of traceability strategies to identify correct linkages. However, the statements still miss one scenario related to the initialization of the system; the first time the system becomes operational, there will have been no previous workpiece. If the first workpiece presented is neither black nor red, the direction to move it in is indeterminate. During the requirements refinement, additional requirements will need to be created to handle this situation.

4.2 Refining the requirements

The pre-RS natural language text can be made more formal by iterating each requirement and expressing them in a notation that is more precise. The following sections examine Requirements Specification Languages (RSL's) including Template-Based Requirements Specifications and the use of the Systems Modeling Language *SysML*. SysML is an extension of UML 2.0 that introduces diagrams more applicable to cyber-physical systems design (OMG, 2016a).

4.2.1 Template-Based Requirements Specifications

Natural language requirements are typically influenced by implicit domain knowledge but are not necessarily constrained enough by it (Kamsties et al., 2001). Using a guided natural language template, a dictionary of prescribed words enables the requirements to be expressed in a more regular, repeatable way. *Boilerplates* are templates, written in the terminology of the RSL, that can be parameterized with objects and events. *Events* are activities that the objects can either facilitate or participate in. Most specifications require a minimal set of compatible templates to properly express the full range of scenarios. We developed a draft set of three templates that are similar to Use Cases:

1. The **<stakeholder>** shall be able to **<capability>** within **<performance>** of **<event>** while **<operational condition>.**
2. The **<system>** shall **<action><entity>** with **<entity>** when **<operating condition>.**
3. Whenever**<event>** occurs **<condition>** holds during the following **<interval>.**

Boilerplates such as these typically address one of three main categories of activities:

1. **Capability Boilerplates** detail tasks or activities that the system is capable of performing.

2. **Requirement Boilerplates** express both functional as well as non-functional requirements. Non-functional requirements specify compliance with measurable metrics such as speed or power usage. Functional requirements define expected capabilities.
3. **Constraint Boilerplates.** Things the system should not do. In the scenarios stated so far, there is no indication that the vertical or horizontal pistons will have software checks or mechanical interlocks that stop one piston from starting to move while the other piston is in motion.

4.2.2 Building the draft ontology

Ontologies are more systematic glossaries of terms that capture domain knowledge in structured ways. They are often very fluid structures while they are being established, explored and refined. This is their normal and expected behavior during requirements elicitation (Noy, McGuinness et al., 2001). As the requirements become more constrained, ambiguous terminology is removed systematically as it is replaced with correct classification elements, known as *ontology classes*. The category groupings that describe each activity or property may also change. During the first stages of the creation of our post-RS, the following ontology classes emerged:

- **Objects:** SolidStateDetector, WorkpieceRed, WorkpieceBlack, WorkpieceOther, Workspace, PistonHorizontal, PistonVertical, LastWorkpiece.
- **Events:** ColorDetected, Presents, Extends, Retracts, Reaches.
- **Attributes:** TopLimit, BottomLimit, LastWorkpiece, IsLocked, IsExtended, IsRetracted, IsLastUsed, IsBlack, isRed, isOther.

4.2.3 Creating and Using the Boilerplate Templates

While the Boilerplates that follow in Table 4.1 were being refined, the idea of expressing the properties of the Workpiece as *objects* was considered. This would have resulted in entities such as **Workpiece.Red** and boolean properties such as **Workpiece-Black.IsLastUsed**. This became clumsy and convoluted initially but perhaps warrants further investigation later. The initial concern was that it introduced concepts that were too abstract for all stakeholders to comprehend.

Table 4.1: Boilerplate Templates

Name	*Format*
BP-01	whenever **<object>** [not] **<attribute>** [and] **<object>** [not] **<attribute>**] then **<object><action>** until **<object>** [not] **<attribute>**.
BP-02	whenever **<object>** [not] **<action>** [and while **<object>** [not] **<action>**] then **<object>** **<attribute>**.

The first set of requirements shown in Table 4.2 describe the operation of the horizontal piston. All black Workpieces are moved by this piston.

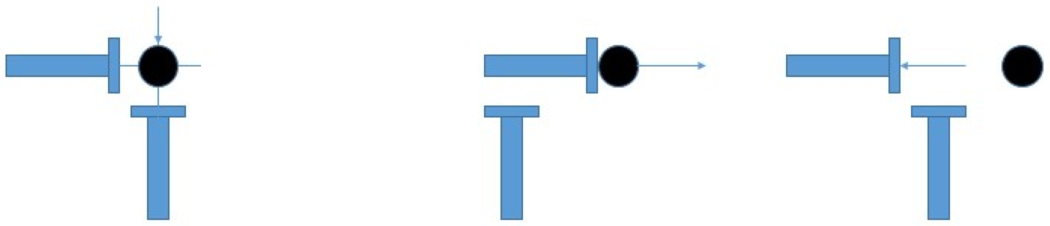


Figure 4.3: Operation of the Horizontal Piston

Table 4.2: Requirements for the Operation of the Horizontal Piston

ID	*Requirement*	*Boilerplate*
R1	whenever Workpiece IsBlack then PistonHorizontal Extends until PistonHorizontal isExtended.	BP-01
R2	whenever PistonHorizontal isExtended then PistonHorizontal Retracts until PistonHorizontal IsRetracted.	BP-01
SR1	whenever PistonHorizontal not IsRetracted then PistonVertical IsLocked.	BP-02
FR1	whenever Workpiece IsBlack then PistonHorizontal IsLastUsed.	BP-02

The operation of the vertical piston, which manages red workpieces is similar. Note the additional functional requirements FR1 and FR2 that remember the colour of the last workpiece that was moved. The safety requirements SR1 and SR2 ensure that a piston can only move when the other piston is fully-retracted.

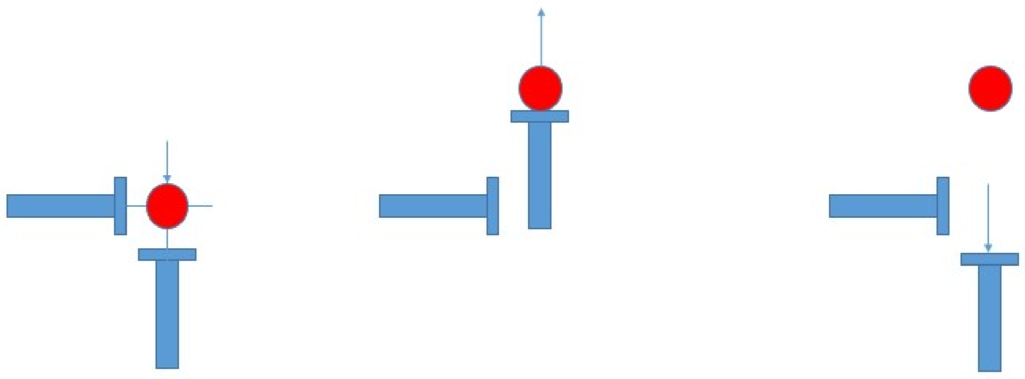


Figure 4.4: Operation of the Vertical Piston

Table 4.3: Requirements for the Operation of the Vertical Piston

ID	*Requirement*	*Boilerplate*
R3	whenever Workpiece IsRed then PistonVertical Extends until PistonVertical isExtended.	BP-01
R4	whenever PistonVertical isExtended then PistonVertical Retracts until PistonVertical IsRetracted.	BP-01
SR2	whenever PistonVertical not IsRetracted then PistonHorizontal IsLocked.	BP-02
FR2	whenever Workpiece IsRed then PistonVertical IsLastUsed.	BP-02

The remaining requirements in Table 4.4 manage the cases when the Workpiece is neither red nor black. Requirement R7 manages the case when no previous workpiece has been moved. This scenario would occur if the first Workpiece encountered after the system is initialized is neither red nor black.

Table 4.4: Requirements for Moving Workpieces of Indeterminate Colors

ID	*Requirement*	*Boilerplate*
R5	whenever Workpiece IsOther and PistonHorizontal IsLastUsed then Workpiece IsBlack.	BP-01
R6	whenever Workpiece IsOther and PistonVertical IsLastUsed then Workpiece IsRed.	BP-01
R7	whenever Workpiece IsOther and PistonHorizontal not IsLastUsed and PistonVertical not IsLastUsed then Workpiece IsBlack.	BP-01

Table 4.5: Requirements for Detecting the Color of the Workpiece

ID	*Requirement*	*Boilerplate*
R8	whenever SolidStateDetector ColorDetected IsRed then Workpiece IsRed.	BP-01
R9	whenever SolidStateDetector ColorDetected IsBlack then Workpiece IsBlack.	BP-01
R10	whenever SolidStateDetector ColorDetected not IsBlack and SolidStateDetector ColorDetected not IsRed then Workpiece IsOther.	BP-01

4.3 Modeling the Requirements in SysML

The boilerplate-format requirements created so far define some of the capabilities of the system. However, each requirement as written here exists as a single, independent and unconnected statement. A list of requirements is not a Requirements Model; there are implicit interdependencies between the actions that must be reflected or expressed through the creation of appropriate model views.

The Unified Modeling Language (UML) is a general-purpose modeling language that is used to describe software applications and their components in a systematic way (Pooley & Stevens, 1998). Modeling languages are semi-formal languages that define the kinds of symbolic elements that can be used to describe a system (Delligatti, 2013). They formalize the allowable *relationships* between the models elements by using *notations* to link the graphical parts of the model. Sets of rules, constructed with elements of the standard UML grammar, ensure that models can be verified as being syntactically-correct or *well-formed*.

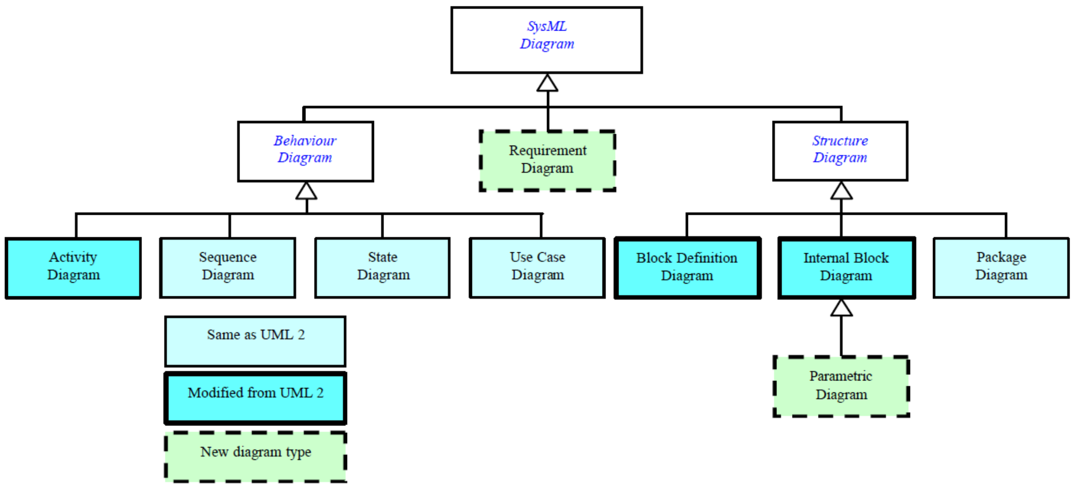


Figure 4.5: The SysML Modeling Language Extensions to the UML (OMG, 2016a, page 167).

The Systems Modeling Language or *SysML* is an extension of the UML created by the Object Management Group (OMG, 2016a). SysML introduced diagrams that are more suitable for describing the characteristics of cyber-physical systems. Figure 4.5 illustrates the Requirement Diagram used in our models to describe the workpiece color sorter in the context of other key diagram types available in SysML.

4.3.1 Characteristics of the SysML Requirement Diagram

A Requirements diagram describes a single requirement, expressed as a Use Case. In its simplest format, the diagram consists of a rectangle with the UML stereotype «requirement». Each requirement is defined by two primary properties:

The *id* is a unique string identifier which carries the primary reference to this requirement. Naming rules are not mandated in SysML but are usually conventions agreed upon by the modelers themselves.

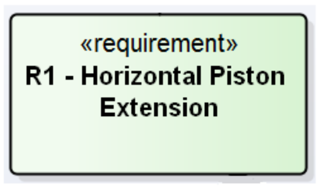
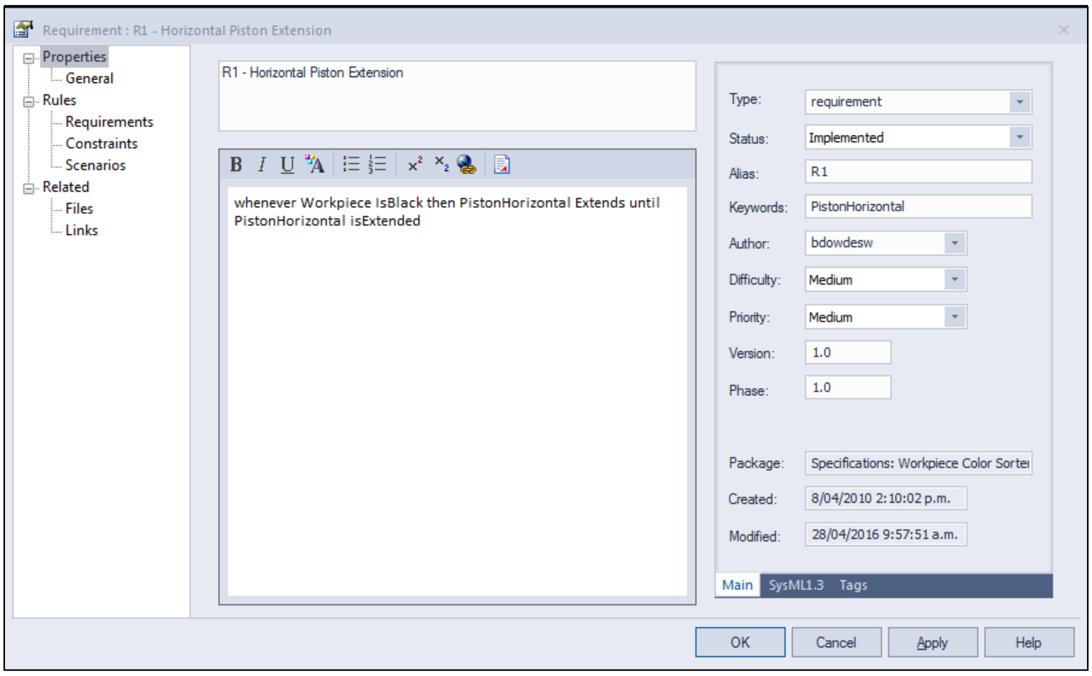


Figure 4.6: SysML Requirements Diagram

The *text* carries the free-form description of the requirement. The model constructed for use in TORUS also includes the full boilerplate text that expresses the capabilities and constraints that the requirement imposes on the named objects.



R1 - Horizontal Piston Extension

whenever Workpiece IsBlack then PistonHorizontal Extends until PistonHorizontal isExtended

Type:	requirement
Status:	Implemented
Alias:	R1
Keywords:	PistonHorizontal
Author:	bdowdesw
Difficulty:	Medium
Priority:	Medium
Version:	1.0
Phase:	1.0
Package:	Specifications: Workpiece Color Sorter
Created:	8/04/2010 2:10:02 p.m.
Modified:	28/04/2016 9:57:51 a.m.

Figure 4.7: Specifying Requirement R1 in Sparx Enterprise Architect.

The Requirements diagram also contains attributes to detail how a requirement is derived from or relates to another requirement, how elements verify each other and how they can be nested to contain each other. Requirements can also trace each other and other types of SysML artifacts by specifying a «trace» relationship between themselves. However, the scope of such trace elements is constrained to be within the model itself.

They cannot reach out dynamically to verify the existence of concrete application artifacts. As a consequence, trace elements within requirements models are limited to being design or historical records of what is believed to be within the application system. They must be updated manually or by importing external information from another traceability system. In contrast, TORUS establishes an external repository which can mine both the requirements model and the application artifacts. Hence, in the models profiled here, no use has been made of the «trace» capabilities of Enterprise Architect. Figure 4.7 shows the data captured for requirement R1 in our model. The stereotype of this SysML element is «requirement». The *Alias* attribute contains our unique requirements identifier. Each requirement has been assigned to one or more requirement sub-groups. The *Keywords* attribute has been used to assign this requirement to the horizontal piston hence its sub-group is PistonHorizontal. The complete SysML model for the workpiece color sorter is shown in Figure 4.9. Constraining the group identifiers to be valid ontology classes will improve the integrity of our later trace analysis when the TORUS view is focusing only on requirements and function blocks that manage a section of interest, such as a single piston.

4.3.2 Requirements and Unit Tests

Test-Driven Design is a natural complement for MBSE. During the creation of the workpiece color sorter later, unit tests were created for each requirement and documented within Enterprise Architect to keep the model up-to-date.

Since the unit tests exist in a natural hierarchy within the requirements model, there is an implicit link to the requirement they are designed to test. Jamro (2014) details a similar approach, developing a Structured Text language extension for unit tests applicable to earlier IEC 61131-3 function blocks. Within our model, each unit test also specifies the objects that it wishes to instantiate during the test. Figure 6.4 shows the code used to

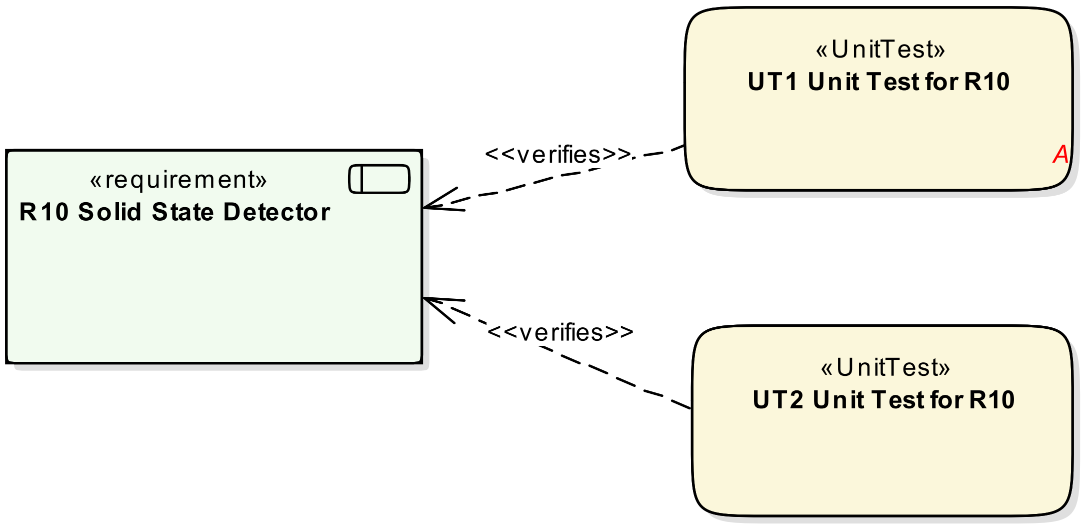


Figure 4.8: Unit Tests for Requirement R10 within the Requirements Model.

implement two unit tests. Chapter 5 explains the object-orientated nature of function blocks in more detail.

The Unit Tests therefore present a way of creating and updating traceability linkages automatically by facilitating a way to analyze the function blocks at a code level. Figure 4.8 illustrates the hierarchy of unit tests within SysML. These traceability hooks are explored in more detail in Chapter 6.

4.3.3 Exchanging Data Using XMI

TORUS mines the data in our requirements model by using the exported XMI (XML Metadata Interchange) file that Enterprise Architect creates (OMG, 2016b). This is an XML-compliant data exchange standard that defines the metadata attributes that are able to carry SysML model data out of the requirements modeling environment for use by other systems.

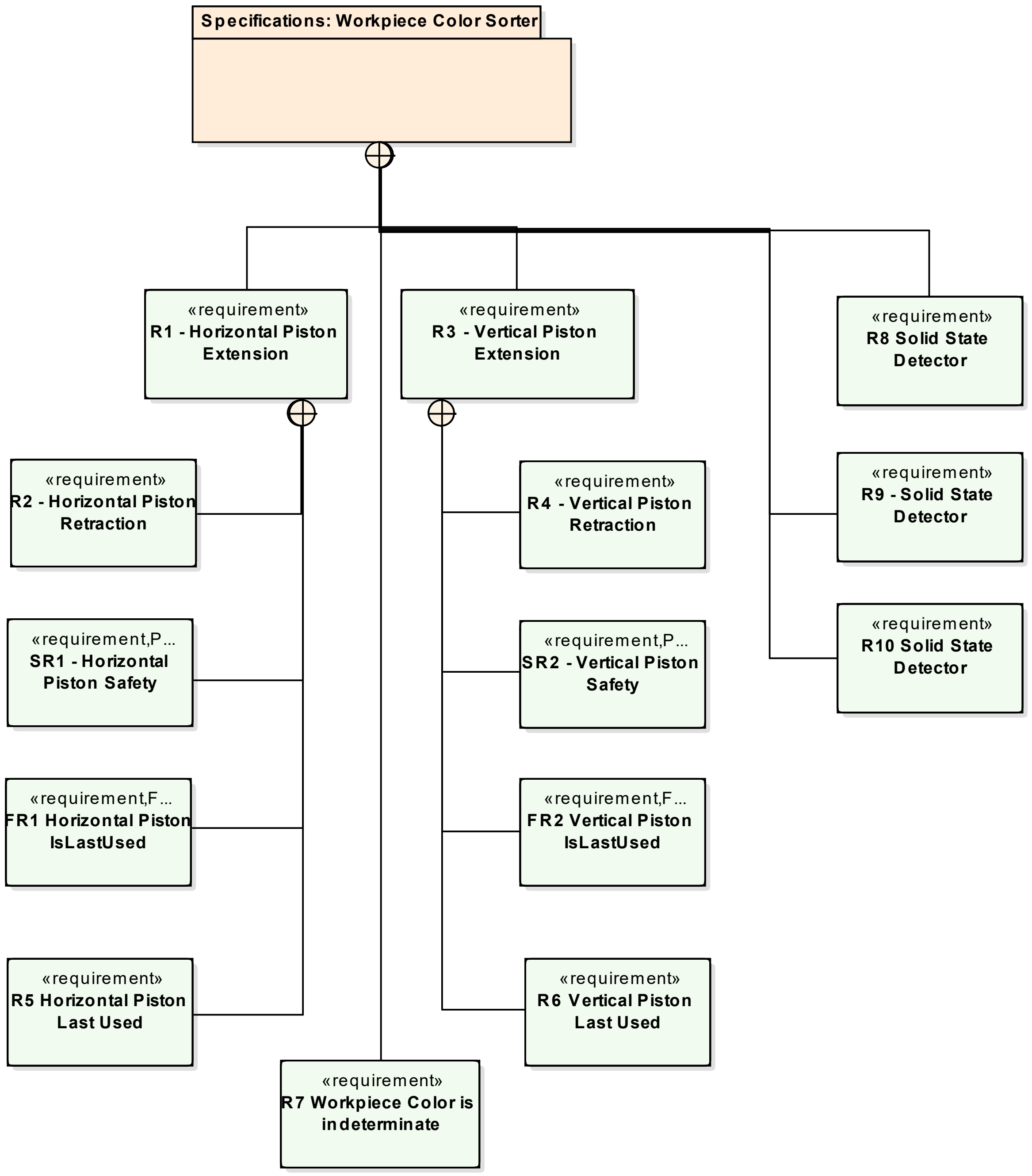

Specifications: Workpiece Color Sorter
«requirement» R1 - Horizontal Piston Extension
«requirement» R3 - Vertical Piston Extension
«requirement» R8 Solid State Detector
«requirement» R2 - Horizontal Piston Retraction
«requirement» R4 - Vertical Piston Retraction
«requirement» R9 - Solid State Detector
«requirement,P... SR1 - Horizontal Piston Safety
«requirement,P... SR2 - Vertical Piston Safety
«requirement» R10 Solid State Detector
«requirement,F... FR1 Horizontal Piston IsLastUsed
«requirement,F... FR2 Vertical Piston IsLastUsed
«requirement» R5 Horizontal Piston Last Used
«requirement» R6 Vertical Piston Last Used
«requirement» R7 Workpiece Color is indeterminate

Figure 4.9: The Complete Workpiece Color Sorter SysML Requirements Model.

The schema for this file was studied to determine how Enterprise Architect encodes the data in the XMI file. Key to relating all the fields for a requirement is a unique GUID (Globally-Unique Identifier). Changing any data within an existing Requirements Diagram preserves this GUID. TORUS imports this GUID and uses it to determine if any attributes have changed. Tests showed that deleting a requirement completely in Enterprise Architect and then re-creating an identical one correctly results in the creation of a new GUID.

4.4 The Formal Definition of a Requirement

The requirements of the workpiece color sorter are not independent. There is a degree of interrelation between them due to necessary dependencies that logically express functionality needed to make the requirements complete. For example, while SR1 and FR1 were not presented by the original stakeholders, their existence was justified by later analysis. However, these requirements are secondary to the primary requirement R1 and they imply a degree of hierarchy.

In addition to boilerplate requirements, we may add requirements specified in other languages, Unit Tests, Use Cases, Scenarios, and Acceptance Tests. Also, in many cases, there may be a need for requirement-to-requirement traces, especially for historical linkages. It is therefore assumed that all requirements engineering artifact's are contained in the set SE of system elements.

Many requirement modeling strategies, including the CESAR Requirements Management Model, organize requirements into hierarchical *Requirements Trees*. The requirements tree defined in Definition 4.4.1 RT expresses the complete workpiece color sorter requirements set shown in Figure 4.9. The formalism we use to express our requirements tree is a *directed graph* (Mäder, Gotel & Philippow, 2009a).

Definition 4.4.1 (RT) *A Requirements Tree RT is a tuple $RT = \langle R, r_0, E_R \rangle$ where:*

1. *R is a set of vertices's requirements.*
2. *r_0 is the root requirement.*
3. *E_R is a set of edges which are ordered pairs of elements of R.*

The finite set $RF = \{RT_1, \ldots\}$ of all requirements trees is called the requirements *forest*, and represents the requirements organization for a system. Later chapters present the formal definitions of function blocks and splices that build on this definition of a requirement.

4.5 Discussion and Concluding Remarks

This section has shown how free-form initial text of the user's needs can be formalized into models that are more precise, richer in content and expose deeper relationships between the requirements. By identifying unique identifiers for each requirement, hooks that were later exploited by TORUS provided ways of building traceability pathways.

Delligatti sounds a note of caution about how easy it is for stakeholders to place too much emphasis on the value of models (Delligatti, 2013, page 9). There is a perception that Model Based Systems Engineering (MBSE) somehow simplifies the task of creating software deliverables, that it makes each task easier and reduces cost. In reality, scaling models to cope with thousands of requirements and creating views of systems that have many distributed components is not easy. For organizations that are beginning to adopt MBSE, startup costs can be high (Estefan, 2010).

However, models whose construction time and cost is appropriate for the project do yield quantifiable benefits (Haskins, 2011). Automated traceability cannot be achieved without a reference source model as a destination to sink traces to and from. Part of the solution to managing complexity and scale arises from the ability to partition requirements and focus traces on identified trouble spots. The ability to repeatable generate traces with little effort as problems are addressed also relies on the ability of the model to remain in-step with all changes. Managing this information in the context of a model is the surest way to maintain and refine changes that impact hierarchal requirements.

Conway's hypothesis states that an organization's design for a system will inevitably realize an entity whose structure is a copy of the organization's communication structure (Conway, 1968). The implication is that models that accurately represent the desired system and create identifiable silos of activity should promote and facilitate communication between stakeholders who have an investment in that area of the system. However, the corollary is that if the model does not sit comfortably alongside the teams current culture, it will neither find acceptance nor be used. The literature suggests that many, highly-detailed initial specifications cease to be kept up-to-date during the lifetime of a project and quickly become obsolete (Brooks, 1975).

The next section, Chapter 5, details the construction of the workpiece color sorter, showing the use of the different types of function blocks that can be created and connected to craft the complete application. In the same way we did during the creation of the requirements model, potential traceability hooks are identified in the algorithms and entities within nxtStudio that were later used within TORUS.

Chapter 5

Modeling Cyber-Physical Systems in IEC 61499

IEC 61499 is a highly modular and compositional architecture, allowing efficient reuse and reconfiguration (Dubinin & Vyatkin, 2008). Applications are constructed by connecting different types of *function blocks* together typically using Model-Driven Development and Test-Driven Design approaches (Hametner, Kormann, Vogel-Heuser, Winkler & Zoitl, 2013). Software applications are sliced and deployed onto available hardware devices and the communication between slices is carried out via standard interfaces.

This chapter profiles the creation of the workpiece color sorter within the nxtStudio Integrated Development Environment (IDE) (nxtControl GmbH, 2016a). The previous chapter confirmed that there were hooks within the requirements model that our traces could connect to. While building the workpiece color sorter, similar connection points within nxtStudio were identified.

Formal definitions of IEC 61499 Basic Function Blocks, Function Block Networks and Composite Function Blocks in the context of the workpiece color sorter application are presented. This provides a robust system topology that TORUS can exploit to facilitate requirements traceability. The work focuses on the structure and syntax of the applications themselves rather than their execution semantics. Formalizations of the execution models and semantics for IEC 61499 can be found in other works such as Vyatkin (2013), Vyatkin (2009), Sinha (2016) as well as Lindgren and Linder (2015).

5.1 IEC 61499 Function Blocks

IEC 61499 function blocks are objects. As such, they demonstrate the most important aspects of object-orientation; information hiding through encapsulation and instantiation of multiple, discrete named instances from pre-defined object types. This is in contrast to the previous standards including IEC 61131 (TC65, 1993) which did not encourage the rigor that object-orientation demands.

5.1.1 Basic Function Blocks

The Basic Function Block or BFB is the fundamental unit that is used to construct all other function blocks from. Figure 5.1 shows one of the workpiece color sorter Piston Controllers that has been constructed from a nxtStudio Basic Function Block template:

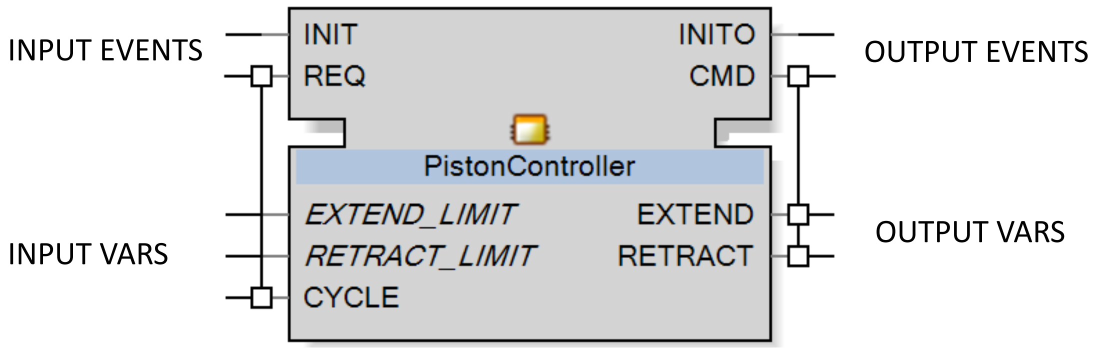


Figure 5.1: Interface to the *PistonController* Basic Function Block.

Each function block is an autonomous device that manages its own set of inputs and outputs which have been configured to support the custom behavior of that component.

These connections are used to exchange control signals and data with the other function blocks that they are connected to. Input Events are command channels that inform the block that new data is now present on its own inputs that is ready to be sampled. In Figure 5.1, the REQ input event is used to inform the internal logic of the Piston Controller block that new data is available on the CYCLE input. Input connections are instantiated internally at the module-level within the function block as data variables whose types include integers, real numbers and strings. Within algorithms, private variables whose scope is restricted to that algorithm can also be created.

Each function block is a finite state machine that maintains its own state transition logic, expressed in a diagram known as the Execution Control Chart or ECC. Figure 5.2 illustrates the ECC for the Piston Controller. Each state transition is managed by software algorithms that are able to evaluate the new data, take decisions and update internal variables. The algorithms are also responsible for presenting new data on the Output Variables of the function block and triggering the control signal to the other function blocks whose Input Events are are connected to the blocks Output Events. Figure 5.1 shows the ECC for the Piston Controller.

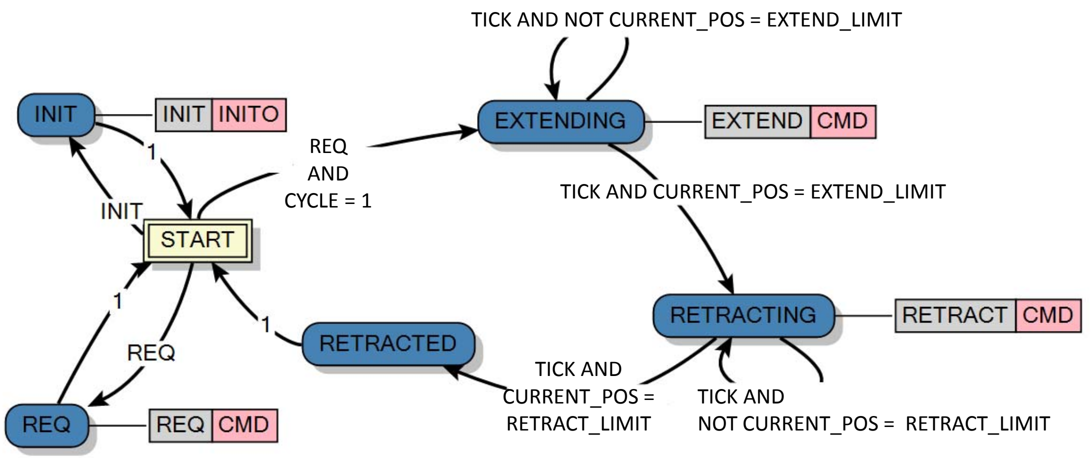

TICK AND NOT CURRENT_POS = EXTEND_LIMIT
INIT
INIT
INITO
1
INIT
START
REQ AND CYCLE = 1
EXTENDING
EXTEND
CMD
TICK AND CURRENT_POS = EXTEND_LIMIT
RETRACTING
RETRACT
CMD
1
1
REQ
RETRACTED
TICK AND CURRENT_POS = RETRACT_LIMIT
TICK AND NOT CURRENT_POS = RETRACT_LIMIT
REQ
REQ
CMD

Figure 5.2: Execution Control Chart or ECC for the Piston Controller Function Block.

The blue rectangles represent finite states that the function block can transition to. The grey rectangles represent each states' *Action Function*, identified by the name of the algorithm that will be executed when the function block transitions to that state.

5.1.2 A Formal Model for a Basic Function Block

Since each function block is a state machine, function blocks are very amenable to formal verification (Guessi, Oliveira et al., 2015). We can define the prototypical Basic Function Block formally as:

Definition 5.1.1 (BFB) *A Basic Function Block BFB is a tuple $BFB = \langle I, V, AG, ECC \rangle$ where:*

1. *$I = \langle EI, VI, EO, VO, WI, WO \rangle$ is a function block interface where EI, VI, EO and VO are finite sets of input events, input variables, output events, and output variables respectively. $WI \subseteq EI \times VI$ and $WO \subseteq EO \times VO$ are sets of input and output associations.*

2. *V is a finite set of internal variables.*

3. *AG is a finite set of algorithms that operate over variables $V_{all} = VI \cup VO \cup V$.*

4. *$ECC = \langle S, T, A \rangle$ is an execution control chart where:*

   (a) *S is a finite set of states with $s_0 \in S$ as the initial state.*

   (b) *$T \subseteq S \times C \times S$ is the set of transitions where $C \subseteq (EI \cup \varnothing) \times \mathcal{B}(\mathcal{V})$ is a set of conditions.*

   (c) *C is the set of conditions that define what causes a state transition defined in the ECC to occur such that:*

$C = EI$ where the transition occurs solely because of an input event.

or

$EI \wedge [$ cond(vars) $]$ because of an event and because of the value

of one of more variables.

or

cond(vars) because one or more variables have reached pre-defined values.

(d) $A : S \rightarrow (AG \cup EO)^*$ is the state action function.

Figure 5.3 shows two instances of the PistonController type that are instantiated to control the horizontal and vertical pistons separately. Each PistonController instance implements exactly the same logic; they differ only in the name of the object that is instantiated and the orientation of the physical piston they control.

Definition 5.1.2 (FB Instance) *An FB instance fb is a pair $(FB, name)$ where:*

1. *FB is a function block type and*
2. *$name$ is a user-defined name for the instance.*

5.1.3 Composite Function Blocks and Networks

A Composite Function Block (CFB) is an encapsulation of two or more function blocks that are connected to each other internally. This is the fundamental methodology used for encapsulation and re-use of Basic Function Blocks within the IEC 61499 standard. IEC 61499 encourages the creation of re-usable customized building blocks and implements this as a practical mechanism for inheritance.

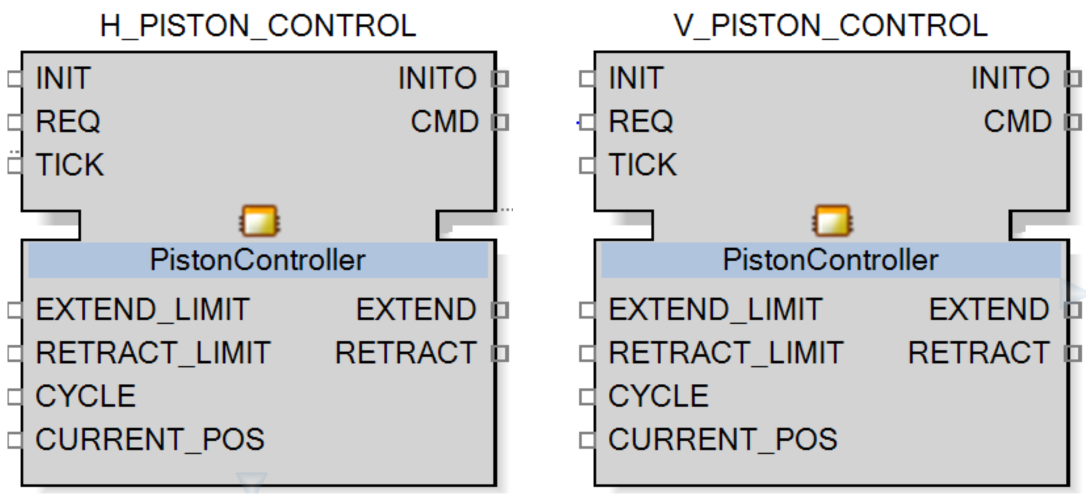


Figure 5.3: Named Instances for the Piston Controller Basic Function Block

True object-orientated polymorphism is not supported but this is deemed to be an advantage in terms of the clarity that this brings to the current encapsulation mechanism. While the topology of a CFB is similar to that of a basic function block, a CFB presents a single interface to the rest of the system, hiding the implementation of the individual function blocks that it is constructed from:

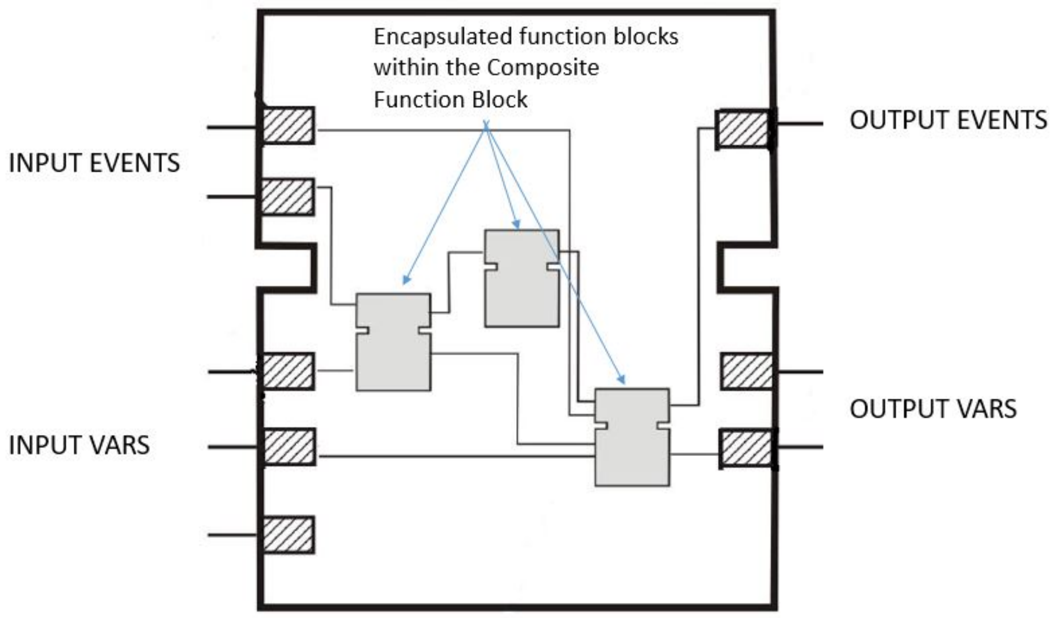

Encapsulated function blocks within the Composite Function Block
INPUT EVENTS
OUTPUT EVENTS
INPUT VARS
OUTPUT VARS

Figure 5.4: The Encapsulation of Function Blocks within a Composite Function Block.

Hence complete applications are built by logically connecting the inputs and outputs of instances via their interfaces to form function block *networks*, consisting of both simple basic function blocks and more complex composite entities. The workpiece color sorter is implemented as such a network, defined as follows in Definition 5.1.3:

Definition 5.1.3 (FB Network) *A FB network is defined as the tuple* $\texttt{FBNetwork} = \langle FB, Conn\rangle$ *where:*

1. FB *is a finite set of function block instances*
2. $Conn \subseteq (FB.EI \times FB.EO) \cup (FB.VI \times FB.VO)$ *is a set of event and variable connections in the network.*
3. $FB.EI$ *refers to the set of all input events of each instance in* FB*.*
4. $FB.VI$ *refers to the set of all input variables of each instance in* FB*.*
5. $FB.EO$ *refers to the set of all output events of each instance in* FB*.*
6. $FB.VO$ *refers to the set of all output variables of each instance in* FB*.*

The $ColorDetector$ block controls a ENV-RGB Solid-State Detector (SSD) camera to sample the color of a workpiece when it is present (Scientific, 2016). The $Sorter$ block triggers either the horizontal or the vertical piston controller blocks that actuate their corresponding pistons.

A CFB allows connections between the elements of its interface and the FB network contained within it. Figure 5.4 shows how the various inputs and outputs of the top-level composite block, such as $INIT$, WP_IN, $INITO$ and CMD are connected to the inputs and outputs of the network contained within it.

5.1.4 A Formal Model for a Composite Function Block

Definition 5.1.4 (CFB) *A composite function block is a tuple:*

$\mathtt{CFB} = \langle I, \mathtt{FBNetwork}, Conn_{CFB} \rangle$ *where:*

1. *I is as defined in Definition 5.1.1, and*

2. *$\mathtt{FBNetwork}$ is a FB network as per Definition 5.1.3, and*

3. *$Conn_{CFB} \subseteq (EI \times FB.EI) \cup (FB.EO \times EO) \cup (VI \times FB.VI) \cup (FB.VO \times VO)$.*

5.1.5 Function Block Topologies

The *topology* of a function block application can be derived as a tree, shown in Figure 5.5:

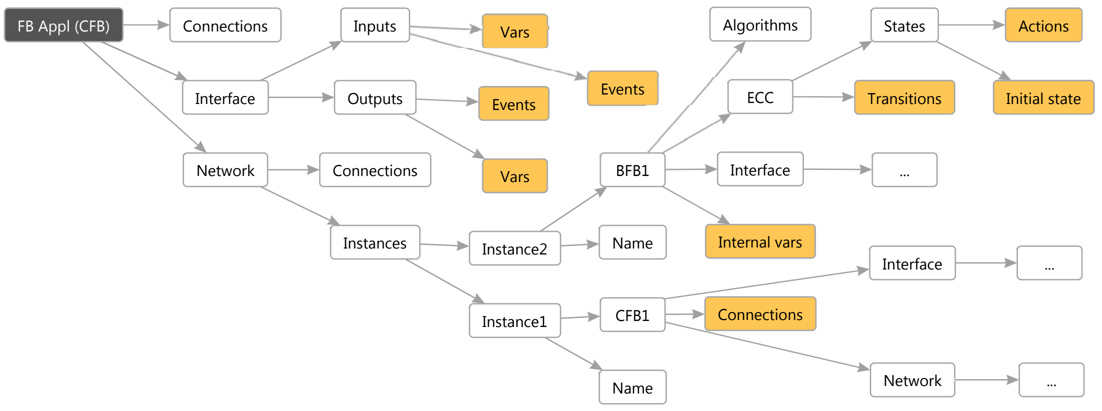


Figure 5.5: The Generic Topology of a Function Block Application.

The topology can be obtained by simply traversing the elements of each tuple starting from the top-level composite function block which represents the application. The topology is useful in creating links between requirements and individual elements in the system. This tree is denoted with the symbol $FBT = \langle SE, fb_0, E_{SE} \rangle$ where SE is the set of all design elements in the topology (function blocks, instances, connections, etc.), fb_0 is the top-level application and serves as the root node of the tree, and E_{SE} models the *contains* relationships between the various elements of the application.

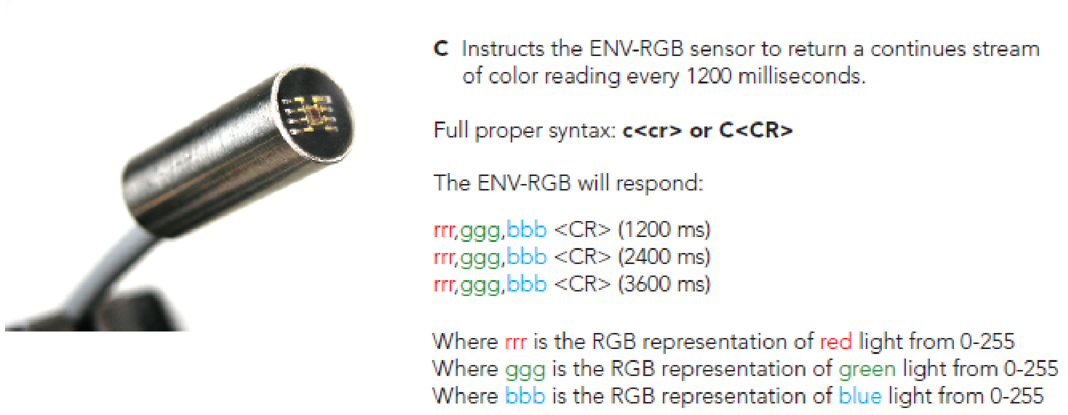
5.2 Crafting the Workpiece Color Sorter in nxtStudio

During the early planning for this research, a number of alternative IEC 61499 Integrated Development Environments (IDEs) were examined. The Function Block Development Kit FBDK (J. Christensen, 2016) was one of the first implementations of IEC 61499 and has been widely cited in research publications (Vyatkin & Chouinard, 2008; J. H. Christensen et al., 2012; Tranoris & Thramboulidis, 2003). The alternative ISaGRAF Workbench was last updated in 2010 but is still widely used by organizations including Schneider Electrical (ISaGRAF, 2010).

The nxtStudio system stood out since it has a strong commercial focus and deploys to a wide range of platforms. The version available to the research community is fully-functional and provides extensive examples and tutorials. It is a complex yet feature-rich development and deployment environment that allows applications to be designed, created and tested on multiple asynchronous IEC 61499 virtual devices, referred to as *SoftPCs*. It also features a comprehensive graphical visualization module that allows the application to be viewed running in real-time. These visualization features allow highly realistic simulations to be built with powerful animation techniques. In production environments, the nxtHMI components created for SCADA PLC's can be deployed directly to compatible visual control consoles (nxtControl GmbH, 2016b). Ultimately, the most compelling reasons for adopting nxtStudio were its stability and that it exposes more promising opportunities to create the traceability hooks that TORUS needs.

5.2.1 Interfaces and Algorithms for the Color Detector

The completed Workpiece Color Sorter is shown in Figure 5.10. The nxtStudio IDE provides different views of the attributes of the function block that is under development. A set of pre-built basic function block templates are available within nxtStudio that are used to clone new function blocks from. The Color Detector function block illustrated here was based on the simplest of these templates. It was customized to build the interface to the solid-state camera. This section shows the creation of this function block and the potential traceability links that were established within nxtStudio during development.



C Instructs the ENV-RGB sensor to return a continues stream of color reading every 1200 milliseconds.

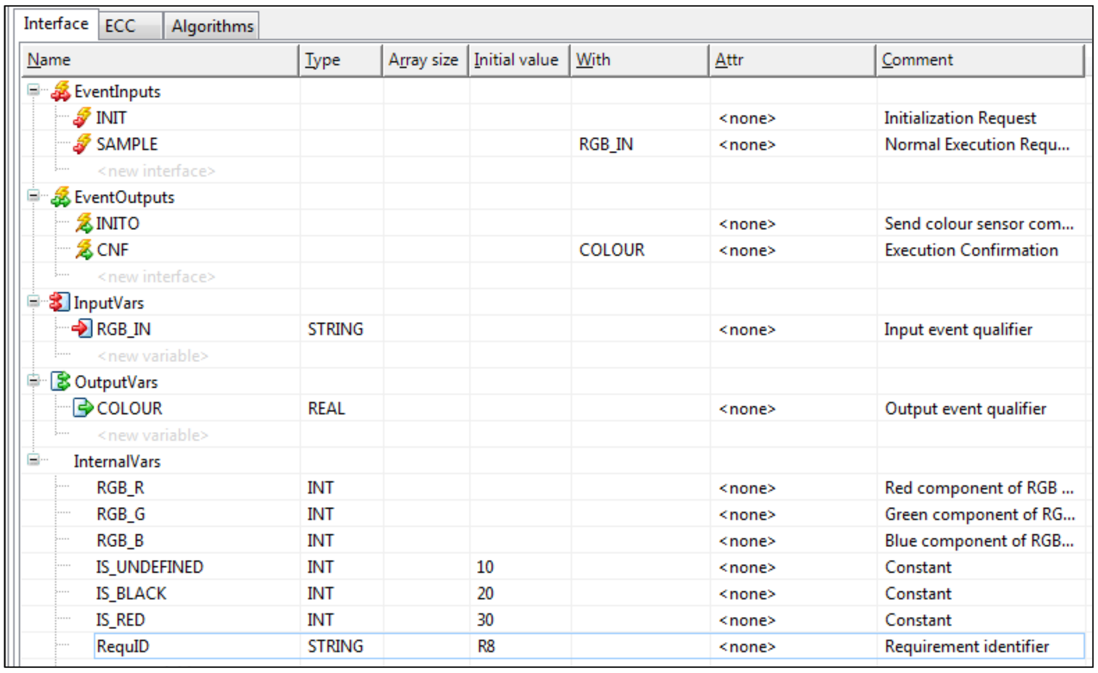
Full proper syntax: **c<cr> or C<CR>**

The ENV-RGB will respond:

rrr,ggg,bbb <CR> (1200 ms)
rrr,ggg,bbb <CR> (2400 ms)
rrr,ggg,bbb <CR> (3600 ms)

Where rrr is the RGB representation of red light from 0-255
Where ggg is the RGB representation of green light from 0-255
Where bbb is the RGB representation of blue light from 0-255

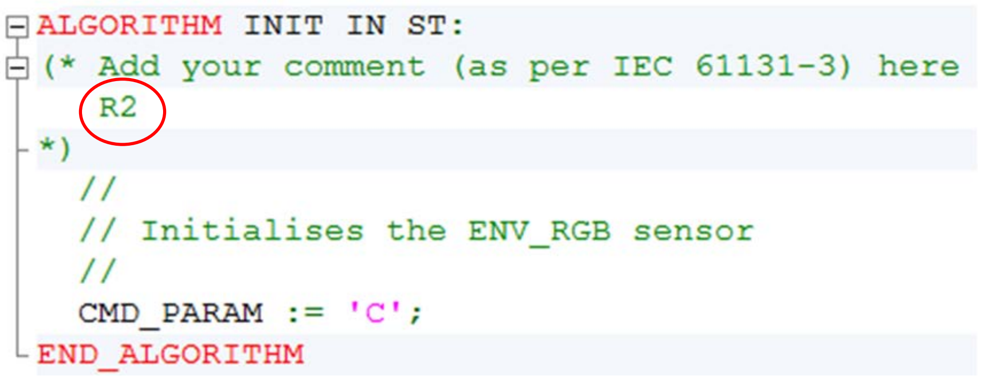
Figure 5.6: The Atlas Scientific ENV-RGB Solid-State Camera Module Data Sheet (Scientific, 2016).



Name	Type	Array size	Initial value	With	Attr	Comment
EventInputs						
INIT					<none>	Initialization Request
SAMPLE				RGB_IN	<none>	Normal Execution Requ...
<new interface>						
EventOutputs						
INITO					<none>	Send colour sensor com...
CNF				COLOUR	<none>	Execution Confirmation
<new interface>						
InputVars						
RGB_IN	STRING				<none>	Input event qualifier
<new variable>						
OutputVars						
COLOUR	REAL				<none>	Output event qualifier
<new variable>						
InternalVars						
RGB_R	INT				<none>	Red component of RGB ...
RGB_G	INT				<none>	Green component of RG...
RGB_B	INT				<none>	Blue component of RGB...
IS_UNDEFINED	INT		10		<none>	Constant
IS_BLACK	INT		20		<none>	Constant
IS_RED	INT		30		<none>	Constant
RequID	STRING		R8		<none>	Requirement identifier

Figure 5.7: The Interface to the Color Detector Function Block.

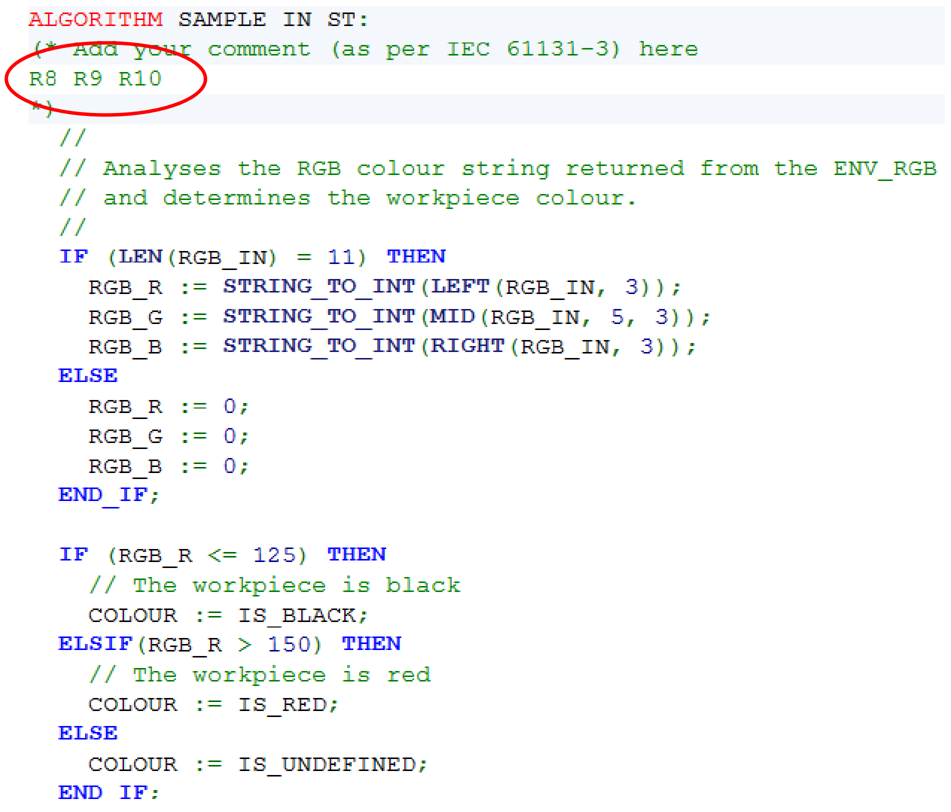
The color sampled by the camera is transmitted via a serial data link as a comma-delimited string in the format described in Figure 5.6. This data is received into the Input Variable RGB_IN via the SAMPLE event. Figure 5.7 shows the nxtStudio *Interface* properties page where each of the function blocks variables, input and output events and associations are specified. An internal variable *RequID* was defined as a string which holds the identifier for one or more of the requirements that this function block helps to fulfill. This traceability artifact provides the most granular link to a requirement since it references the entire function block rather than a specific piece of code. Links such as these are useful when the intellectual property encapsulated within a Composite Function Block is protected and hidden. In these scenarios, the trace cannot penetrate any deeper into the resource.



```
ALGORITHM INIT IN ST:
(* Add your comment (as per IEC 61131-3) here
   R2
*)
  //
  // Initialises the ENV_RGB sensor
  //
  CMD_PARAM := 'C';
END_ALGORITHM
```

Figure 5.8: Trace References within the INIT Algorithm of the Color Sorter Function Block.

The nxtStudio XML-format used to store function block algorithm source code contains a number of predefined XML attributes. Additional attributes cannot be modified or persisted except for the *Comment* attribute. Example traceability references are shown in Figure 5.8 for the INIT and Figure 5.9 for the SAMPLE algorithm. The requirement ID's are shown circled in red.



```
ALGORITHM SAMPLE IN ST:
(* Add your comment (as per IEC 61131-3) here
R8 R9 R10
*)
  //
  // Analyses the RGB colour string returned from the ENV_RGB
  // and determines the workpiece colour.
  //
  IF (LEN(RGB_IN) = 11) THEN
    RGB_R := STRING_TO_INT(LEFT(RGB_IN, 3));
    RGB_G := STRING_TO_INT(MID(RGB_IN, 5, 3));
    RGB_B := STRING_TO_INT(RIGHT(RGB_IN, 3));
  ELSE
    RGB_R := 0;
    RGB_G := 0;
    RGB_B := 0;
  END_IF;

  IF (RGB_R <= 125) THEN
    // The workpiece is black
    COLOUR := IS_BLACK;
  ELSIF(RGB_R > 150) THEN
    // The workpiece is red
    COLOUR := IS_RED;
  ELSE
    COLOUR := IS_UNDEFINED;
  END_IF;
```

Figure 5.9: Trace References within the SAMPLE Algorithm of the Color Sorter Function Block.

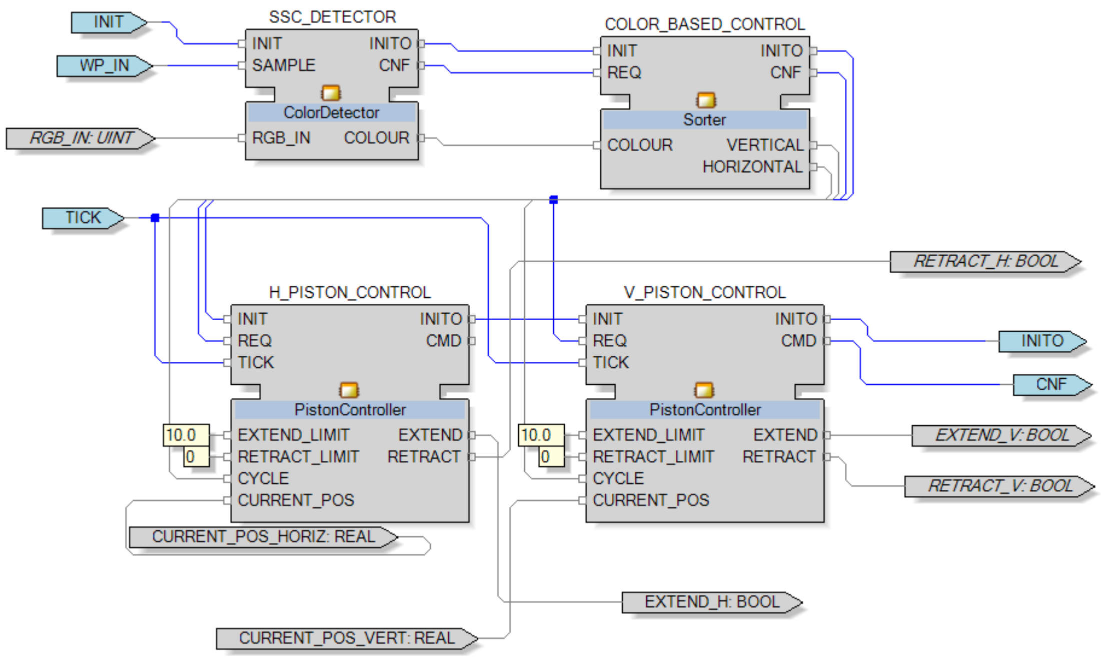


Figure 5.10: The complete Workpiece Color Sorter Controller.

5.3 Conclusions and Further Questions

This chapter on the development of the workpiece color sorter has described the architecture of a typical IEC 61499 cyber-physical system. The formal definition of the function blocks, coupled with the definition of a requirement from Chapter 4 lay the foundation for defining what data a splice needs to acquire to be able to connect these artifacts together in a meaningful way.

It was reasonable to assign unique requirement identification codes during the creation of the requirements model; that is an expected part of the requirements management process. However, ensuring that each function block and algorithm in the workpiece color sorter gets tagged with those same ID codes during development is difficult, time-consuming and error-prone. While it is desirable for developers to do that, nothing in the nxtStudio IDE enforces or encourages such behavior.

In the literature review, the need to manually create and manage traceability linkages was identified as a barrier to the adoption of traceability solutions. In Chapter 4, the use of Unit Tests as pathways to code artifacts was proposed as a way of automatically creating trace linkages. In Chapter 6, the use of both the tagged function blocks as well as code mining within these algorithms is demonstrated during the creation of the TORUS splices.

Chapter 6

Building TORUS

The scope of a traceability initiative should span from the requirements model through to the implementation of application algorithms in both directions. This *bi-directional* nature of traceability is especially important in safety-critical systems, where regulatory compliance mandates deep traceability. This can include traces that span from requirements to design, design to code and code to test cases in both directions (Cleland-Huang, Gotel & Zisman, 2012). Relationships between artifacts, no matter where they reside in the system, should be navigable. Omoronyia and Sindre (2010) demonstrated how automated trace capture and maintenance using ontologies reduced trace creation effort. Facilitating such capabilities using the TORUS framework has been a key focus of this research.

Traceability within highly complex systems presupposes the existence of suitable information at both the requirements and the implementation level that can be mined for matching. Chapter 4 presented a case study of such a requirements model while Chapter 5 described the application that was created from that specification. While unique, systematic identification codes were created for each requirement in the modeling environment, not all entities in the code carried requirement ID's created by the developer. Few application development tools support room in the metadata beneath the code layers that could accommodate such references. Any structured metadata that is created usually resides in external source code change management repositories (Ying, Murphy, Ng & Chu-Carroll, 2004). Its purpose is most often to facilitate version control rather than traceability. In-situ contextual metadata that does get created by developers is often stored as semi-structured or free-form comment lines adjacent to the function code itself, one of the approaches we used. Maletic et al. propose an approach where visible source code within the development system is actually stored in an XML document behind the scenes (Maletic, Collard, Marcus et al., 2002). Their concepts mirror the structure of the IEC 61499 XML standard used by nxtStudio. The nxtStudio development system augments the IEC standard with additional XML attributes that can carry a range of data. These tags are fine-grained and can be created down to the individual Event Control Charts (ECCs) algorithm level. However, we showed that nxtStudio does not allow designers to create their own custom attributes that can be persisted.

TORUS proposes a novel way to create metadata in its own external repository that facilitates the traceability we desire. It exploits our ability to mine both the requirements model and the source code and then uses that information to create persistent linkages. These linkages are designed to survive the inevitable changes that occur during development. This chapter describes the first versions of TORUS and the way the splices capture the information they need to create traceability links. Novel ways for using Unit Tests to automatically create some of this data demonstrate how the requirements model can help to facilitate linkages down to the algorithm code level without requiring the developers to maintain them manually.

6.1 TORUS, Requirements Models and Systems

TORUS is intended to sit between a requirements model and the IEC 61499-compliant XML data structure of the application. The IEC 61499 Function Block standard defines an XML schema that is the mechanism for storing the Function Blocks, their *ECC* and algorithm code. This data structure is later compiled to create the distributed function block application. The way TORUS bridges these entities is shown in Figure 6.1:

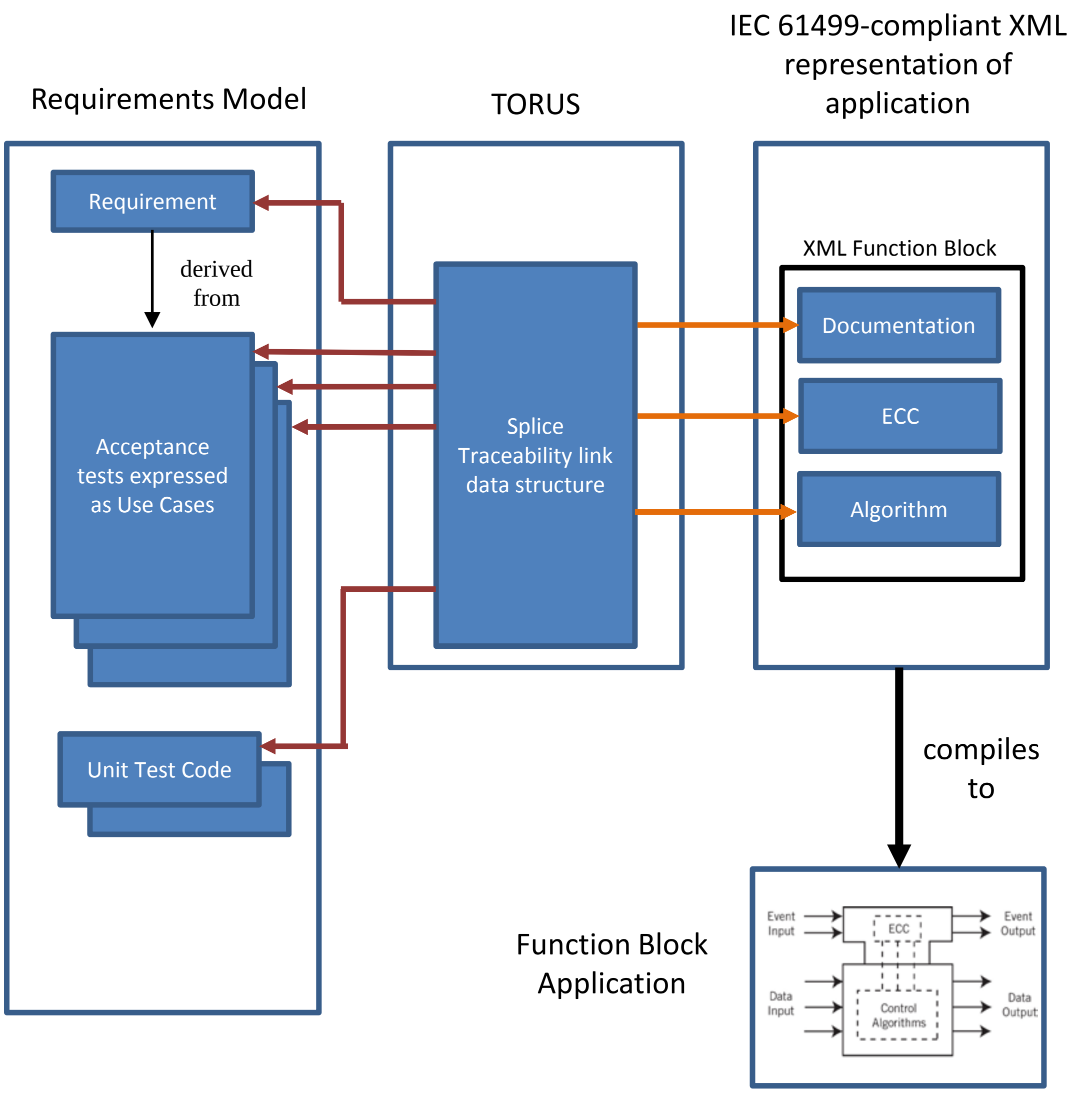


Figure 6.1: TORUS Interactions within the Design Environment.

TORUS operates by mining data from both the requirements and the function block models. The trace linkages TORUS persists are known as *splices* since they join or *splice together* virtual threads to connect each requirement to one or more algorithms that implement them. TORUS builds a structure we have referred to as the *Skein*. This is the set of splices that describes the state of all the requirements found within the system. The Skein allows us to visualize the linkages between entities as if they were the warp and weft of the threads woven into a tapestry. The pattern that emerges when the complete set of discrete statuses is analyzed presents metrics which measure how closely the software implements the stated requirements. Skein metrics are discussed further in Chapter 7.

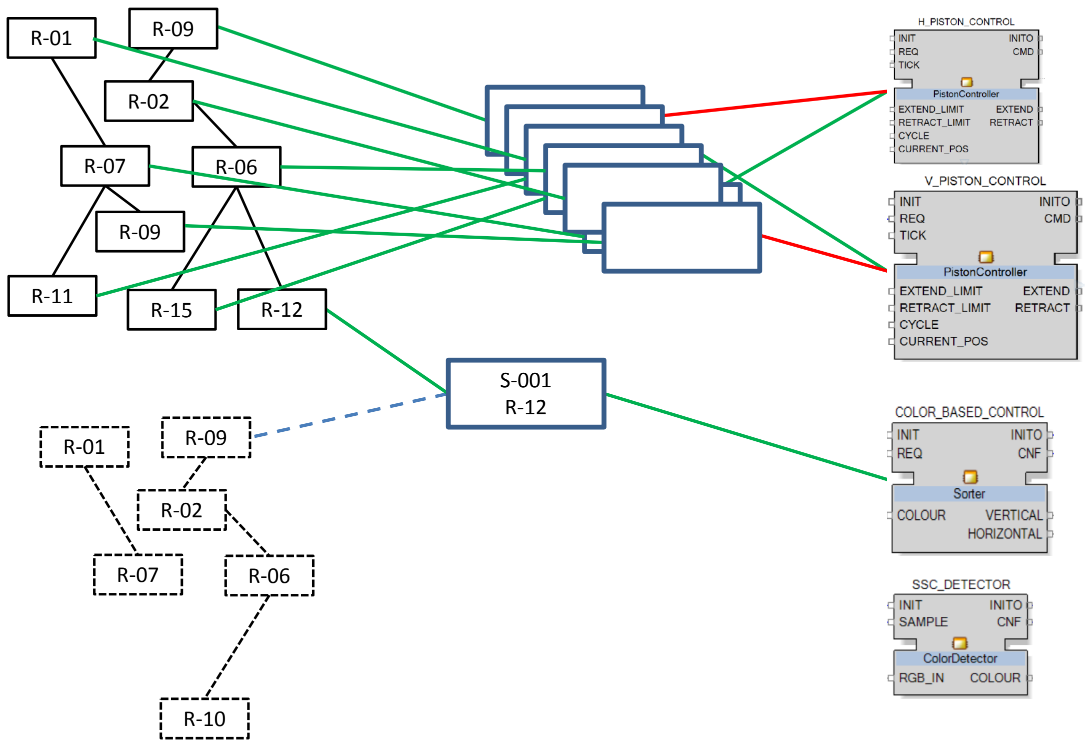


Figure 6.2: Visualising the Skein.

Splices are created automatically during the primary and subsequent analysis passes when a new requirement is encountered. One of the most significant features of TORUS is that the Skein is persistent. Each time the TORUS model is refreshed, new traces are reconciled with previous historical data about that requirement that have been preserved by each splice during previous scans. In Figure 6.2, the splice structures are shown as blue rectangles in the center of the diagram. Green lines reaching back to the left show the pathway to the original requirement stored in the requirements model. Green lines reaching forward to the right indicate that a splice has been able to create a linkage to a function block. TORUS uses evidence such as code ID's or unit test information to make its decisions. Lines that are red indicate that either no artifact could be traced to or that a problem has been identified along the way. Requirement R-12 is shown in-context with its splice S-001. The dotted blue trace reaches back to the pool of historical splices, showing in this case that requirement R-12 was once known as R-09 in an earlier model.

The pattern that emerges helps to ascertain how faithfully the requirements are currently implemented within the application code and how they have changed over time. TORUS does not expect either the requirements management system or the XML function block data structures to be able to store significant amounts of traceability information within themselves. Any non-ambiguous trace information that those systems can provide will obviously be used. However, TORUS expects that a lot of the traceability activity will involve intelligent data mining rather than relying on TORUS-specific metadata that can be created and saved within those external systems.

6.2 The TORUS Interface to Enterprise Architect

TORUS was implemented using a set of Java classes that manage the storage of information captured by analyzing the requirements model and the IEC 61499 application source. The first analysis pass focuses on requirements, creating a linked list of Requirement instances, each carrying the status of a single requirement. Unit Tests for those requirements, if found, are stored in a separate linked list of $UnitTest$ instances. Figure 6.3 shows the TORUS class hierarchy for the Sparx Enterprise Architect interface.

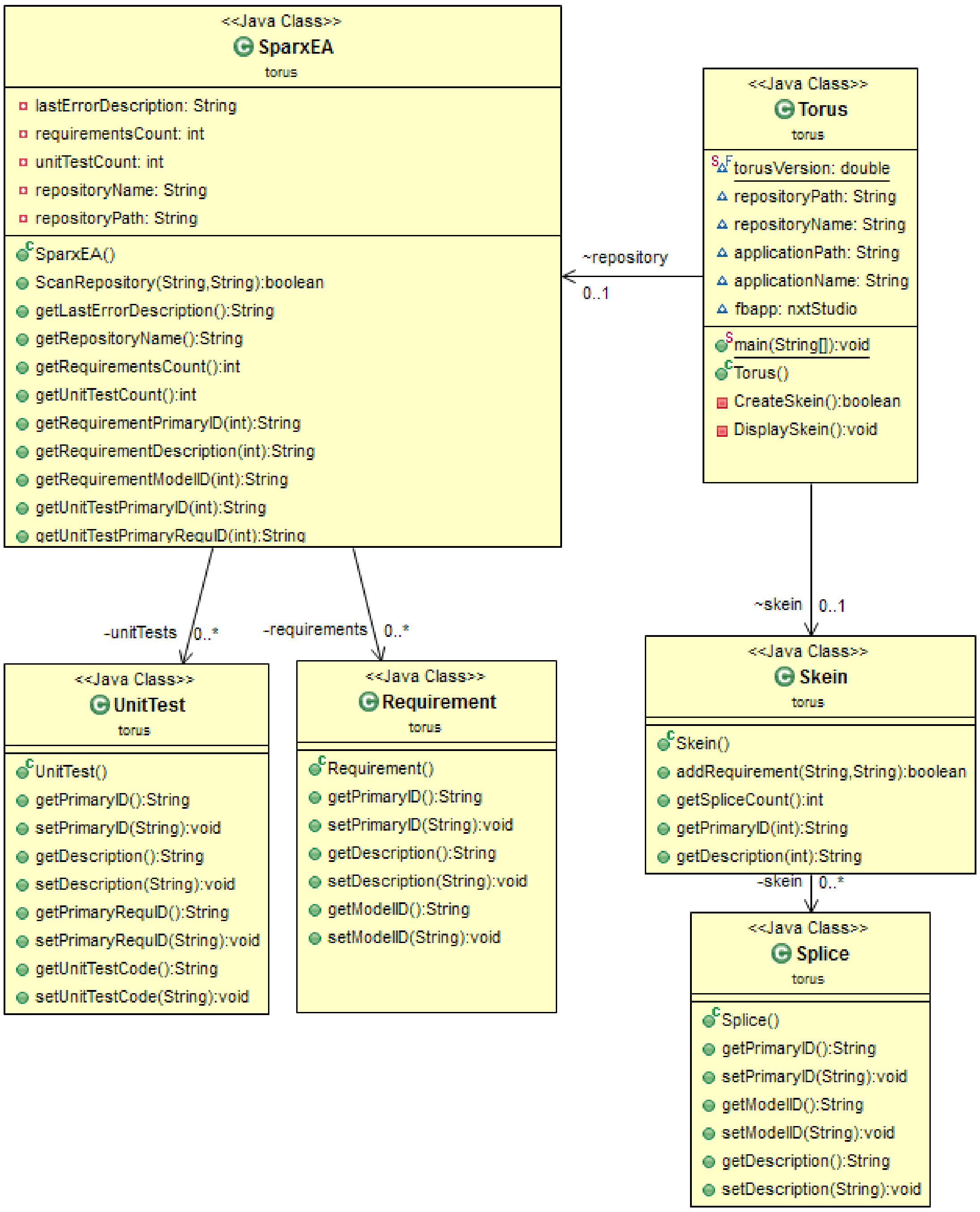


Figure 6.3: The TORUS Class Hierarchy for the Sparx Enterprise Architect Interface.

6.2.1 Abstracting the Requirements Information

The SparxEA class has been designed to decode and analyse the Document Object Model (DOM) that Sparx have used to encode the model data into the XML-format XMI file they export. An identical requirements model created in another requirements management system such as IBM Rational Rhapsody will not necessarily encode the data in the same way.

The XMI-format Enterprise Architect requirements model contains a pre-defined set of XML attributes to carry a set of requirements and their unit tests. When updating the properties of the Requirement instance, a unique model element identifier GUID is saved so that subsequent changes to the model can be identified. Unit Tests are also carried in the XMI file and these are stored in a separate linked-list with the unit test identifiers and their model element GUID's. Each $UnitTest$ class instance holds a single unit test with its source code. A model attribute called *Alias* in the XMI file carries the ID of the parent requirement that this unit test exercises. By the end of this first analysis pass, a set of splice instances will have been created as a linked-list held within the $Skein$ class, each one tracing to a distinct requirement within the Enterprise Architect model.

To enable TORUS to support multiple requirements management systems, a generic set of classes were created to store the splice and skein information. A separate Sparx-specific class $SparxEA$ implements instances of these generic objects, storing information that is important to TORUS in a standardized format. The $SparxEA$ class therefore encapsulates implementation details peculiar to Enterprise Architect that TORUS abstracts into a single, generic model. Figure 6.3 shows the interfaces to the reusable $Requirement$ and $UnitTest$ classes. $SparxEA$ uses these and presents a common, abstracted interface to the main $Torus$ class that instantiated it originally. This instance supplied the name of the Sparx requirements repository file and its location when it triggered the analysis pass initially.

Once the $SparxEX$ instance has processed and stored the requirements and unit tests, it reports the results to the main $Torus$ process. TORUS could just as easily have processed an IBM Rational Rhapsody model by instantiating an appropriate handler class. From that point on after the first analysis pass is complete, the requirements are available to be processed further via common methods. Abstractions such as these allow the TORUS framework to be extended easily.

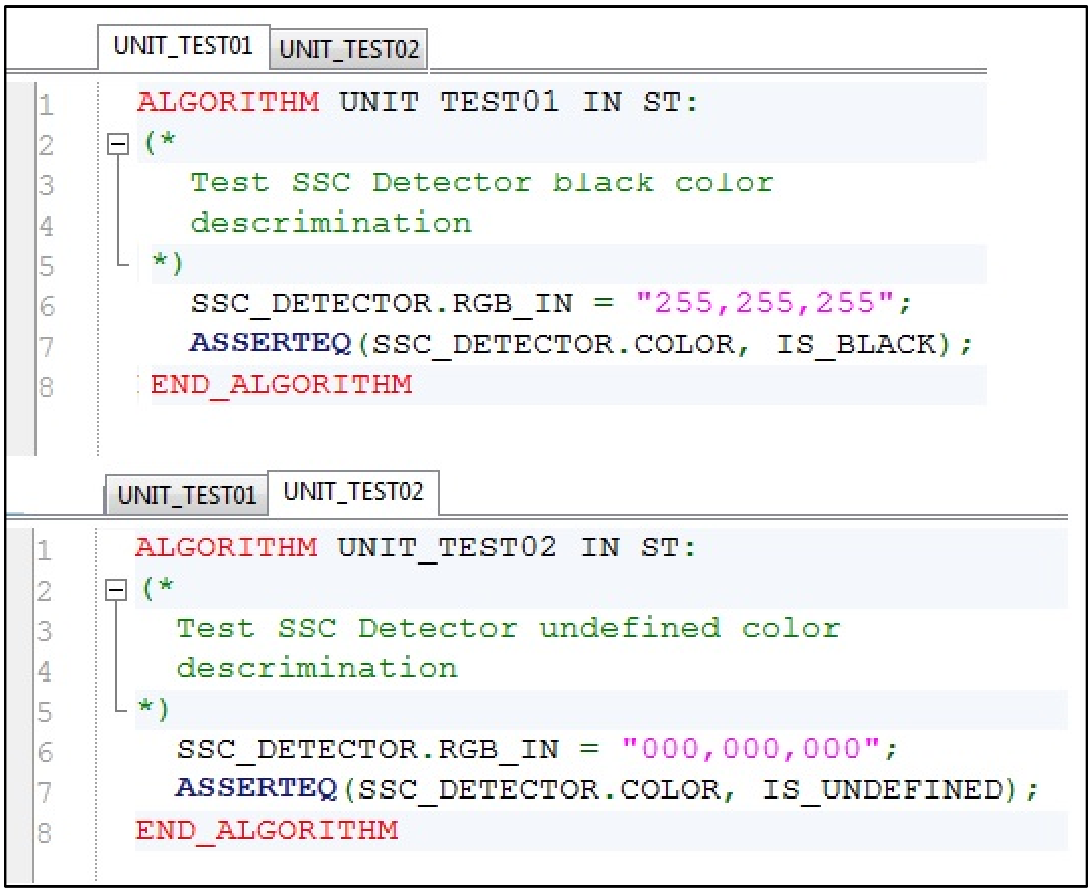
6.2.2 Unit Tests as Traceability Hooks

Automating the creation of traces has remained a key focus of this research. Engineers expect to have to update the requirements model manually when design changes occur in their solution. Having the additional burden of manually re-linking traces is an unwanted task.

Unit Tests offer a unique opportunity to automate trace creation. At present, nxtStudio does not support unit testing within itself so the Structured Text Language that is used for coding function block algorithms in nxtStudio was used as a model for what they might look like. Structured Text Language (STC) is a popular language used for programming Programmable Logic Controllers and is one of the five languages supported by IEC 61131 (TC65, 1993). This is discussed further in the Future Work section of the Conclusions in Section 8.

The $UnitTest$ class was created to analyse and store unit tests that were created in Enterprise Architect. This was used to model unit tests as shown in Figure 6.4. The example sets a value on the input to the camera detector that is black and then performs a JUnit-style Assertion Test on the data output from the function block.

While we cannot actually run these unit tests, they serve as an example of what is possible. The analysis algorithms within the $SparxEX$ class detect the presence of



```
ALGORITHM UNIT TEST01 IN ST:
(*
   Test SSC Detector black color
   descrimination
 *)
   SSC_DETECTOR.RGB_IN = "255,255,255";
   ASSERTEQ(SSC_DETECTOR.COLOR, IS_BLACK);
 END_ALGORITHM
```

```
ALGORITHM UNIT_TEST02 IN ST:
(*
  Test SSC Detector undefined color
  descrimination
*)
  SSC_DETECTOR.RGB_IN = "000,000,000";
  ASSERTEQ(SSC_DETECTOR.COLOR, IS_UNDEFINED);
END_ALGORITHM
```

Figure 6.4: Example Unit Tests for the Solid State Camera Detector Function Block.

objects by parsing the code of the unit tests. In the example shown, the function block $SSC\_DETECTOR$ is referenced. Since each unit test references its parent requirement using an *Alias* attribute in Enterprise Architect unit test diagram, TORUS is able to infer that the function block $SSC\_DETECTOR$ is responsible for at least partially fulfilling requirements R9 and R10. Figure 4.8 in Chapter 4 shows the requirements hierarchy of unit tests in more detail.

6.2.3 Creating the TORUS Skein

The second phase of the analysis iterates through each of the requirements stored in the $SparxEA$ object to create the Skein structure. Within the $Skein$ class, an indexed linked list of $Splice$ objects is maintained. During this phase, a new splice is created only when a new requirement is identified. The status of the splice at this initial stage reflects only that the requirement exists and the number of unit tests found for that particular requirement. A similar method could be adopted for tracking User Acceptance Tests but that has not been implemented in the current version.

The requirements model provides unique identifiers for each entity that it contains. These are retrieved when creating the splices via the interface to $SparxEA$. Later, these will be used to detect and report historical changes in the requirements model since they are also stored as $Splice$ objects. In this way, a splice provides a map that allows later analysis methods to retrieve information about requirements and their relationships to other entities present in the model. In a similar way, subsequent analysis passes will attempt to locate the function blocks that fulfill this requirement.

6.3 The Formal Definition of a Splice

A Splice defines the relationship between a requirement, its use cases, its unit tests and one or more Function Blocks that implement it. The following generalized definition of a splice, illustrated in Figure 6.5, assumes the existence of a set of requirements R and a set SE of existing or planned system elements. The abstraction of the topology of the requirements model discussed previously ensures that this definition of a splice can be used without needing to know the specifics of the system that created it.

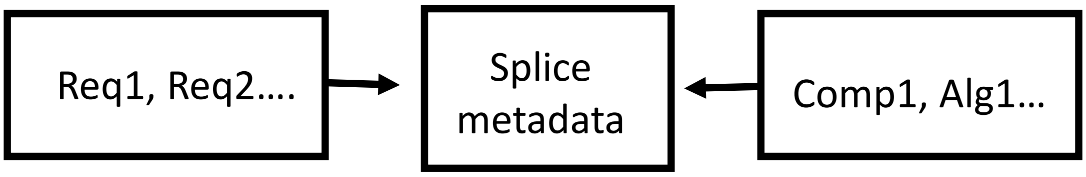


Figure 6.5: An Illustration of a Generic Splice.

Given a CESAR requirements forest $RF = \langle R, r_0, E_R \rangle$ and an IEC 61499 system topology $FBT = \langle SE, fb_0, E_{SE} \rangle$, we can define a splice as follows:

Definition 6.3.1 (Splice) *Given sets R and SE of system requirements and system elements respectively, a splice is defined as a tuple $sp = \langle \mathtt{R}, \mathtt{MD}, \mathtt{SE} \rangle$ where:*

1. $\mathtt{R}$ *is a subset of system requirements where* $\mathtt{R} \subseteq R$,
2. $\mathtt{MD}$ *is a metadata object, further explained in Definition 6.3.2,*
3. $\mathtt{SE}$ *is a set of linked system elements where* $\mathtt{SE} \subseteq SE$.

A set of splices that make up the system *Skein* is denoted by SK.

6.3.1 Adapting TORUS with Customized Metadata

The concept of a trace as we have proposed it here was discussed by Ramesh and Jarke (2001). They noted that traceability strategies often require customizing the solutions since the data needs are not always similar in different industry sectors. The splice structures we have presented employ the metadata MD to provide specific information about a splice such as its type and status. The information can be specified manually by the user but it is more efficient when it can be generated automatically. Definition 6.3.2 shown here was designed specifically for use with function blocks. During TORUS design and performance evaluations, this structure helped to optimize access to the model artifacts when analyzing large requirement sets.

Definition 6.3.2 (Splice Metadata) *The* $\mathtt{MD} = \langle Pr, Tp, Ss, Ds\rangle$ *is a metadata object where:*

1. Pr *is the set of parent splices,*
2. Tp *is the type of the splice,*
3. Ss *is the status of the splice,*
4. Ds *is a free-form, textual description of the splice.*

Since requirements can themselves be seen as elements produced during the design of a system, or $R \subset SE$, Definition 6.3.1 also allows splices between requirements sets. Such splices can be useful in organizing requirements, especially to create historical linkages between requirements and storing the reasons for changes in requirements.

6.3.2 Splice Types for Function blocks

The splice type Tp is the property of the splice which describes the nature of the relationship between the requirement that is being traced and the other artifacts including unit tests, acceptance tests, function blocks and code algorithms that fulfill it. Table 6.1 lists the splice types currently supported within the TORUS framework for function blocks:

Table 6.1: Enumerated Splice Types of TORUS Entities for Function Blocks.

Splice Type	*Description of Entity being Traced To*
IS_UNDEFINED	The splice type is undefined.
IS_REQUIREMENT	A requirement in the Requirements Model.
IS_FUNCTION_BLOCK_ALGORITHM	An algorithm within a Function Block.
IS_UNIT_TEST	A Unit Test stored within the Requirements Model.
IS_FUNCTION_BLOCK_FOR_UNIT_TEST	A target Function Block that has been located that matches the one cited in the referenced Unit Test stored within the Requirements Model.

6.3.3 Splice Statuses

The status of a splice Ss defines how complete the trace is considered to be by reporting if the entity specified in the Tp can be successfully found. Table 6.2 lists the splice statuses currently supported within the TORUS framework. It is expected that as TORUS evolves, the status types will remain a single set but will partition naturally into classifications more suited to a particular class of systems.

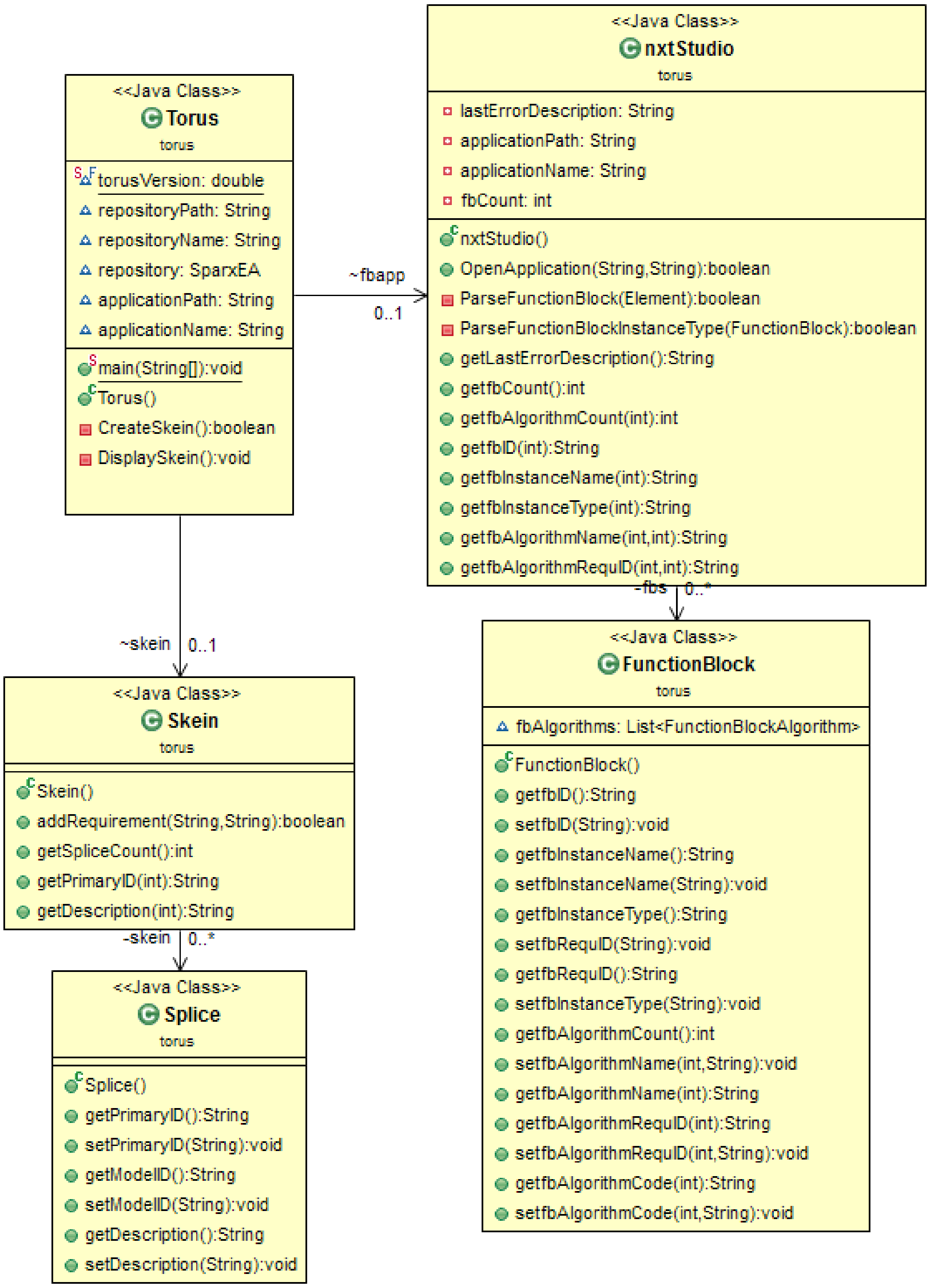
Table 6.2: Enumerated Types of TORUS Splice Trace Statuses

Trace Status	*Description of Trace Status*
UNFULFILLED	No application entities have been found that satisfy this requirement.
FOUND_REQUIREMENT	The link traces to the requirement in the Requirements Model that has been identified as being related to the other entities this splice is connected to.
USE_CASES_EXIST	Both a primary requirement and at least one Use Case for it has been located within the Requirements Model.
UNIT_TESTS_EXIST	Both a primary requirement and at least one Unit Test exists within the Requirements Model.
FUNCTION_BLOCK_EXISTS	One single Function Block or Composite Function Block has been located that matches the requirement in the Requirements Model.
MULTIPLE_FUNCTION_BLOCKS_EXIST	This requirement is fulfilled by more than one Function Block or Composite Function Block.

6.4 The TORUS Interface to nxtStudio

The second analysis pass scans the IEC 61499 application model created within nxtStudio. This is a multi-part XML-format file that holds a primary definition of all the objects created for the application. Where an artifact is an instance of a pre-defined class, the source for the parent object is stored in a separate XML-format file within the folder hierarchy.

A linked list holds a set of $FunctionBlock$ objects, with one entry for each function block found in the application source code. The $FunctionBlock$ class also maintains a linked list of all the algorithms within that function block. The list of $FunctionBlockAlgorithm$ objects can be iterated via the methods provided by the $FunctionBlock$ class that exposes properties for the number of algorithms found, their



<<Java Class>>
Torus
torus
torusVersion: double
repositoryPath: String
repositoryName: String
repository: SparxEA
applicationPath: String
applicationName: String
main(String[]):void
Torus()
CreateSkein():boolean
DisplaySkein():void
~fbapp
0..1
<<Java Class>>
nxtStudio
torus
lastErrorDescription: String
applicationPath: String
applicationName: String
fbCount: int
nxtStudio()
OpenApplication(String,String):boolean
ParseFunctionBlock(Element):boolean
ParseFunctionBlockInstanceType(FunctionBlock):boolean
getLastErrorDescription():String
getfbCount():int
getfbAlgorithmCount(int):int
getfbID(int):String
getfbInstanceName(int):String
getfbInstanceType(int):String
getfbAlgorithmName(int,int):String
getfbAlgorithmRequID(int,int):String
-fbs
0..*
~skein
0..1
<<Java Class>>
Skein
torus
Skein()
addRequirement(String,String):boolean
getSpliceCount():int
getPrimaryID(int):String
getDescription(int):String
-skein
0..*
<<Java Class>>
Splice
torus
Splice()
getPrimaryID():String
setPrimaryID(String):void
getModelID():String
setModelID(String):void
getDescription():String
setDescription(String):void
<<Java Class>>
FunctionBlock
torus
fbAlgorithms: List<FunctionBlockAlgorithm>
FunctionBlock()
getfbID():String
setfbID(String):void
getfbInstanceName():String
setfbInstanceName(String):void
getfbInstanceType():String
setfbRequID(String):void
getfbRequID():String
setfbInstanceType(String):void
getfbAlgorithmCount():int
setfbAlgorithmName(int,String):void
getfbAlgorithmName(int):String
getfbAlgorithmRequID(int):String
setfbAlgorithmRequID(int,String):void
getfbAlgorithmCode(int):String
setfbAlgorithmCode(int,String):void

Figure 6.6: The TORUS Class Hierarchy for nxtStudio.

names and any requirements information they may have been tagged with when they were created in nxtStudio. The code for each algorithm is also extracted from the application and stored within the $FunctionBlock$ object.

6.4.1 Abstracting the Function Block Information

Alternative IEC 61499 development systems such as FBDK (J. Christensen, 2016) and 4DIAC (Strasser et al., 2008) do not store their application information in exactly the same XML file structure. While the IEC 61499 standard (IEC, 2013) provides extensive information defining all the entities required, there are subtle differences in the way each development system persists its data. For this reason, in the same way the $SparxEA$ class abstracts the requirements presented in Enterprise Architect, so the nxtStudio class encapsulates a set of $FunctionBlock$ and $FunctionBlockAlgorithm$ classes to abstract nxtStudio-specific application information.

This ensures that the central TORUS classes are able to maintain a common, clearly-defined set of methods to retrieve and match information from both the requirements model and the application. Implementing a complementary 4DIAC, IsaGraf or FBDK helper class would use the same generic classes but process the application source according to its own particular Document Object Model.

6.5 Defining the Splice Metadata

The abstractions described so far have allowed a generic, system-agnostic splice structure to be defined that is expected to work equally well with other requirements management and development systems. Table 6.3 details the data attributes that can be stored in each splice.

Table 6.3: Splice Data Attributes

Attribute name	*Data Type*	*Description*
primaryID	String	The unique identifier for this Splice within TORUS. This is an incrementing number that is used as a pointer to this primary record by another object.
description	String	Textual description of the requirement this Splice is linked to. Extracted from the requirements management system. This is stored because if the parent requirement is later removed, TORUS will be able to determine what it was called the last time it was found. This is important for the history analysis.
modelID	String	Unique Requirements Model ID (often a GUID). Extracted from the requirements management system.
Link Table	Linked-List	A linked list of traces, one for each relationship that this splice is currently maintaining:
traceID	String	Unique identifier for the entry in the linked-list that this trace points to.
traceType	SpliceType	One of the enumerated splice types of TORUS Entities.
traceStatus	SpiceStatus	One of the enumerated types of splice trace statuses.

6.6 Examining TORUS

The latest version of TORUS including all source code is available from this link: $https://dl.dropboxusercontent.com/u/168752514/Torus_v2.zip$

6.7 Conclusions

This section detailed the creation of the first TORUS prototype and its architecture. The iterative design science methodology used provided a lot of much-needed flexibility to try out different splice and skein storage mechanisms. The checklist proposed in Table 3.1 was useful for evaluating progress day-by-day and retaining focus on what was important. Stopping and reflecting, iterating designs and running experimental data exposed a number of alternative ways of traversing the linkages. A number of potential constraints related to scalability were considered as well as examining some issues that might later impact the performance of the visualization strategies that are planned. These are discussed further in the analysis and future research sections of Chapter 8.

Chapter 7

Analyzing the Skein

The previous chapters detailed the refinement of the requirements set, the building of the workpiece color sorter and finally the creation of the TORUS prototypes. During the test runs of TORUS, the skein of splices was created as it probed the requirements model and the source code of our application.

This chapter explores what the skein tells us about the state of our requirements, our application, and how *complete* we should consider them to be. The status of each splice was determined partially by being able to create traceability pathways between the requirements and the code. However, that is only part of the answer. The formalisms and definitions created previously now allow us to perform mathematical operations on the splices to reveal deeper implications. By applying a more rigorous, formal treatment to the data they contain, we are able to extract metrics that help us to estimate the completeness of our applications. By understanding what the combined statuses of all the individual splices tells us, the scope and possible limitations of what the TORUS approach can deliver become clearer.

7.1 Analyzing the First Model and Application Code

Each TORUS prototype was tested against a range of requirement and function block model data sets that were created for the workpiece color sorter. Table 7.1 shows the first trace results:

Table 7.1: Traces Returned in Data Set 01

Requirement ID	*Splice ID*	*Traced to Function Block*	*Implied Splice Status*
FR2	SP00		Requirement is not fulfilled.
R1	SP01	FB2 COLOR_BASED_CONTROL of type Sorter	Requirement is fulfilled by a single function block FB2 traced using RequID attribute in the block Sorter.
FR1	SP02		Requirement is not fulfilled.
R2	SP03	FB2 COLOR_BASED_CONTROL of type Sorter	Requirement is fulfilled by a single function block FB2 traced using RequID attribute in the block Sorter.
SR1	SP04		Requirement is not fulfilled.
R10	SP05	FB3 SSC_DETECTOR of type ColorDetector.	Requirement is fulfilled by a single function block FB3 traced to ECC algorithms AG1 SAMPLE, AG2 RESET and unit test UNIT_TEST_02.
R3	SP06		Requirement is not fulfilled.
R4	SP07		Requirement is not fulfilled.
SR2	SP08		Requirement is not fulfilled.
R5	SP09		Requirement is not fulfilled.
R6	SP10		Requirement is not fulfilled.
R7	SP11		Requirement is not fulfilled.
R8	SP12	FB3 SSC_DETECTOR of type ColorDetector.	Requirement is fulfilled by a single function block FB3 traced to ECC algorithm AG2 SAMPLE.
R9	SP13	FB3 SSC_DETECTOR of type ColorDetector.	Requirement is fulfilled by a single function block FB3 traced to ECC algorithms AG1 SAMPLE and Unit Test UNIT_TEST_01.

The Enterprise Architect requirements model *Specifications_Model_02.xml* contains fourteen requirements. TORUS ran this data set against the nxtStudio application *Workpiece Color Sorter_v1* which contains four function blocks. This data set is referred to as *Data Set 01*. The code in the *Requirement ID* column is the unique identifier to the requirement assigned within Enterprise Architect. The splice identifiers in the *Splice ID* column were assigned automatically by TORUS during its first analysis phase which begins by probing the requirements model. TORUS always attempts to build the *forward traceability* pathways first since *backwards traceability* emerges as a natural consequence of reaching a function block. A deeper trace is established if a pathway can be found to an algorithm within a function block. The granularity of these traces was chosen to be the function block ECC algorithm level since using clues from the unit tests was expected to yield greater accuracy at that depth.

In a similar way, the nxtStudio IDE generates unique identification codes for each of the entities that it contains. Each function block is automatically assigned a system-generated function block number, such as FB2. The descriptive names of function blocks such as $COLOR\_BASED\_SORTER$ were chosen and assigned by the designers.

7.1.1 Analyzing Requirements that are Not Fulfilled

The simplest splices are those that capture requirements that are not yet fulfilled by any code found within the application. Often, these are early-stage requirements present in the model whose acceptance or unit tests have not yet been created. Figure 7.1 visualizes two splices for requirements FR1 and FR2 from *Data Set 01* that have been identified as being unfulfilled. The set of unfulfilled splices $SP_{unfulfilled}$ is defined as:

$$SP_{unfulfilled} = \{sp_0, sp_1, ...sp_n\} : SP_{unfulfilled} \in SP$$

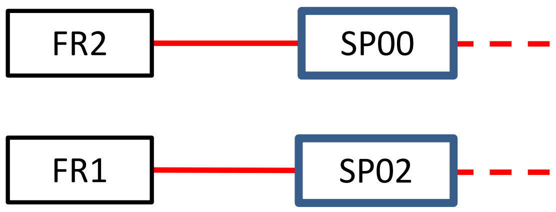


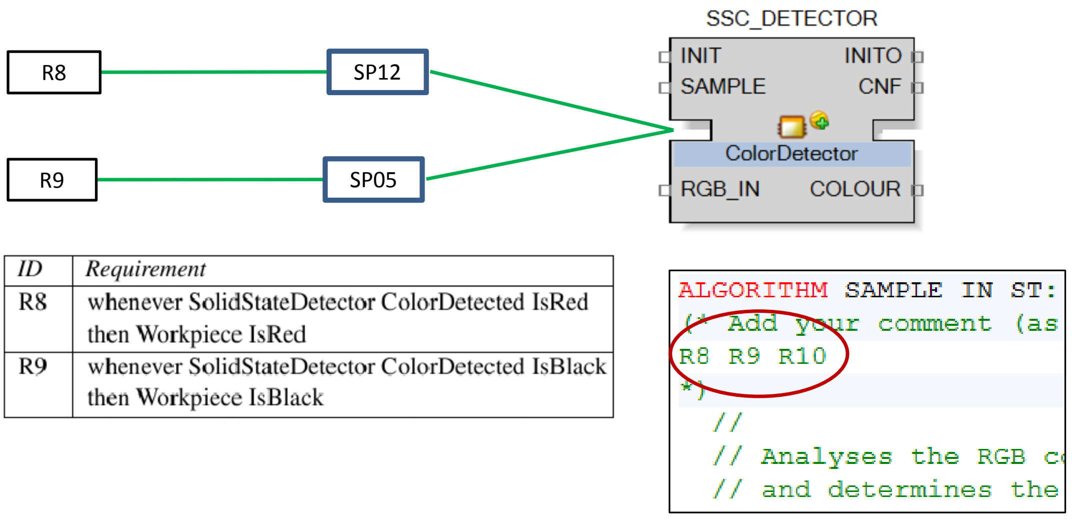
Figure 7.1: The Splices that Capture Unfulfilled Requirements.

where for each of the splices $sp_i \in SP_{unfulfilled} = \langle R', Tp', Ss', FB' \rangle \ni FB' = \varnothing$ since no function blocks have been created or found that implement this requirement. The overall splice status Ss' = UNFULFILLED, was determined by evaluating the set of all splice statuses contained within the list of traces. The complete list of enumerated types of TORUS entities and trace statuses were summarized previously in Tables 6.1 and 6.2.

7.1.2 Analyzing Requirements that are Correctly Fulfilled

Requirements R8 and R9 were correctly identified as being fulfilled by the function block $SSC\_DETECTOR$. TORUS was able to trace the appropriate function block algorithms since the correct ID codes for the requirements were manually added during development. Figure 7.2 shows the requirement IDs in the header section of the code for the function $SAMPLE$. In this visualization the green connecting threads indicate a trace that has connected to both the requirement and one or more application artifacts. The splice data structure contains a primary record that uniquely identifies the splice with a field called *spliceID*, assigned by TORUS. The current TORUS model holds information about the single requirement it is tracing that will be important later if the original requirement can no longer be located in the model. The historical aspects of TORUS splices are discussed further in Section 7.2.

Traces are stored as a series of linked records within each splice. Figure 7.3 shows the trace to requirement R8 as the first record. It is classified as being a link to an entity of type IS_REQUIREMENT. The status of this link is FOUND_REQUIREMENT. The

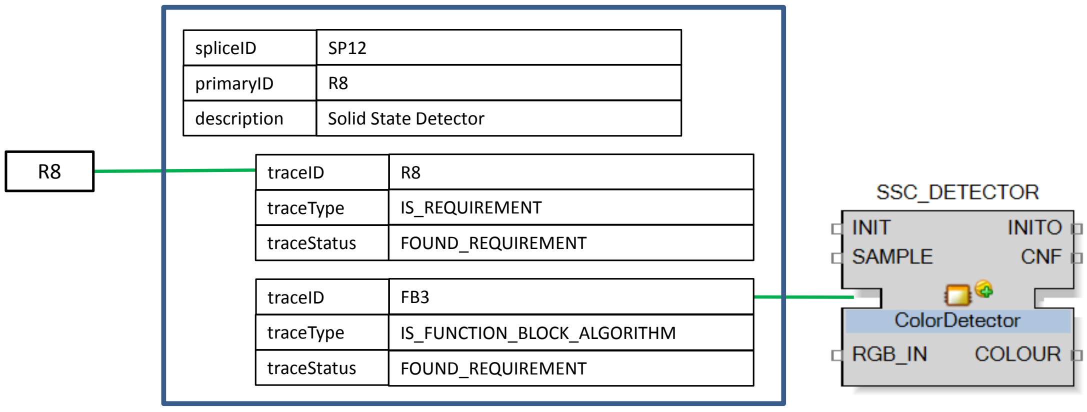


ID	Requirement
R8	whenever SolidStateDetector ColorDetected IsRed then Workpiece IsRed
R9	whenever SolidStateDetector ColorDetected IsBlack then Workpiece IsBlack

```
ALGORITHM SAMPLE IN ST:
(* Add your comment (as
R8 R9 R10
*)
  //
  // Analyses the RGB c
  // and determines the
```

Figure 7.2: Requirements Classified Correctly Using Function Block Algorithms.

second record contains a trace to the $SSC\_DETECTOR$ function block, classified as type IS_FUNCTION_BLOCK_ALGORITHM with status FOUND_REQUIREMENT.



spliceID	SP12
primaryID	R8
description	Solid State Detector

traceID	R8
traceType	IS_REQUIREMENT
traceStatus	FOUND_REQUIREMENT

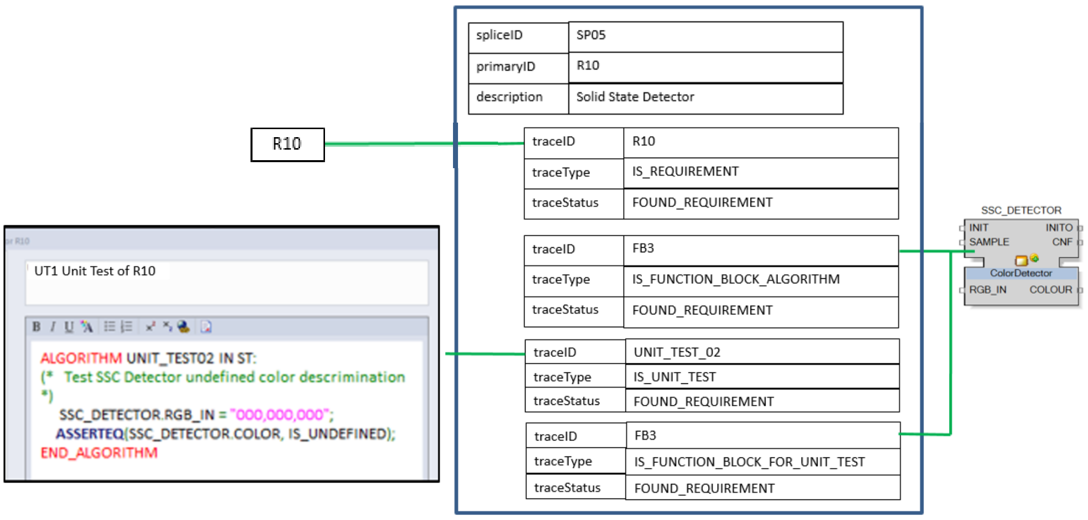
traceID	FB3
traceType	IS_FUNCTION_BLOCK_ALGORITHM
traceStatus	FOUND_REQUIREMENT

Figure 7.3: Linkage Information Recorded for the R8 Splice.

When analyzing the skein of an application, TORUS navigates between entities using these traces, always using the splice trace link tables as the starting point. The display and measurement functions also rely on this map since the *traceType* unambiguously identifies the generic type of the entity that is located at the end of a trace. The type also helps to determine which part of the data store holds the full entity record while the *traceID* provides the unique look-up key to the record in that list.

7.1.3 Analyzing Requirements using Unit Tests

Requirements R9 and R10 were traced using unit tests as well as information supplied by the developers in algorithms. The first two linkages shown in the trace link tables in Figure 7.4 are similar to those discussed earlier. They established linkages first to the primary requirement and then to the function block algorithm using tags in the code documentation. The third link was established by recognizing that there was a unit test stored in the requirements model under the hierarchy for requirement R10. The fourth link is the trace from the splice to the function block referenced in the unit test.



UT1 Unit Test of R10

```
ALGORITHM UNIT_TEST02 IN ST:
(*  Test SSC Detector undefined color descrimination
*)
   SSC_DETECTOR.RGB_IN = "000,000,000";
  ASSERTEQ(SSC_DETECTOR.COLOR, IS_UNDEFINED);
END_ALGORITHM
```

spliceID	SP05
primaryID	R10
description	Solid State Detector

traceID	R10
traceType	IS_REQUIREMENT
traceStatus	FOUND_REQUIREMENT

traceID	FB3
traceType	IS_FUNCTION_BLOCK_ALGORITHM
traceStatus	FOUND_REQUIREMENT

traceID	UNIT_TEST_02
traceType	IS_UNIT_TEST
traceStatus	FOUND_REQUIREMENT

traceID	FB3
traceType	IS_FUNCTION_BLOCK_FOR_UNIT_TEST
traceStatus	FOUND_REQUIREMENT

Figure 7.4: Linkage Information for the R10 Splice Established from the Unit Tests.

Using unit tests as traceability hooks has significant advantages over the use of tags such as the *RequID* and requirement references stored in the function block algorithms:

1. **Unit Tests have to correctly instantiate the objects they test.** The object references specified are therefore real entities that TORUS can expect to find in the application code. This implies that both *finding* and *failing to find* matches are

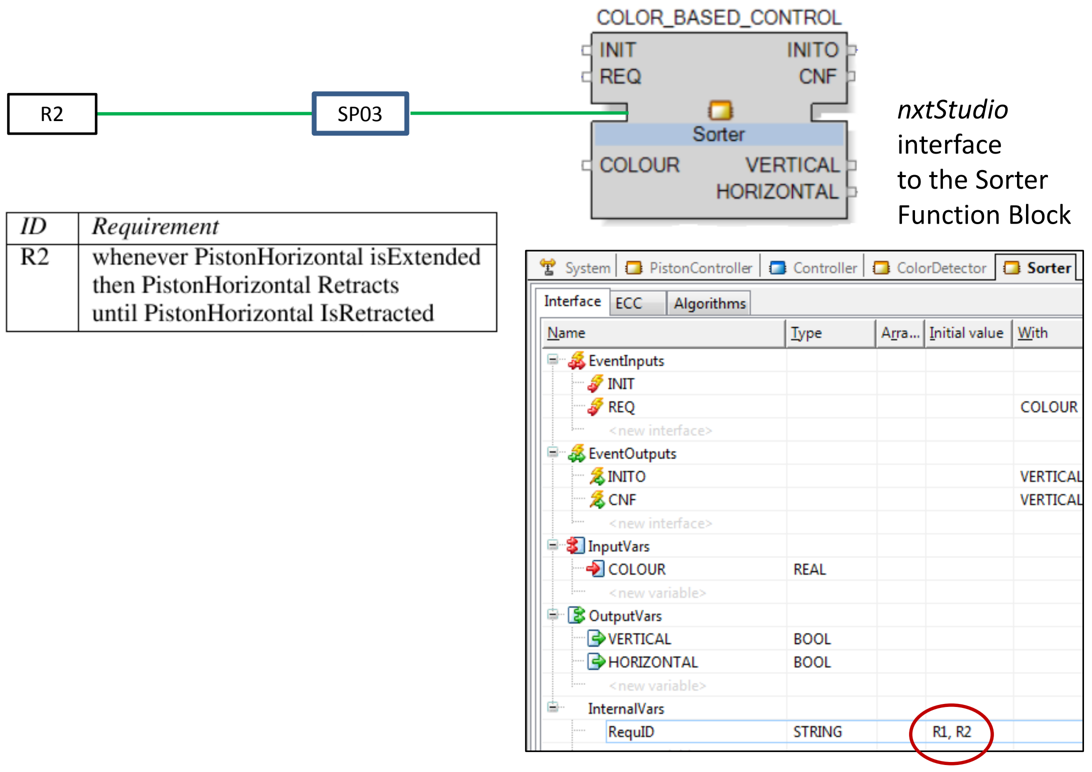
highly accurate indicators of how well the requirement has been fulfilled. TORUS uses a code-parsing algorithm that allows it to identify lists of object names that are referenced in each unit test. This implies that when a requirement is designed to be fulfilled by more than than one function block, TORUS can establish splice linkages to each and every function block that is cited in the unit test code.

2. **The object references are unique with the application.** Using requirement references in algorithms or by tagging interfaces in the way that the *RequID* attribute did only succeeded in marking the parent type of the object, not the object instance itself. In contrast, unit tests can reference specific and distinct objects by name. The implication of this is that requirements such as R1, R2, SR1 and FR1 can be associated directly with the object $H\_PISTON\_CONTROL$. In the same way, requirements R3, R4, SR2 and FR2 can be associated unambiguously with the instance of the same type $V\_PISTON\_CONTROL$.
3. **Unit tests are a well-established software engineering best-practice.** Since they are useful throughout the entire development cycle, they are significant contributers to the integrity and reliability of Model Based Systems Engineering and Test-Driven Design approaches. Since they have to run successfully to pass, unit tests have to capture the object names of real entities that they are testing. The traceability overhead of creating traces in this way is effectively zero.
4. **Unit Tests have to be refactored to keep in-sync with application code changes.** As such, the traceability maintenance overhead of this practice is also effectively zero.
5. **The number of trace linkages established in this way can be significantly large**. Once each of the matching entities, including the set of unit tests and each of the function blocks referenced has been traced, this section of the skein achieves a much finer degree of granularity. The implication of this is that there is a lot more information available to analyse when TORUS is reporting issues.

When an activity such as refactoring has broken something the unit tests, in conjunction with the splice history, can be used to quickly locate and classify such problems automatically. The implications of this for historical analysis is discussed further in Section 7.2.

7.1.4 Requirements that Were Not Classified Correctly

R2 is an example of a requirement that TORUS thinks it has traced correctly using evidence available from the models. However, Figure 7.5 shows that what the trace has really uncovered is an incorrectly-classified requirement:



ID	*Requirement*
R2	whenever PistonHorizontal isExtended then PistonHorizontal Retracts until PistonHorizontal IsRetracted

Name	Type	Arra...	Initial value	With
EventInputs				
INIT				
REQ				COLOUR
<new interface>				
EventOutputs				
INITO				VERTICAL
CNF				VERTICAL
<new interface>				
InputVars				
COLOUR	REAL			
<new variable>				
OutputVars				
VERTICAL	BOOL			
HORIZONTAL	BOOL			
<new variable>				
InternalVars				
RequID	STRING		R1, R2	

Figure 7.5: Requirement R2 that is Not Classified Correctly.

The TORUS trace has been shown in-context with the original requirements Use Case that relates to the movement of the horizontal piston. This is identified within the model as $PistonHorizontal$. TORUS established a trace from R2 by creating splice SP03 and then building a forward trace to the $COLOR\_BASED\_CONTROL$ function block. This function block was identified as fulfilling R2 because the function block interface attribute ***RequID*** contained a reference to this requirement, shown circled in red in the diagram. The interface also contains a reference to requirement R2.

Clearly, the $COLOR\_BASED\_CONTROL$ function block does not fulfill either of those requirements. Its role is to determine what the color of the workpiece is and make that data available to the piston controllers $H\_PISTON\_CONTROL$ and $V\_PISTON\_CONTROL$. It could be argued that this function block indirectly fulfills the requirements but that is a tenuous assumption. It is more likely that the attribute ***RequID*** was incorrectly coded during development. This class of TORUS result would be most useful during a requirements review. However, since the statuses of the TORUS splices are not reporting any issues, the problem would not trigger any warning alerts. If it had, then the engineers could re-classify the $COLOR\_BASED\_CONTROL$ as being responsible for fulfilling requirements R8, R9 and R10 instead.

This technique of using a tagged attribute such as ***RequID*** is fundamentally flawed. In the first instance, it has demonstrated that the practice of manually updating the requirement ID within the correct nxtStudio object is not reliable. The function block was probably re-factored a number of times during its design and it is easy to leave obsolete or incorrect free-form comment data within the source code. However, there is a deeper issue here. The block $COLOR\_BASED\_CONTROL$ is an instance of the prototype block $Sorter$. The Interface properties of this function block are editable but all instances, as is true of all object-orientated artifacts, share the same property values at design-time. While that would work well for single-instance objects, those that are based on the same instance type such as $H\_PISTON\_CONTROL$

and $V\_PISTON\_CONTROL$ would report the same value of their *RequID* attribute when TORUS probes them. This is the normal, expected behavior of objects so this technique for identifying function blocks has limited value. Note that this issue has also caused requirement R2 to be misclassified for the same reason. A similar issue could have arisen for requirements R8 and R9 discussed previously however, since this function block is instantiated only once, there is no ambiguity about which instance is being referred to.

A more promising approach would be to rationalize the naming of the object in the requirements model. Rather than referring to the horizontal piston as $PistonHorizontal$ it would be more consistent to name it $H\_PISTON\_CONTROL$. This would allow the text of the requirement to be analyzed formally in conjunction with the domain ontology. In this case, TORUS could have identified that the trace destination was to a function block that was not mentioned in the requirements statement. This is a similar approach to what was taken to use information from unit tests to create traceability links in Section 7.1.3.

7.1.5 Extending the Scope of the Analysis

To further demonstrate the potential of TORUS, two additional scenarios were introduced. The first examined the results of correcting some of the requirements errors present in *Data Set 01* and comparing the *Data Set 02* skein it generated to that of the previous historical *Data Set 01* skein. At the same time, additional errors were introduced by refactoring both the requirements model, the unit tests and the application function blocks that deliberately caused existing, fulfilled requirements to be broken.

The second scenario scaled the requirements of *Data Set 01* to iteratively increase the number of requirements in the model. The aim was to determine if the time required for TORUS to analyze increasingly larger requirements sets became unreasonably long.

If TORUS can scale comfortably to manage a model with 1,000,000 requirements in it such as the one outlined by Alford and Lawson (1979), then the methodology shows great promise.

7.2 Splice History within the Skein

Each splice persists data about the artifacts it is tracing. During a refresh after the requirements model is maintained or after development work on the function block application, historical splices are re-evaluated. Problem scenarios emerge that are similar but more complex than those discussed previously. For example, a splice that can no longer locate it's primary requirement indicates that a feature has been deprecated or re-written; what it points to is no longer recognized as being the original requirement. If the algorithms and unit tests linked to that splice still exist, they are recognized as orphans that are candidates for removal. If they could be successfully re-linked to a different or refactored requirement, the integrity of the splice would be re-established and its status would reflect that accordingly. *Data Set 02* was obtained after making changes to both the requirements model and the application. Table 7.2 shows only those traces that have changed.

Storing historical changes has similar advantages to that obtained from a source control system. When a change has occurred, the history of that artifact can often be used to help make sense of what has happened. Any piece of code, algorithm or ECC that cannot be traced back to a parent requirement is an orphan. In the case of code, this indicates a possibly unacceptable risk scenario. There is no rationalization for code that cannot justify its existence by referring back to the requirement that it fulfills. Within an algorithm that does link to a requirement, any piece of code that is not executed by a unit test is by definition unreachable or at least untested. Unreachable code often results from incorrectly implementing a requirement. There should also be code within each algorithm created by the ECC to respond to each of its state changes.

Table 7.2: Traces Changed in Data Set 02

Requirement ID	*Splice ID*	*Traced to Function Block*	*Implied Splice Status*
FR2	SP00	FB0 V_PISTON_CONTROL	Requirement is fulfilled by a single function block FB0. Traced via UNIT_TEST_03.
R1	SP01	FB1 H_PISTON_CONTROL	Requirement is fulfilled by a single function block FB1 Traced via UNIT_TEST_04.
R2	SP03	FB1 H_PISTON_CONTROL	Requirement is fulfilled by a single function block FB1 Traced via UNIT_TEST_05.
SR1	SP04		Original requirement can no longer be located.
R10	SP05	FB3 SSC_DETECTOR of type ColorDetector.	Requirement is not fulfilled. Function block can no longer be located.
SR12	SP14		Requirement is not fulfilled.

7.2.1 Analyzing Requirements That Have Changed

When new or corrected information is added to either the requirements model or the application, TORUS detects that the refreshed trace information does not match the historical skein from a previous probe. In *Data Set 01*, requirement FR2 was not fulfilled but the subsequent probe has established a new traceability pathway via UNIT_TEST_03 to the $V\_PISTON\_CONTROL$ function block. The first three new splices from the most recent probe correctly trace the requirement to its fulfillment. The original trace record, shown in the separate partition at the bottom of the list, illustrates how the splice has captured this change.

Subsequent redesign and development can also break requirements that were formerly fulfilled successfully. Requirement R10 was originally traced to the function block $SSC\_DETECTOR$ in the first analysis. By the time *Data Set 02* was probed, the function block could no longer be found, as shown in Figure 7.7.

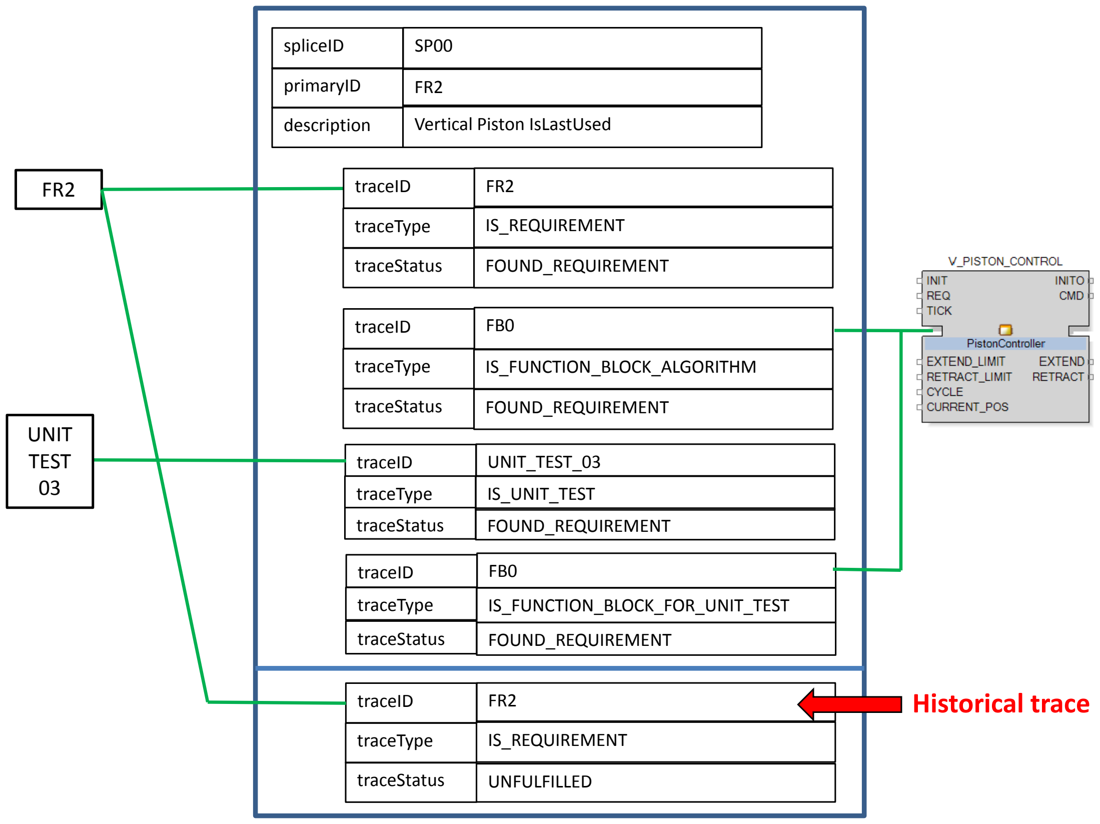


Figure 7.6: Current and Historical Traces for Requirement FR2.

In this splice, the first traces shown at the top of the list are new. They identify that UNIT_TEST_03 and the requirement R10 are still traceable in the current models. The historical traces shown in the lower box report that the requirement was successfully traced in a previous probe to a function block that is no longer able to be located within the current model. The red resultant overall status shown for the current probe indicates that TORUS has evaluated this hierarchy and determined that the current status of this requirement has now moved backwards to become UNFULFILLED.

Note that the current status of the system is always evaluated by using the most recent splices from the current probe. This is because the newest or the most up-to-date linkages are those established to the entities that are currently accessible in the models. All other historical traces are exactly that; they are deemed to be historical because they capture a system state that no longer exists. The value of historical splices lies in their

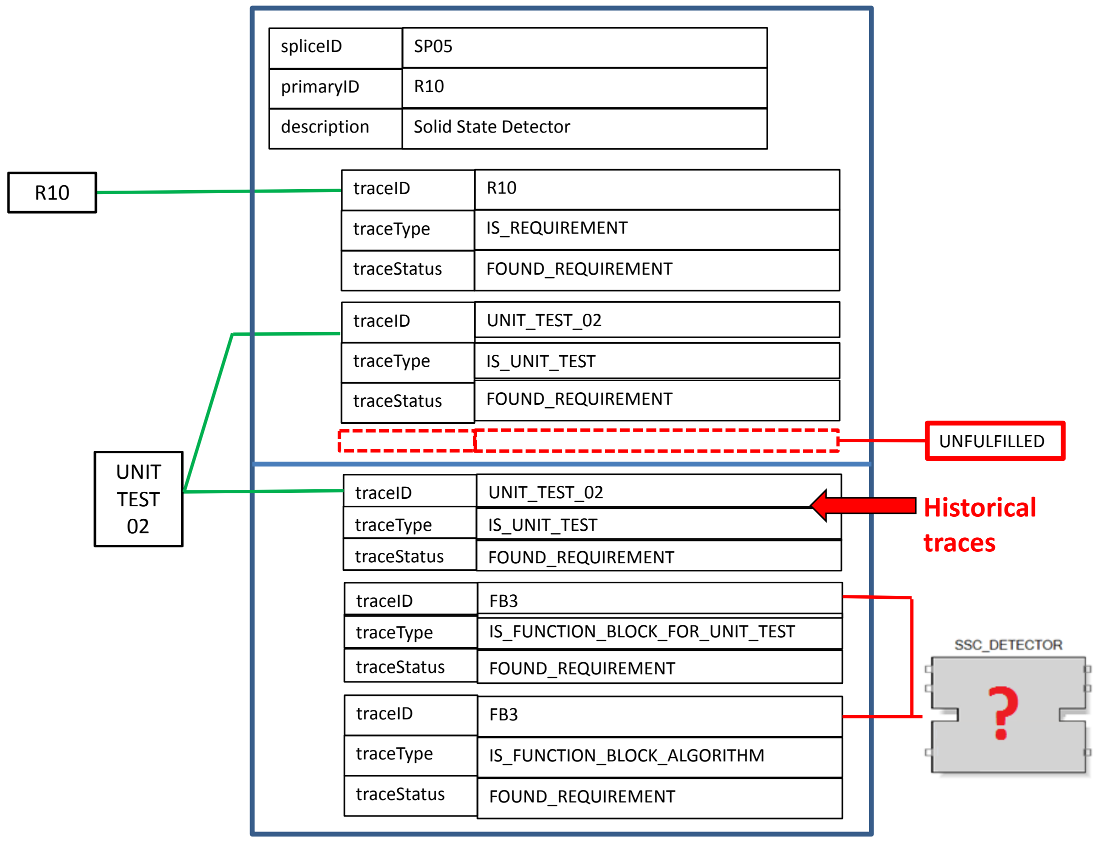


Figure 7.7: Analysis of the Current and Historical Linkages for Requirement FR2.

ability to provide ways of seeing what the system was like previously when trying to analyze current issues.

7.2.2 Detecting Orphans and Going Backwards

SR1 and SR12 illustrate a similar change. Requirement SR1 was originally unfulfilled but TORUS is now no longer able to locate it in the model. The designers possibly intended Requirement SR12 to supersede SR1 but since SR12 is still unfulfilled by either unit tests or locatable function blocks, they cannot be reconciled automatically to each other.

It is not reasonable for TORUS to just stop warning about orphan requirements after a certain number of probes have failed to locate them. The better approach is to use these alerts to deprecate requirements properly by setting their status appropriately within Enterprise Architect. Figure 4.7 in Section 4.3.1 shows the *Status* attribute for that requirement set to *Implemented*. Enterprise Architect also allows this attribute to be set to *Deprecated*. However, the correct process for managing orphan artifacts from these models automatically is not clear-cut and remains a topic for future research. It is coupled with the issues concerning how the skein should best be visualized and includes questions of how orphan code found within function blocks should be reported.

7.3 Operations on Splices

By defining the TORUS entities formally, it is then possible to analyze them using formal methods. One of the most important outcomes of this is that it allows us to implement *constraints* into the requirements models and the TORUS framework. Mathematical constraints allow TORUS to determine deeper aspects of how complete and consistent our applications currently are.

7.3.1 Detecting Singleton Constraints

A splice can trace either a single requirement or a group of aggregated requirements. Each requirement can in turn be implemented by either a single Basic or Composite function block or by a group of collaborating function blocks. Singleton Design Patterns are used in IEC 61499 systems where only a single function block or single composite function block is responsible for managing a particular task.

In our model, only the $V\_PISTON\_CONTROLLER$ is allowed to move the vertical piston. Other sub-systems can participate in the request to actuate the piston but it cannot be moved directly by any other sub-system. $H\_PISTON\_CONTROLLER$ is constrained in the same way and cannot fulfill any of the requirements R3, R4, SR2 and FR2. Figure 7.8 illustrates a situation where this constraint would be violated.

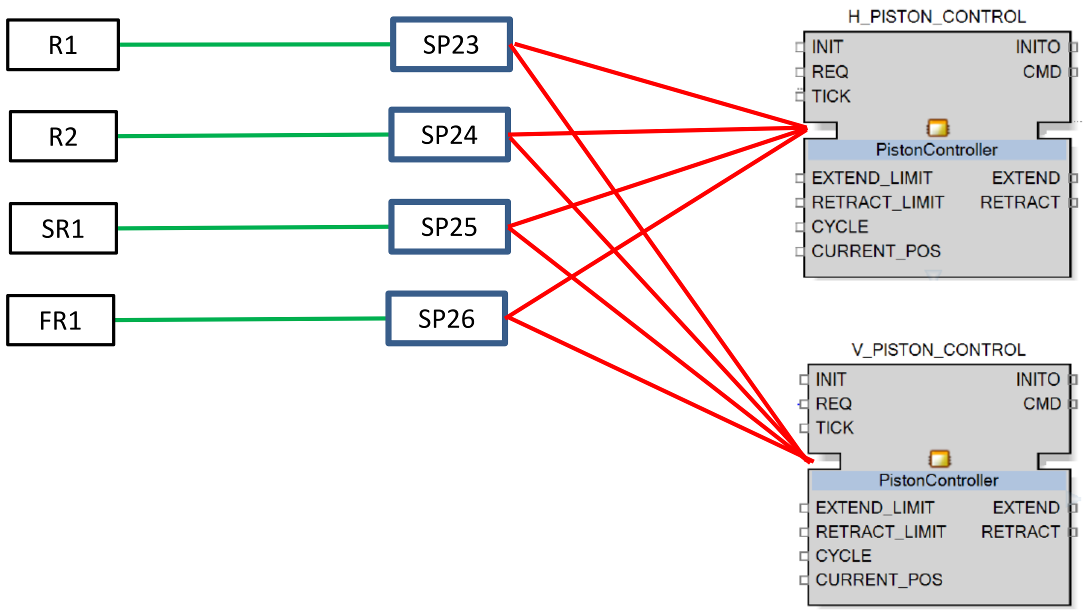


Figure 7.8: Violation of a Many-to-One constraint on a Singleton Function Block.

Definition 7.3.1 (Singleton) *A Singleton or One-to-One constraint exists when:*

$sp' = \langle R', Tp', Ss', FB' \rangle$ *where:*

1. $R' \in RM$ *and*

2. $Tp' \neq$ *IS_UNDEFINED and*

3. $Ss' \neq$ *UNFULFILLED and*

4. $\mid FB' \mid = 1$ *since only one* BFB *or one* CFB *may implement the requirement.*

7.3.2 Extending the Scope of Splice Types

We can also consider an extended class of splice types that refine the status to be *tested* or *untested*. A tested splice means its requirements have been tested on the system elements the splice traces to. Untested splices model the opposite condition when requirements are found to be traced to elements but not tested. A $superseded$ splice links a trace to a replacement trace, and can be useful when additional information such as unit tests are added for given requirements.

Splice types capturing relationships between requirements and system elements include:

- $satisfy$ when requirements must be satisfied by linked system entities,
- $constrain$ when requirements constrain linked entities such as in our Singleton example,
- $dissatisfy$ when requirements represent mis-use cases or undesired behaviors that cannot be satisfied by the entities that the requirement is traced to.

7.3.3 Arithmetic Operations on Splices

Arithmetic operations on splices allow us to combine and compare them as ways of exploring the relationships between system entities more fully. For example, the addition operation of two splices fuses their information:

Definition 7.3.2 (Splice Addition) *The addition of two splices $f_3 = f_1 + f_2$ is defined as follows:*

In the case of splices with the same types and statuses. if $f_1.Tp = f_2.Tp$ and $f_1.Ss = f_2.Ss$, we can fuse splices under further conditions, described as follows:

- *If $f_1.\mathtt{R} = f_2.\mathtt{R}$, then $f_3 = \langle f_1.\mathtt{R}, \langle \{f_1, f_2\}, f_1.Tp, f_1.Ss, Ds\rangle, f_1.\mathtt{SE} \cup f_2.\mathtt{SE}\rangle$. Here, $Ds = f_1.Ds \oplus f_2.Ds$ and $\oplus$ is the string concatenation operator. This case of addition depicts the condition in which the same requirements are linked to two different sets of system artefacts and we can aggregate them into a single splice.*
- *If $f_1.\mathtt{SE} = f_2.\mathtt{SE}$, then $f_3 = \langle f_1.\mathtt{R} \cup f_2.\mathtt{R}, \langle \{f_1, f_2\}, f_1.Tp, f_1.Ss, Ds\rangle, f_1.\mathtt{SE}\rangle$. Here, $Ds = f_1.Ds \oplus f_2.Ds$. This depicts the condition in which the same components*

must satisfy two different sets of requirements and we can aggregate them into a single splice.

Otherwise: $f_3 = \langle \varnothing, r2r, \langle \{f_1, f_2\}, untested, Ds \rangle, \varnothing \rangle$.
Here, $Ds =$ "$Addition:$" $+ f_1.Ds \oplus f_2.Ds$.

When two splices have the same metadata, except their free-text description fields, their addition results in an integrated splice containing all the requirements and system entities from both splices. Otherwise, the addition results in a new splice that supersedes the two previous splices. When considering large requirements sets that have undergone extensive changes, this operation can effectively compact the skein by removing duplicate splices which represented the same data in different ways. Without these types of constrained, formally-defined arithmetic operations, such optimizations would be extremely hard to identify manually or implement automatically via simple comparison algorithms. Subtraction can be defined in a similar way:

Definition 7.3.3 (Splice Subtraction) *If two splices f_1 and f_2 have the same types, statuses and link to the same set of system artifacts, their subtraction:*

$f_3 = f_1 - f_2$

retains these elements. The set of requirements:

$f_3.\mathtt{R}$ *is* $f_1.\mathtt{R} \backslash f_2.\mathtt{R}$ *and the* f_1 *and* f_2

are retained as the parents of f_3. Symmetrically, if f_1 and f_2 have the same types, statuses and link to the same set of requirements, their subtraction:

$f_3 = f_1 - f_2$

retains these elements, the set of system artifacts $f_3.\mathtt{SE}$ is $f_1.\mathtt{SE} \backslash f_2.\mathtt{SE}$ and the f_1 and f_2 are retained as the parents of f_3.

In all other cases, $f_3 = \langle \varnothing, r2r, \langle \{f_1, f_2\}, untested, Ds \rangle, \varnothing \rangle$. *Here,* $Ds =$ "$Subtraction:$" $+ f_1.Ds \oplus f_2.Ds$.

7.3.4 Applying TORUS Algorithms During System Testing

Splice operations are especially useful during the system testing phase. As testing progresses from the component level, through sub-systems and then to the system level requirements, splice statuses are changed to *tested* from *untested*. Often, it is possible to use the organization of splices to create a testing strategy where lower-level splices are tested first followed by higher-level requirements. Algorithm 1 traverses the system topology downwards to find if all splices linked to component c or one of its descendants that remains untested. For the workpiece color sorter case study, the top-level function block network cannot be considered tested until all splices related to $H\_PISTON\_CONTROL$ and $V\_PISTON\_CONTROL$) have been flagged as $tested$.

```
Input: Component $c \in SE$, topology $FBT$, all splices $SP$
Result: Splice set $SP' \subseteq SP$
1 $SP' = \varnothing$;
2 while $c \neq null$ do
3     $SP' = SP' \cup \{f \in SP \mid c \in f.\mathtt{SE} \wedge f.Ss = untested\}$;
4     $c = FBT.getChild(c)$;
5 end
6 return $SP'$;
```

Algorithm 1: Finding Untested Splices.

TORUS enables a number of additional operations and algorithms that are extremely useful during system development. We can traverse the splices to find missing acceptance tests for requirements or find orphan pieces of code that cannot be traced back to any requirement during the development phase. Historical linkages of splices allow us to backtrack and attempt to discover the rationales for those changes.

During development, we can further refine splices to link requirements to specific states or algorithms in the ECC of a basic function block or to connections in the networks of composite function blocks. Algorithm 2 shows how we can find all splices that apply to component c being developed, either directly, or indirectly through a parent component. We can further refine this algorithm to create separate sets of splices based on their types or statuses. We can also partition the requirements belonging to the set of splices returned by Algorithm 2 into functional requirements, non-functional requirements and unit tests. For example, the component $H\_PISTON\_CONTROL$ analyzed in the algorithm found splices linked to all the requirements shown in Table 4.2:

```
Input: Component $c \in SE$, topology $FBT$, all splices $SP$
Result: Splice set $SP' \subseteq SP$
1 $SP' = \emptyset$;
2 while $c \neq null$ do
3     $SP' = SP' \cup \{f \in SP \mid c \in f.\mathtt{SE}\}$;
4     $c = FBT.getParent(c)$;
5 end
6 return $SP'$;
```

Algorithm 2: Finding Direct or Inherited Splices.

7.4 Splice Metrics

Each splice can be thought of as carrying part of the truth within its metadata about the current state of the system. There will always be an inherent margin of error in what the splices have inferred about what they have gathered. That information can be aggregated and analyzed statistically in the same way that any data set that embodies known types of errors can.

7.4.1 Precision and Recall

When considering the data TORUS has retrieved from the models, we need to apply appropriate formal techniques to the data to create our analysis metrics.

Figure 7.9 presents a scenario that is typical of the tasks that cyber-physical systems are called upon to perform. If the Solid-State Camera from our workpiece color sorter was focused on this scene, its ability to discriminate between the orange cones and the red cones would depend on a number of environmental factors. The brightness of the lighting, cones that obscure other cones from view and shadows all play a part in how reliably the camera could see the cones.

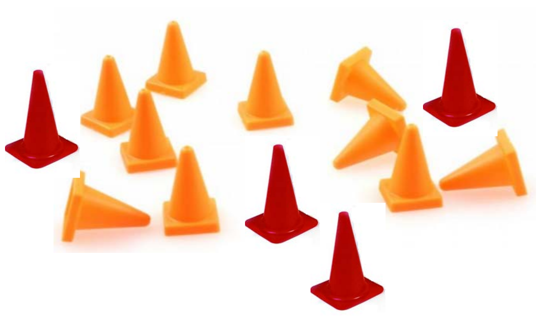


Figure 7.9: Identifying Cone Colors.

The *Precision* of a sample is defined as the fraction of retrieved instances that are accurately identified. A high-precision or *relevant* result is one that returns significantly more correctly-identified results than irrelevant ones. Suppose our detector returned a result that said there were seven orange cones found in a scene where there were actually ten orange cones present. Amongst that sample, by checking the spatial position of the cones the camera had detected, only four were found to really be orange cones. Three of those locations were found to contain red cones. The Precision of this sample is defined as:

$$Precision = \frac{\text{Number of Orange Cones really in the sample}}{\text{Size of the sample returned}} = \frac{4}{7} = 57\%$$

This means that of the seven cones recognized within the scene, only four were really orange.

In contrast, *Recall* is defined as the fraction of relevant instances that are retrieved. It can also be thought of as the *sensitivity* of the analysis:

$$Recall = \frac{\text{Number of Orange Cones identified in the sample}}{\text{Number of Orange code really there}} = \frac{4}{10} = 40\%$$

In our sample, the Solid-State Camera reported that it only detected seven cones in total and within that sample, only four were identified correctly or *recalled* out of a total field of ten that were actually there. A perfect precision score of 1.0 would imply that every result returned was relevant. In our case, that means that it would have correctly identified seven cones that were really orange in a sample size of seven. However, precision says nothing about how many additional orange cones there were in the entire scene that it has missed. In contrast, a perfect recall of 1.0 implies that all the orange cones were retrieved amongst a sample that also returned all the red cones.

7.4.2 Applying Recall and Precision within TORUS

We can apply a similar approach to calculate the recall and precision of the skein that TORUS has returned from our data sets. A perfectly-formed system would have all its requirements correctly traceable, by one or more correctly-applied techniques, to the application entities that are present. Figure 7.10 illustrates the Perfect Skein. In this example, each requirement has been correctly traced to the function block that implements it using just one unit test. The order of the requirements in this diagram more closely matches the layout of the original requirements model presented earlier in Figure 4.9. The splice $primaryID's$ shown still match those returned in the data sets. Every requirement has been found so both the precision and recall for this "Perfect" Skein is 1.0. This implies that in this system:

- All the requirements returned in the sample were traceable to an application artifact.
- While no requirements are reported as being unfulfilled, there may still be other requirements in the model that were missed because their attributes were not tagged appropriately. That is less likely since Enterprise Architect elements are unambiguous and unit tests are intrinsically linked to their parent requirements. TORUS reports when it has found part of a requirement that it determines it does not have information to classify correctly.

- High precision means that TORUS has returned mostly correctly-classified traces and few incorrectly classified ones. The precision would be lower if like the Solid-State Camera, TORUS returned traces that mis-connected requirements to artifacts.

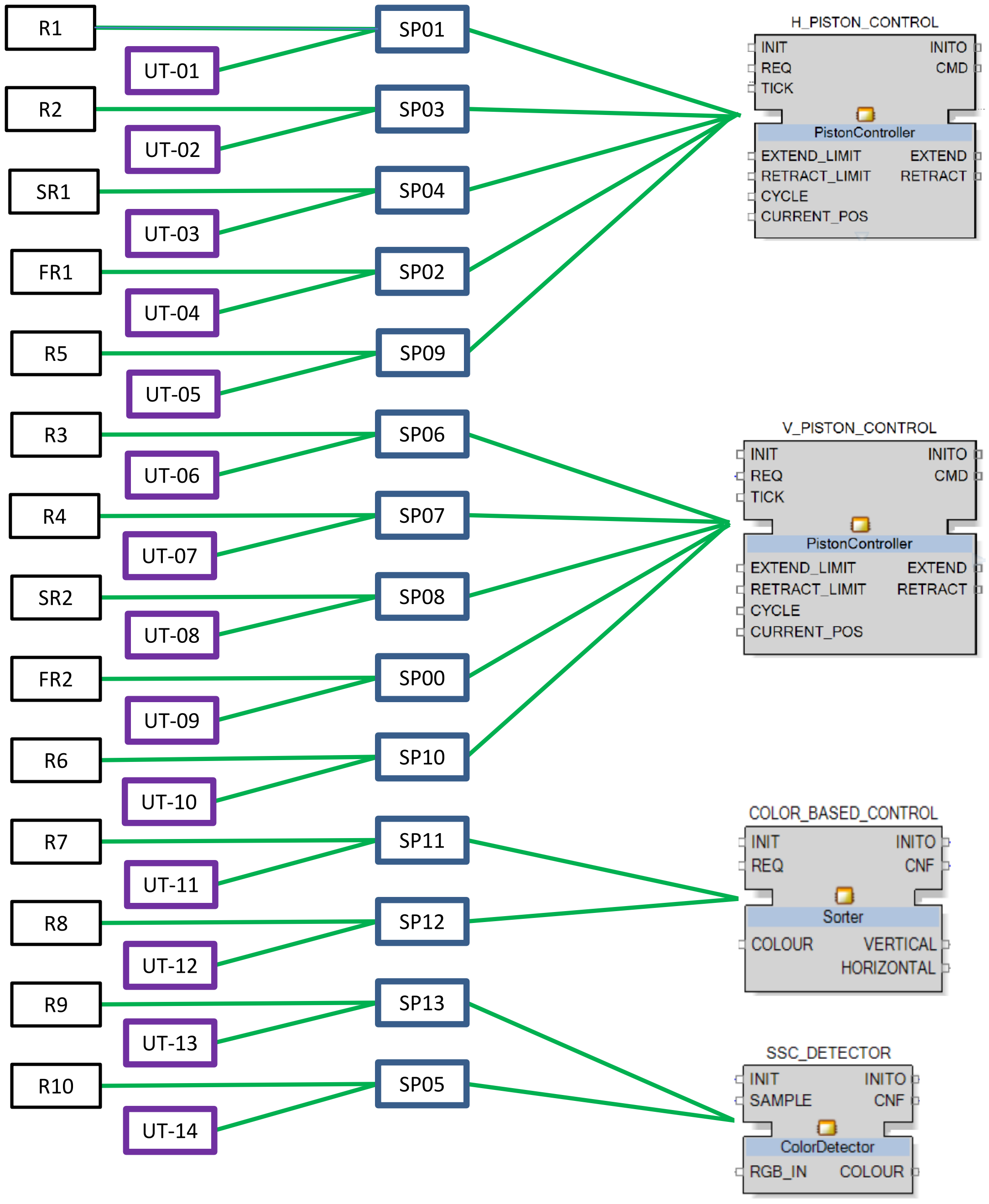


Figure 7.10: The Perfect Skein.

- High recall implies that TORUS returned a larger proportion of all the possible traces. The recall would be lower if TORUS failed to identify traces using the available information. In other words, it would fail to create splice linkages to elements that were there.

However, this ideal system is very different from that reported by the real skein for *Data Set 02*, shown in Figure 7.11. The green links again show that the traces reached a verified destination while the red links indicate that the requirements remain unfulfilled. The red dashed traces indicate that the splice was created in a previous probe and that the destination of the original link can no longer be found. In the case of SR1, the requirement no longer exists in the model and for R8 and R10, the function block can no longer be reached.

In this non-perfect scenario, precision and recall do not necessarily provide a more informative measure of completeness than they do for the perfect skein discussed previously. The precision is a measure of the *true positives* returned in the sample divided by the number of elements that belong to the class. This is a concept from Information Retrieval Theory as applied to traceability that defines recall as the measure of the number of traces correctly assigned to the correct target artifact (Cleland-Huang, Gotel & Zisman, 2012). Hence recall is a measure of the quality of the traces returned. In *Data Set 02*, all function blocks that can be traced back to a requirement that they belong to are reasonable. This is in contrast to the case in *Data Set 01* where requirement R2 was traced incorrectly to the $COLOR\_BASED\_CONTROL$. The recall for *Data Set 02* is therefore:

$$Precision = \frac{\text{Number of traces to valid artifacts}}{\text{Number of traces returned}} = \frac{9}{24} = 37\%$$

The recall is defined in terms of the traces that should actually be there. TORUS can only estimate this based on the number of requirements that exist and the number of function blocks that can be counted. If we define a minimum best-count in the same way as shown in the perfect skein but excluding the count of the unit tests, at least 30 trace links should have been established. This is based on 15 requirements being linked

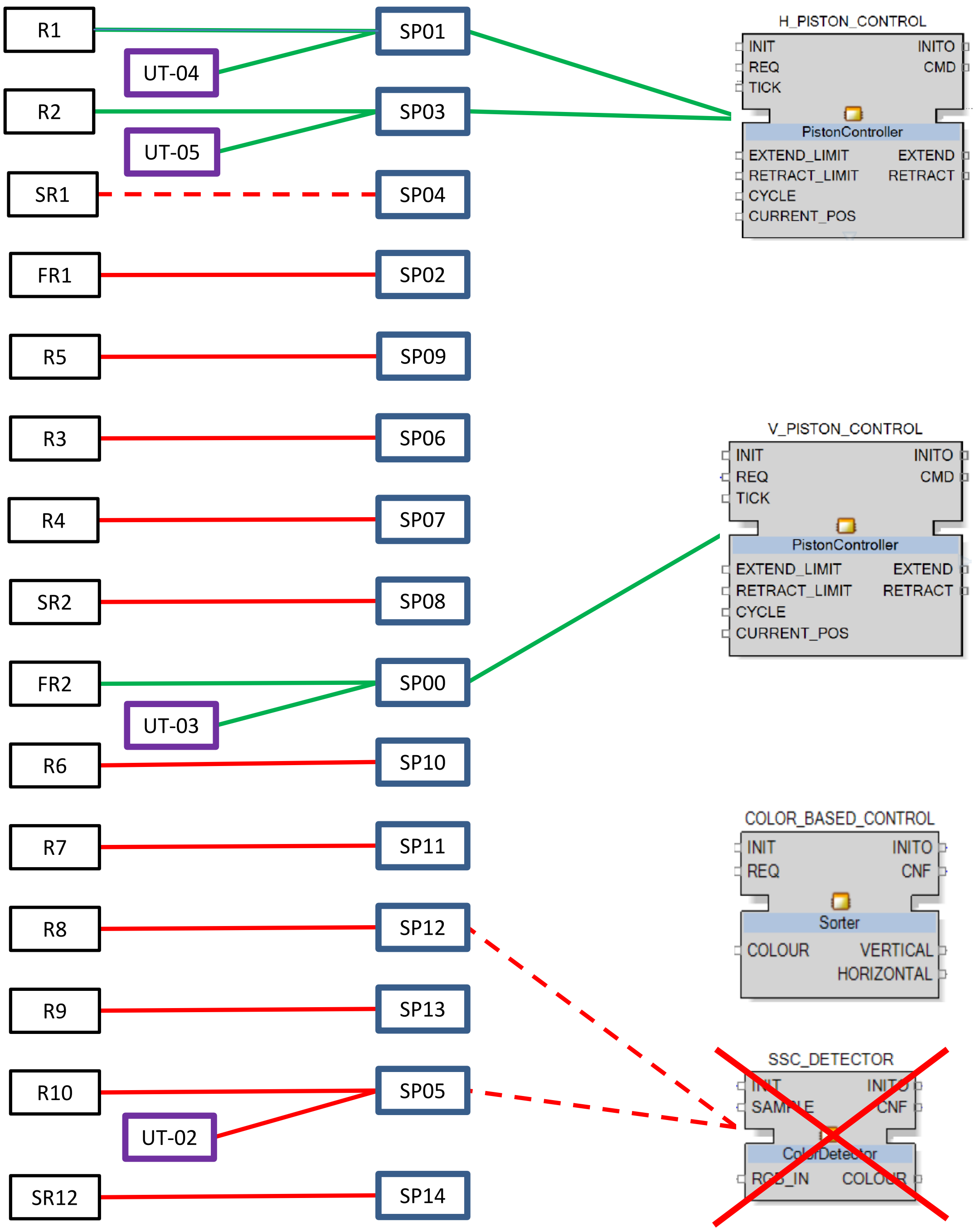


Figure 7.11: The Skein for *Data Set 02*.

through 15 splices to at least one artifact:

$$Recall = \frac{\text{Number of traces to valid artifacts}}{\text{Number of traces that should exist}} = \frac{9}{30} = 30\%$$

Egyed and Biffl (2005) discuss the use of precision and recall as ways of evaluating automatically generated traces. They measured the strength of each trace generated and calculated a ratio based on the number of code functions implementing a requirement and the number of functions that two requirements share. This was used to determine the threshold of what they defined as a weak or untrusted trace. These were deemed false-positives. Overall, they found that the weakest 10% of trace links contained only 1% of true traces.

7.4.3 Skein Contours as Alternatives to Precision and Recall

While precision and recall provide a measure of the quality of traces, they do not necessarily provide a useful measure of how complete the system is. Cleland-Huang and Heimdahl (2012) comment that precision and recall often exist in a state of tension. 100% recall can be achieved by capturing all possible links. However, this results in a low precision that does not really convey anything useful.

In the TORUS approach, the unit tests clearly provide a much higher level of confidence that the trace pathways to a function block or code artifact are accurate. The hierarchy established between the requirement and the unit test within the requirements model implies that when a match is found to a function block instance, then it is more likely that the trace has been established reliably. Given that, we can establish a new concept called the *Skein Contour*. The Skein can be visualized as having a terrain or landscape that is populated by artifacts from the requirements model, the unit tests, the TORUS splices, the function blocks and the code algorithms. How many of each type that can be traced to each other can be thought of as establishing *contours* that map the elevation or height of the landscape, as shown in Figure 7.12.

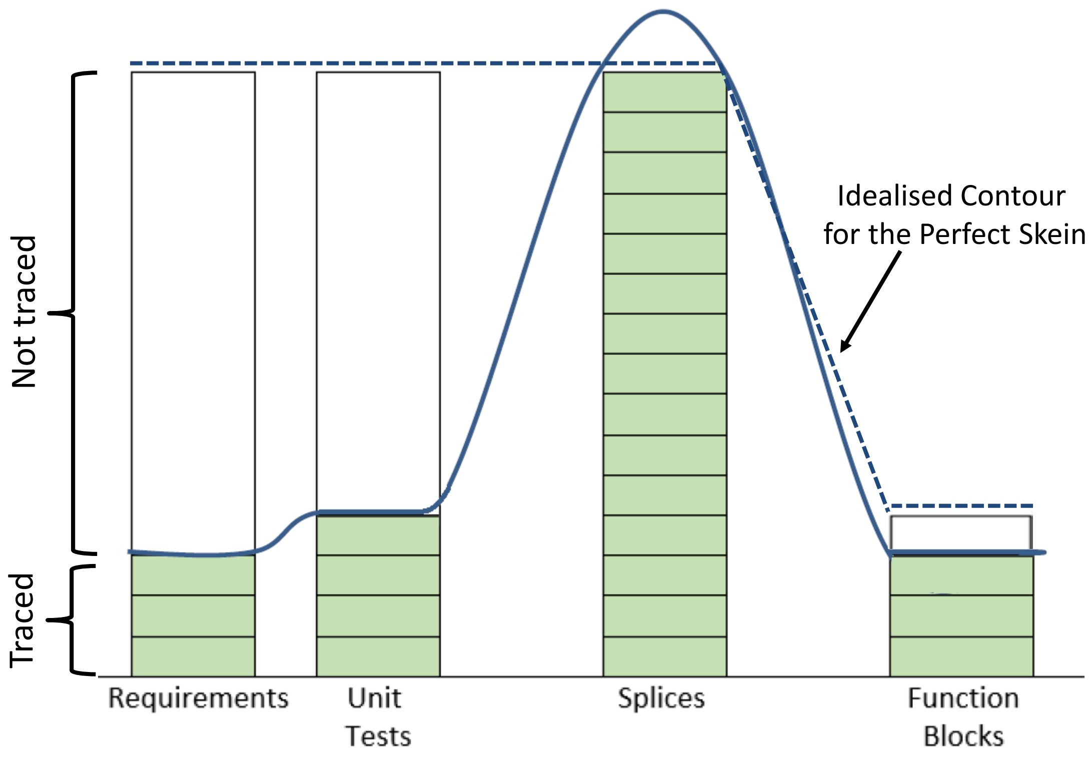


Figure 7.12: The Contour of the Skein for *Data Set 02*.

Each green block on the diagram represents an artifact that has been traced-to successfully. These highlight the parts of the application and requirements model that are more complete. A perfect skein, shown as the dotted blue line, would not have the bellcurve contour of the real skein shown here. Instead, the path would describe a level plane from the top of the requirements which would be at the same height as the splices column. We can evaluate the proportion of each completed artifact by adding them:

$$Contour = \frac{\frac{\text{Traced Requirements}}{\text{Total Requirements}} + \frac{\text{Traced Unit Tests}}{\text{Total Unit Tests}} + \frac{\text{Traced Function Blocks}}{\text{Total Function Blocks}}}{\text{Number of Categories}}$$

$$= \frac{\frac{3}{15} + \frac{4}{15} + \frac{3}{4}}{3} = 40\%$$

While for *Data Set 02* this figure is similar to that obtained for the precision and recall, it represents a totally different concept. The Skein contour has effectively added a weighting to each group of artifacts and then totaled their contribution. In this case, it means that it has been possible to trace 40% of the artifacts present in the system. If the number of ECC algorithms was factored in, the proportion of untraced orphan code could also help shape the skein contour.

We can also propose that the skein has a *weight*. Each splice can be thought of as having a density defined by the number of entities it traces to. Where a requirement has a large number of unit tests and the splice is able to reach successfully to all the function blocks and algorithms that fulfill those requirements, then that part of the system *weighs* more. The density of each splice is a measure of how complete that section of the system is. This has implications for the types of views of the skein that should be implemented to help teams visualize and understand the state of their system better.

7.5 Scaling TORUS

Section 7.1.5 proposed extending the scope of the analysis of the data sets to iteratively increase the number of requirements and function blocks in the models. TORUS is designed to be able to scale and work reliably on large, complex requirements sets and applications. How well do the different parts of TORUS perform under load? A three-phase approach was designed to look at how the different sections of the TORUS framework perform:

7.5.1 Analyzing a Large Requirements Model

The requirements model from *Data Set 02* was used to clone a new XMI-format model with one million unique requirements in it. As TORUS loaded it, the time taken to process and classify each section was measured to see how quickly TORUS could extract and store 1,000,000 distinct requirements.

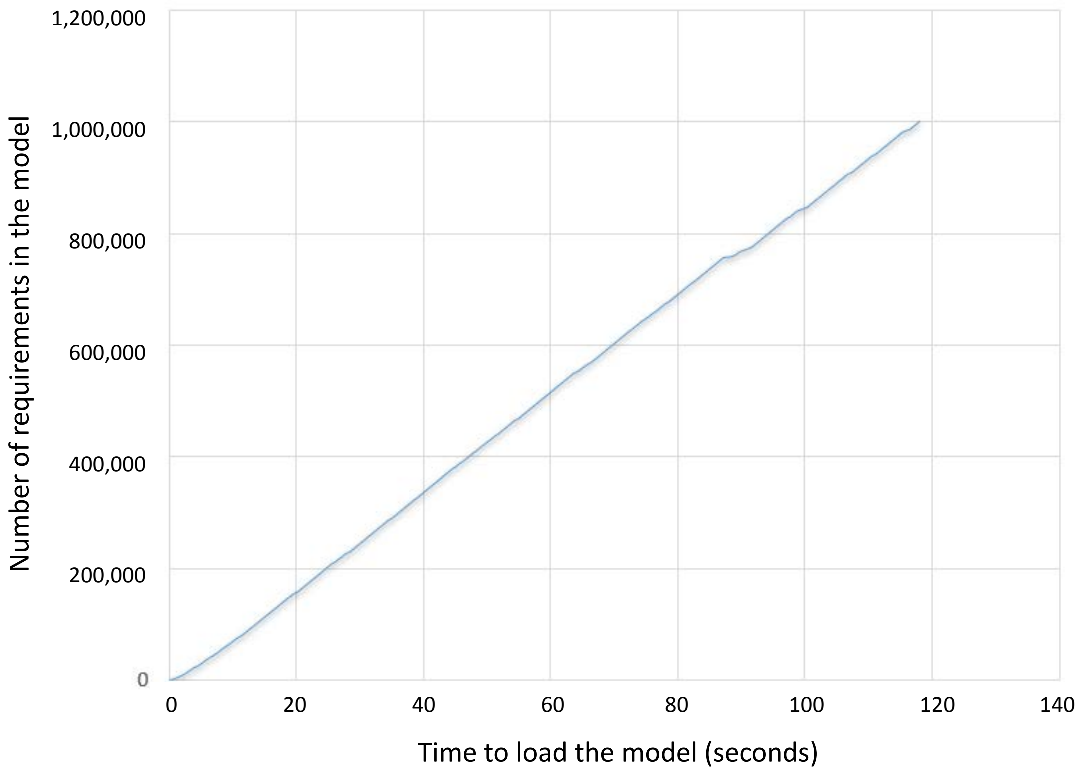


Figure 7.13: Time to Load 1,000,000 Requirements into TORUS

Each requirement was given a unique $primaryID$ during the creation of the test XMI-format file. Figure 7.13 shows that the load time, sampled after every block of 100 requirements was processed, is essentially linear. The performance of this phase is influenced by two main factors:

- The XMI is an XML-format, ASCII-sequential file. Accordingly, the reading time to traverse the entire requirement set in a forwards direction just once is linear with respect to the number of requirements.

- A Java $HashMap$ is used to store the linked list of $Requirement$ objects in the $SparxEA$ class. $HashMap$ indexing is inherently very efficient. Since the release of JDK version 8, local binary trees are used to file items with similar index keys, allowing them to bucket or *hash* together in optimal ways. Since TORUS is inserting each $Requirement$ object into the list in-sync with the reading of the

XMI-file, only one pass is required to populate the list. All operations are indexed insertions and additions rather than requiring a re-ordering to the structure after changes.

Hence this operation should show **O(n)** linear complexity and this is borne out by the results. They show a consistent linear increase in the time required to load progressively larger requirement sets.

7.5.2 Analyzing a Large Function Block Application

The loading and storage of the nxtStudio application code in the $nxtStudio$ class was studied in a similar way. Each application is stored as a single XML-format file. Access to secondary object type-definition files is made to retrieve and store its ECC algorithms when inspecting each function block instance.

As expected, the overall load time for the sample application containing one million function blocks is considerably longer than the time to one million requirements. The IEC 61499 nxtStudio file contains a large amount of IDE-specific data that is not related to the function block information TORUS is interested in. However, like the requirements implementation, the load time for the function blocks also demonstrates **O(n)** linear complexity.

In practical terms, it is unlikely that there would be one function block per requirement in a typical cyber-physical system application. The number of function blocks required is highly dependent on what the application does and how highly-optimized the design is. It is also important to remember that unlike requirements, function blocks are abstractions that will later generate differing amounts of object code. Depending on their complexity, the relationship between the eventual code line count and the number of function blocks is not a simple relationship. The workpiece color sorter demonstrates a 4:1 ratio of requirements to function blocks that may or may not be typical of other applications. What is hoped is that the more detailed and refined the requirements are, the higher the quality of the application should be.

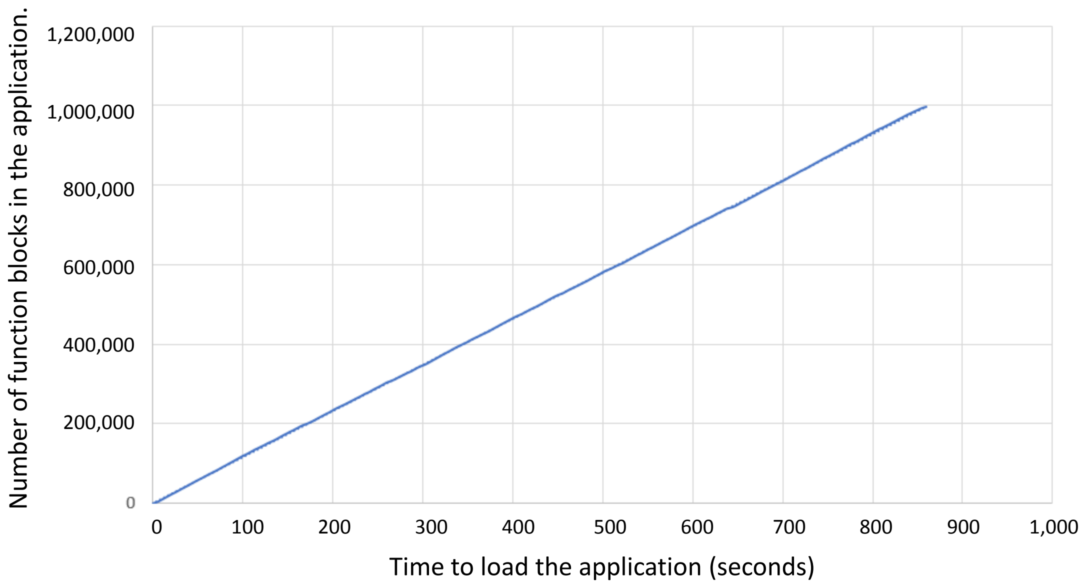


Figure 7.14: Time to Load 1,000,000 Function Blocks into TORUS

7.5.3 Analyzing the Creation of the Skein

The final phase of any TORUS probe is the creation or refreshing of the Skein. TORUS always generates one splice for each requirement with one primary trace linkage to it. This results in a minimal 1:1 ratio of requirements and splices in the current TORUS implementation. Then the function block application is probed via the $nxtStudio$ repository of function blocks, building up traces within the splice that reach out to each application entity based on each trace relationship that has been uncovered.

Our cloned requirements and function block models are of necessity randomized. The 4:1 ratio of requirements to function blocks ratio exhibited in the workpiece color sorter was chosen as an appropriate ratio. Each requirement and function block was given a unique $primaryID$ to ensure that in our proposed example, for one million unique requirements there will be 250,000 possible, unique function blocks they

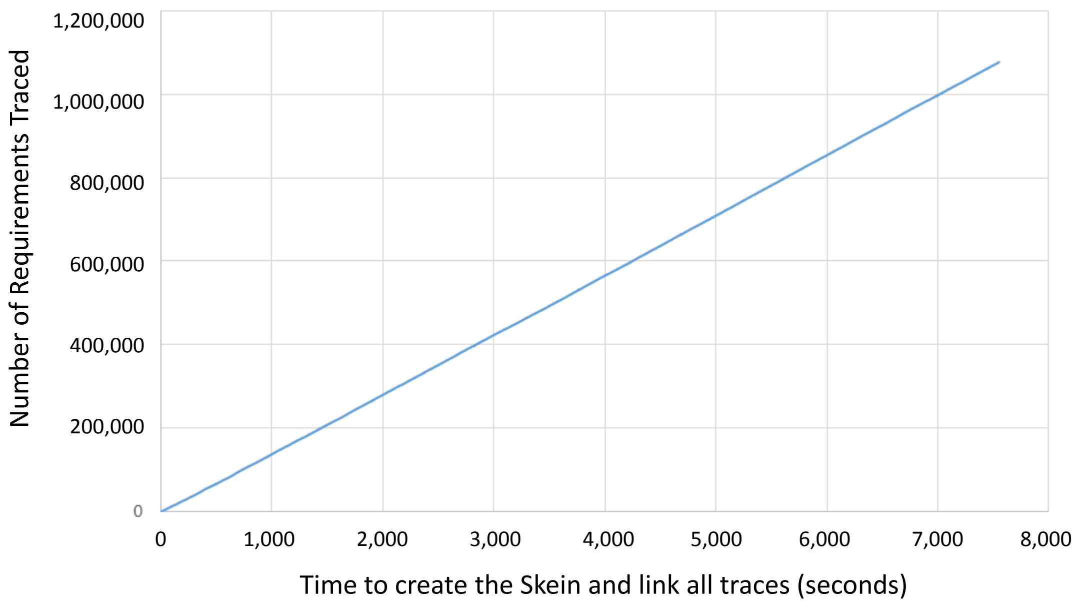


Figure 7.15: Time to Trace 1,000,000 Requirements and 250,000 Function Blocks.

could trace to. To create traceability hooks, each splice was first linked to its parent requirement. A randomized number of n function blocks were then traced to by first extracting n function blocks index keys from the available pool. This yielded a list of function block instance names just as if they had been identified through the analysis of the requirement's own unit tests. Each of these function blocks were then retrieved from a list using the normal function block lookup methods available in the $nxtStudio$ class. This ensured that we were testing the proper methods TORUS would use in a real scenario. For practical purposes, the maximum number of traces that any single splice would establish was limited to six.

Figure 7.15 shows that the skein creation also demonstrates **(O(n))** complexity. This is expected since the creation of each trace requires the function block information to be retrieved from a $HashMap$ that has unique indexes. The traces themselves are created with composite index keys so their later retrieval times will also be linear. The complete set of tasks to create the skein are therefore a series of steps, each one demonstrating linear complexity:

```
Input: Requirements Model $RM$, Function Blocks $FB$;
Result: Skein $SK$ consisting of all $SP$;
1 foreach $R \in RM$ do
2     Create a splice $sp \in SP$ linked to $R$ ;
3     foreach $fb \in FB$ referenced by $R$ do
4         $sp.SE = sp.SE \cup \{fb\}$;
5     end
6 end
```

Algorithm 3: Creating the Skein.

7.5.4 Practical Time Considerations

For most practical situations, a requirements set of one million items will be rare. During our most extreme trial, TORUS took 125 minutes to create one million splices and establish 3,232,017 traces to a pool of 250,000 possible function blocks. With a smaller model consisting of 100,000 requirements, TORUS established 1,117,227 traces to a pool of 10,000 function blocks in 6.7 minutes. This suggests that even with the current prototype, TORUS demonstrates acceptable performance for medium to large requirements models and applications.

7.6 Strengths, Weaknesses and Limitations

Is this workpiece color sorter a representative example of a typical cyber-physical system? *Data Set 01* only contains thirteen requirements but these emerged iteratively during the elucidation performed on the initial pre-RS statements first presented in Section 4.1. Is the scale and complexity of this example significant enough to be representative of real-world cyber-physical systems? Given what TORUS has been able to trace within this requirements model and the resultant application, we would argue that it is representative for the following reasons:

New requirements emerged that were not present in the pre-RS. Refinement of this apparently trivial example and modeling it within Enterprise Architect demonstrated the need to create extra safety and functional requirements to ensure that the application would operate in a more consistent and safe manner. Hence, what may at first appear to be a trivial requirement specification quickly becomes much more complex. The literature review has already examined safety-critical situations that clearly did not address the risks they embodied adequately (Ho, 2015; Docrill, 2015).

TORUS presented a representative range of trace statuses. This simple model produced the full range of splice types possible at this stage of TORUS's evolution, including multiple traces from a single requirement to application entities. The historical analyses that follow will demonstrate how requirements such as R10, which initially produced five traces, increase their complexity and value as they effectively capture the changes that occur over time.

Unit Tests have been shown to be very significant. Scale and complexity have been profiled previously as barriers to the adoption of traceability. In this example, it was demonstrated that unit tests required little or no manual effort to create traces from. Further, they retain their integrity during re-factoring. This reinforces the value of Model-Based Systems Engineering and Test-Driven Design as methodologies that actively contribute to the quality of traces within frameworks such as TORUS without significantly raising the trace creation and maintenance overheads.

The example chosen to model incorporates many of the most important traceability scenarios encountered in cyber-physical system design. Coupled with the results of the scalability testing, TORUS shows great promise. The limitations that are now apparent relate mostly to how cleverly and accurately TORUS can mine code in the application. While the Enterprise Architect requirements model created has been shown to be hierarchal with tightly-linked secondary artifacts such as unit tests, the application code model is far less heterogeneous. TORUS has been shown to work well with a well-structured requirements model but it is clear that its performance would degrade significantly when asked to mine models of lower quality. To address that, the formal methods profiled already suggest that they would be a viable route to create more innovative capabilities. The next section examines the future directions that the analysis presented here suggests.

Chapter 8

Conclusions and Further Research

The original question that motivated and drove this research was: *"How do we use formal methods to facilitate the traceability of large, complex requirement sets for safety-critical Cyber-Physical Systems?"* During all of the time spent thinking, reading and experimenting, it has clearly been important to stay cognizant of the need to keep any proposed solution grounded in the real-world.

Therefore it felt right to present each of the conclusions in this final section side-by-side with the implications for the future research directions that they point towards. While this first phase of the research has proposed the TORUS framework as a novel and promising way of facilitating traceability, traceability alone is not enough of an answer. The immediate goal of the research has been to demonstrate that automating trace creation and maintenance through this formal framework is both feasible and delivers immediate benefits. However, without rich-enough models for requirements and robust cyber-physical system architectures to tether those traces to, the long-term goal of being able to accurately gauge the *completeness* of an application cannot be met. The research challenge was always to provide a compelling enough motivation for adopting Model-Based Systems Engineering practices that outweigh the cost and time required to build rich-enough models.

Future versions of TORUS could arguably get a little nearer to that goal by incorporating validation and verification methods into the TORUS framework. Coupled with delivering sound metrics such as those presented earlier, we are also coming closer to understanding how development teams can comprehend and manage large, complex system requirements. Being confident that you know where you are along a project's development path is knowledge beyond price. Tools and frameworks that support such a voyage are like the instruments of a navigator; in skilled hands, they help make the passage safer. However, wherever people, software and great expectations are involved, the maps all too often say *"Here there be dragons..."*

Figure 8.1: *"Here there be Dragons..."* (Zell-Ravenheart, Oberon, 2016)

8.1 Abstraction Reconsidered

Earlier in this research, a requirements model was created manually using CESAR boilerplates. Sparx Enterprise Architect was then able to visualize it as a set of SysML Requirements Diagrams. Creating the model in Enterprise Architect was painless due to the extensive tutorial support available from Sparx as well as the ease-of-use of the application. The IDE provides ways of quickly extracting high-quality Acrobat Portable Data Format (PDF) images of the model views which were used extensively within Chapter 4.

After completing the Function Block application in nxtStudio, the nxtStudio HMI Visual Interface Environment provided a way of creating an animated view of the pistons moving under the control of the application. The end result was a functioning prototype that was later used when exercising the TORUS application.

However after getting this far, a deep time of reflection and doodling on white boards ensued. Three primary questions emerge from the diagrams shown here in Figure 8.2.

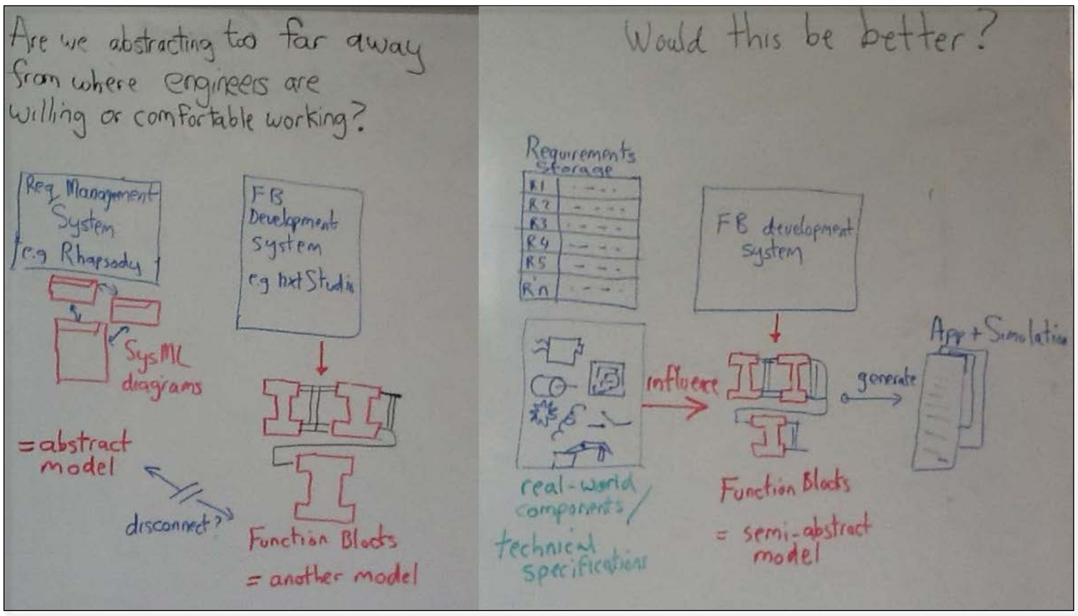

Are we abstracting too far away from where engineers are willing or comfortable working?
Req. Management System e.g Rhapsody
SysML diagrams
= abstract model
disconnect?
FB Development System e.g nxtStudio
Function Blocks = another model
Would this be better?
Requirements Storage
R1
R2
R3
R4
R5
Rn
FB development system
influence
generate
App + Simulation
real-world components / technical specifications
Function Blocks = semi-abstract model

Figure 8.2: Are we Abstracting Too Far Away From Where Engineers Wish to Work?

1. In what ways did the effort expended to create the SysML model affect the design choices made in nxtStudio as function blocks were prototyped, connected to each other and algorithms were designed and coded?
2. Did the effort expended in Enterprise Architect produce anything more than a connected list of interrelated requirements, use cases and acceptance tests?
3. More importantly is the IEC 61499 abstraction, presented so powerfully in nxtStudio, a better, richer and more useful requirements model that the SysML model created earlier?

If we are to encourage practitioners to adopt Model-Based Systems Engineering approaches, we have to be able to justify the cost and effort that will go into creating models by facilitating excellent traceability, validation and verification. The Function Block abstraction does not imply a direct one-to-one code-level mapping between algorithms in the compiled application and their representation within the IEC 61499 compositional architectural model. What it has done is provide an excellent way of modeling functionality that represents concepts such as data, input and output channels, in the context of their role in fulfilling requirements. The nxtStudio model has effectively allowed us to build an *Executable Specification*. We were able to use it to simulate our solution without having to deploy it on specialized hardware. We were also able to build the functionality iteratively, exploring technical problems as we refined the design.

This approach is very close to the way engineers work within real projects, prototyping parts of the solution to address open questions about the design choices that need to be made. Performing this sort of iterative modeling and design directly in nxtStudio has the advantage that the model is continually being refined in a much more tangible way. In contrast, the model created in Enterprise Architect would need to be updated separately to keep it in-step with the nxtStudio model. While SysML Block, State Machine and Sequence diagrams can be used to model function blocks, are they abstracted too far away from the application compared to what nxtStudio models capture and present?

Once the development work was under-way in nxtStudio, the SysML diagrams were not used extensively. Rather, the CESAR requirement statements became a checklist of specified functionality. In this respect, an Excel spreadsheet could just have easily been used instead of Enterprise Architect. Clearly modeling solutions have a place; IBM Rational Rhapsody has extensive code-generation capabilities and built-in emulators, supporting Model Based Systems Engineering and Test-Driven Design. However, by the time the function blocks were being created, details of the various inputs, outputs and algorithms not captured in the SysML diagrams became far more important design questions. Modeling directly in nxtStudio is an attractive and viable alternative. The prototypes produced in Enterprise Architect and Rational Rhapsody can be made into very functional simulations. The issue remains to be answered if what they produce currently is still too abstract for engineers to use effectively. This research suggests that they are and that nxtStudio and TORUS together with a simple requirements management tool is an ideal tool chain to support Model Based Systems Engineering and Test-Driven Design.

8.2 Using Splices During Modeling and Development

We observed that the most effort in understanding what the trace model splices presented was localized within the initial requirements engineering and design phases. There, requirements were refined many times. On the other hand, the most value from splices was reaped in the development and testing phases as a means to see how many requirements were covered and to identify untested requirements. Hence, the automation of the splice creation focused on the initial requirements engineering and design phase. Visualization support in the later phases is highly desirable.

The exponential increase in requirement numbers and consequently splice numbers was not unmanageable. While the tool scaled easily to the various variants of the workpiece colour sorter, all of whom have a high degree of reuse of the piston controller function block, there is a need to test it more widely on systems with different degrees of reuse of blocks and higher heterogeneity.

8.3 Unit Tests as Facilitators of Traceability

Unit Tests emerged as a powerful tool to facilitate traceability. Their ability to allow traces to be created automatically and remain up-to-date with little effort is a significant contributor to the success of the TORUS approach. While nxtStudio does not support unit testing as a built-in part of its environment, other IEC 61499 development systems such as 4DIAC and IsaGraf do.

JUnit-style unit tests work by instantiating objects and exercising their properties. This allows single function blocks to be tested but the paradigm does not work when trying to test requirements deployed across heterogeneous distributed applications. Developing a unit test framework that allowed timing constraints to be exercised across distributed IEC 61499 sub-applications would be a worthwhile future project. Integrating that with subsequent versions of TORUS would build upon the work already done and explore additional ways of creating richer trace pathways, delving deep into function block algorithms.

8.4 Validation, Verification and Formal Methods

The term "completeness" has been used extensively in this thesis however the techniques required to determine this characteristic of the system are complex. The scope of the present work has been constrained to show that requirements that can be traced to artifacts are in some way more *complete* than those that cannot. A fuller evaluation of a system should encompass aspects of both validation and verification. To do that requires both analysis of the way the code is structured and concepts such as coupling and cohesion. Since IEC 61499 function blocks are truly object-orientated, they yield well to such measures of code quality. Formal methods were presented as ways of evaluating traces and exposing deeper aspects of the information they provide. Future research should include allowing TORUS to mine what is out at the edges of those traces to search for orphan and unreachable code. This would include reaching beyond the end of the trace to see if what it is connected to does indeed fulfill the requirements it is bound to.

8.5 Visualizing the Skein and its Contours

Visualizing large requirement sets is an extremely hard task. Traditional paper-based specifications cannot capture the very dynamic nature of development. The literature section discussed keeping trace history up-to-date as a barrier to the adoption of traceability tools and the use of Model Based Systems Engineering.

Throughout this work, the splice has been presented as a thread that connects simple boxes containing requirements on the left and function blocks on the right. This simple representation has proven to be highly-effective during discussions with colleagues about this work. The rational behind the terminology used to describe a splice's characteristics became very easy to explain. The simplistic green and red trace statuses were perceived to be both evocative and simple to understand. The number of traces that emanate from a splice conveys a rich message about how well-fulfilled or complete the requirement might be.

The purpose of such visualization is to allow teams to comprehend and discuss their system during development reviews. For that to be effective when the requirements model is large needs all the more stronger and richer exemplars. At this stage, TORUS only provides the basic information that is needed to represent the current state of a requirement. Going beyond this to produce more compelling views is a challenging topic that needs further research:

Managing scale by focusing on sections of the requirements is one approach. Many avionics and automotive systems organize their systems into broad sub-systems such as braking, navigation, engine management and environmental control. Within our workpiece color sorter, the horizontal and vertical piston controllers form natural sub-system divisions. The ability to hide splices that are not relevant to a discussion and focus only on complete sub-systems alone would declutter the view presented. Further refinements could allow only problem areas to remain visible or to highlight patterns exposed through the formal analysis TORUS performs on the entities. However, the sheer number of traces created in even small models hints at the graphic rendering challenges this problem presents. The refresh performance when changing views would have to be significant to ensure that it can keep pace with discussions going on within the teams during meetings, especially if these views are being shared in real-time between remotely-distributed teams. The success of the simple box and line paradigm adopted here suggests that developing more intuitive representations for splices and the skein would be a valuable contribution.

In the research methodology section, the works of Norman (2013) and Hevner (2004) were discussed. They asserted that efficacy of design was not enough; good designers have to incorporate both style and aesthetic's in their artifacts. Gelernter's description of *"machine beauty"* was touched upon, the marriage between simplicity and efficacy that drives innovation in science as much as it does in technology (Gelernter, 1999, page 3). Extending the visualization capabilities of TORUS would be fascinating and challenging research that would need some wicked thinking to tackle something that surely qualifies as one of Buchanan's *"wicked problems"* (Buchanan, 1992, page 15).

Using Splice History more constructively would bring a new depth to the analysis of problems. Examples were given of requirements that were complete that moved back to an uncompleted stage as function blocks were re-factored. The history aspects of splices were only explored briefly in this research but they present significant opportunities for future research. Each historical splice captures a snapshot of part of the system from a previous time. Such information would be invaluable when trying to determine how an implementation or requirement has evolved and changed. Formal methods can operate on historical splices as easily as they work on current ones, exposing deeper patterns and constraint violations to support design and diagnosis.

8.6 Final Thoughts

The introduction to this chapter mentioned dragons. So often, when mariners or explorers of the past were describing what they encountered on their journeys, they anthropomorphized their fears or challenges through tales of fearsome dragons. In so much of the academic literature presented during this thesis, traceability within large cyber-physical systems has been portrayed as an intractable, dragon-like problem. TORUS presented a novel way to explore potential solutions in a space that Finkelstein (2012) asserts is much in need of sound practices and tools.

However, in this work, I hope I have also conveyed my fascination with this challenge and the opportunities that further research in the area would present. If these wicked problems are indeed dragons, then they certainly have enticing personalities.

References

Alford, M. W. & Lawson. (1979). *Software Requirements Engineering Methodology (Development)* [Internet web page]. TRW Defence and Space Systems Group. Retrieved 11 December 2015, from `http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA073132`

Alippi, C. (2014). *Intelligence for Embedded Systems.* Springer.

ANSI/AAMI. (2006). Medical device Software - Software life cycle processes. *Association for the Advancement of Medical Instrumentation.*

Applegate, L. M. & King, J. (1999). Rigor and Relevance in MIS Research–Introduction. *MIS Quarterly*, *23*(1), 1–2.

Architect, E. (2010). *Sparx Systems.* Inc.

Arpinen, T., Hämäläinen, T. & Hännikäinen, M. (2011). Meta-model and UML profile for requirements management of software and embedded systems. *EURASIP Journal on Embedded Systems*, *2011*(1), 1.

Automotive, S. (2010). Automotive SPICE Process Assessment Model. *Final Release, v4*, *4*, 46.

AUTOSAR. (2016). The AUTOSAR Architectural Description Language. Retrieved 19 June 2016, from `http://www.autosar.org/find02_07.php`

Avison, D. E., Lau, F., Myers, M. D. & Nielsen, P. A. (1999). Action Research. *Communications of the ACM*, *42*(1), 94–97.

Barnat, J., Bauch, P., Beneš, N., Brim, L., Beran, J. & Kratochvíla, T. (2015). Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing*, 1–19.

Bell, T. E. & Thayer, T. A. (1976). Software requirements: Are they really a problem? In *Proceedings of the 2nd international conference on Software Engineering* (pp. 61–68).

Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A. & Letier, E. (2010). Requirements reflection: requirements as runtime entities. In *Proceedings of the 32nd acm/ieee international conference on software engineering-volume 2* (pp. 199–202).

Black, G. & Vyatkin, V. (2010). Intelligent component-based automation of baggage handling systems with IEC 61499. *Automation Science and Engineering, IEEE Transactions on*, *7*(2), 337–351.

Boehm, B. W., Brown, J. R. & Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on software engineering* (pp. 592–605).

Bohr, M. (2009). The new era of scaling in an SoC world. In *Solid-state circuits conference-digest of technical papers, 2009. isscc 2009. ieee international* (pp. 23–28).

Borkin, M. A., Vo, A. A., Bylinskii, Z., Isola, P., Sunkavalli, S., Oliva, A. & Pfister, H. (2013). What makes a visualization memorable? *Visualization and Computer Graphics, IEEE Transactions on*, *19*(12), 2306–2315.

Borland. (2016). *Borland CaliberRM.* Retrieved 19 June 2016, from `http://www.ktgcorp.com/borland/products/caliber/index.html`

Born, M., Favaro, J. & Kath, O. (2010). Application of ISO DIS 26262 in practice. In *Proceedings of the 1st workshop on critical automotive applications: Robustness & safety* (pp. 3–6).

Bowen, J. & Stavridou, V. (1993). Safety-critical systems, formal methods and standards. *Software Engineering Journal*, *8*(4), 189–209.

Bowen, J. P. & Hinchey, M. G. (1995). Ten commandments of formal methods. *Computer*, *28*(4), 56–63.

Bozzano, M., Cimatti, A., Katoen, J.-P., Katsaros, P., Mokos, K., Nguyen, V. Y., ... Roveri, M. (2014). Spacecraft early design validation using formal methods. *Reliability engineering & system safety*, *132*, 20–35.

Braun, P., Broy, M., Houdek, F., Kirchmayr, M., Müller, M., Penzenstadler, B., ... Weyer, T. (2014). Guiding requirements engineering for software-intensive embedded systems in the automotive industry. *Computer Science-Research and Development*, *29*(1), 21–43.

Breaux, T. D. & Gordon, D. G. (2013). Regulatory requirements traceability and analysis using semi-formal specifications. In *International working conference on requirements engineering: Foundation for software quality* (pp. 141–157).

Breen, S., Aranda, S., Ritchie, D., Kofoed, S., Maguire, R. & Kearney, N. (2012). You cannot manage what you cannot measure. *assessment of a prototype remote monitoring system for haematological cancer patients undergoing chemotherapy. J Supp Care Cancer*, *21*(Suppl 1), S69.

Brewster, D. (1860). *Memoirs of the life, writings, and discoveries of Sir Isaac Newton* (Vol. 2). Edmonston and Douglas.

Brooks, F. P. (1975). *The Mythical Man-Month* (Vol. 1995). Addison-Wesley Reading, MA.

Brooks, F. P. (1987). No silver bullet: essence and accidents in software engineering. *IEEE Computer*, *20*, 10–19.

Broy, M., Kruger, I. H., Pretschner, A. & Salzmann, C. (2007). Engineering Automotive Software. *Proceedings of the IEEE*, *95*(2), 356–373.

Broy, M. & Schmidt, A. (2014). Challenges in engineering cyber-physical systems. *Computer*(2), 70–72.

Buchanan, R. (1992). Wicked problems in design thinking. *Design issues*, *8*(2), 5–21.

Charette, R. N. (1991). *Applications strategies for risk analysis*. Intertext Publications, Inc.,/McGraw-Hill, Inc.

Charette, R. N. (2009). This car runs on code. *IEEE Spectrum*, *46*(3), 3.

Christensen, J. (2016). *The FBDK Function Block Development Kit.* Retrieved 29 May 2016, from `http://www.holobloc.com/doc/fbdk/`

Christensen, J. H., Strasser, T., Valentini, A., Vyatkin, V., Zoitl, A., Chouinard, J., . . . Kopitar, A. (2012). The IEC 61499 Function Block Standard: Software Tools and Runtime Platforms. *ISA Automation Week.*

Chung, L., Nixon, B. A., Yu, E. & Mylopoulos, J. (2012). *Non-functional requirements in software engineering* (Vol. 5). Springer Science & Business Media.

Cleland-Huang, J., Gotel, O. & Zisman, A. (2012). *Software and systems traceability* (Vol. 2) (No. 3). Springer.

Cleland-Huang, J., Heimdahl, M., Hayes, J. H., Lutz, R. & Maeder, P. (2012). Trace queries for safety requirements in high assurance systems. In *International working conference on requirements engineering: Foundation for software quality* (pp. 179–193).

Condra, L. (2015). *Electronic Component Obsolescence* [Internet web page]. Boeing Commercial Airplanes Group. Retrieved 11 December 2015, from `http://www.boeing.com/commercial/aeromagazine/aero_10/elect_textonly.html`

Conway, M. E. (1968). How do committees invent? *Datamation*, *14*(4), 28–31.

Corporation, S. (2016). *3SL Cradle.* Retrieved 19 June 2016, from `http://www.threesl.com/`

Dehlinger, J. & Lutz, R. R. (2004). Software fault tree analysis for product lines. In *High assurance systems engineering, 2004. proceedings. eighth ieee international symposium on* (pp. 12–21).

Delligatti, L. (2013). *SysML distilled: A brief guide to the systems modeling language*. Addison-Wesley.

Denning, P. J. (1997). A new social contract for research. *Communications of the ACM*, *40*(2), 132–134.

Diepenbeck, M., Kühne, U., Soeken, M. & Drechsler, R. (2014). Behaviour driven development for tests and verification. In *Tests and proofs* (pp. 61–77). Springer.

Docrill, P. (2015). Volkswagen worker grabbed and killed in German Plant. Retrieved 14 December, 2015, from `http://www.sciencealert.com/volkswagenworkergrabbedandkilledbyrobotingermanplant`

Dorfman, M. & Flynn, R. F. (1984). ARTS - an automated requirements traceability system. *Journal of Systems and Software*, *4*(1), 63–74.

Dos Santos, P. S. M. & Travassos, G. H. (2011). Action research can swing the balance in experimental software engineering. *Advances in Computers.*

Drechsler, R. & Kühne, U. (2015). Formal Modeling and Verification of Cyber-Physical Systems.

Dubinin, V. & Vyatkin, V. (2008). On definition of a formal model for IEC 61499 function blocks. *EURASIP Journal on Embedded Systems*, *2008*, 7.

Dubois, H., Peraldi-Frati, M.-A. & Lakhal, F. (2010). A model for requirements traceability in a heterogeneous model-based design process: Application to automotive embedded systems. In *Engineering of complex computer systems (iceccs), 2010 15th ieee international conference on* (pp. 233–242).

Easterbrook, S., Singer, J., Storey, M.-A. & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285–311). Springer.

Eclipse. (2016a). Eclipse ProR Workbench. Retrieved 19 June 2016, from `http://www.eclipse.org/rmf/pror/`

Eclipse. (2016b). Eclipse ReqCycle Workbench. Retrieved 19 June 2016, from `http://www.eclipse.org/proposals/polarsys.reqcycle/`

Egyed, A., Biffl, S., Heindl, M. & Grünbacher, P. (2005). Determining the cost-quality trade-off for automated software traceability. In *Proceedings of the 20th ieee/acm international conference on automated software engineering* (pp. 360–363).

El Emam, K. & Koru, A. G. (2008). A replicated survey of IT software project failures. *IEEE software*, *25*(5), 84–90.

Estefan, J. A. (2010). Developing a Strategy and Roadmap for Advancing the State-of-the-Practice of MBSE within Your Organization. George Mason University.

FAA. (2011). Software Considerations in Airborne Systems and Equipment Certification.

Fernández, D. M., Penzenstadler, B., Kuhrmann, M. & Broy, M. (2010). A meta model for artefact-orientation: fundamentals and lessons learned in requirements engineering. In *Model driven engineering languages and systems* (pp. 183–197). Springer.

Finkelstein, A. (2012). Requirements and Relationships: A Foreword. *Software and Systems Traceability. London: Springer London*, *6*.

Gaeta, J. P. & Czarnecki, K. (2015). Modeling aerospace systems product lines in SysML. In *Proceedings of the 19th international conference on software product line* (pp. 293–302).

Gartner. (2014). *Market Guide for Software Requirements Definition and Management Solutions* [Internet web page]. Gartner, Inc. (US). Retrieved 13 March 2016, from `https://www.gartner.com/doc/2866625/market-guide-software-requirements-definition`

Gelernter, D. (1999). *Machine Beauty: Elegance and the Heart of Computing (Repr ed).* Basic Books.

Gerber, C., Hanisch, H.-M. & Ebbinghaus, S. (2008). From IEC 61131 to IEC 61499 for distributed systems: a case study. *EURASIP Journal on Embedded Systems*, *2008*, 4.

Gerring, J. (2006). Single-Outcome Studies A Methodological Primer. *International Sociology*, *21*(5), 707–734.

Gervasi, V. & Zowghi, D. (2010). On the role of ambiguity in RE. In *Requirements engineering: Foundation for software quality* (pp. 248–254). Springer.

Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P. & Antoniol, G. (2012). The quest for ubiquity: A roadmap for software and systems traceability research. In *Requirements engineering conference (re), 2012 20th ieee international* (pp. 71–80).

Gotel, O. & Mäder, P. (2012). Acquiring tool support for traceability. In *Software and systems traceability* (pp. 43–68). Springer.

Gotel, O. C. & Finkelstein, A. C. (1994). An analysis of the requirements traceability problem. In *Requirements engineering, 1994., proceedings of the first international conference on requirements engineering* (pp. 94–101).

Graefe, R. & Basson, A. H. (2014). Control of Reconfigurable Manufacturing Systems Using Object-Oriented Programming. In *Enabling manufacturing competitiveness and economic sustainability* (pp. 231–236). Springer.

Guessi, M., Cavalcante, E. & Oliveira, L. B. (2015). Characterizing architecture description languages for software-intensive systems-of-systems. In *Proceedings of the third international workshop on software engineering for systems-of-systems* (pp. 12–18).

Guessi, M., Oliveira, L. B. R., Garcés, L. & Oquendo, F. (2015). Towards a Formal Description of Reference Architectures for Embedded Systems. In *Proceedings of the 1st international workshop on exploring component-based techniques for constructing reference architectures* (pp. 17–20).

Hametner, R., Kormann, B., Vogel-Heuser, B., Winkler, D. & Zoitl, A. (2013). Automated test case generation for industrial control applications. In *Recent advances in robotics and automation* (pp. 263–273). Springer.

Hamilton, M. & Zeldin, S. (1976). Higher order software - A methodology for defining software. *Software Engineering, IEEE Transactions on*(1), 9–32.

Hamilton, M. & Zeldin, S. (1980). The relationship between design and verification. *Journal of Systems and Software*, *1*, 29–56.

Hänninen, K., Maki-Turja, J. & Nolin, M. (2006). Present and future requirements in developing industrial embedded real-time systems-interviews with designers in the vehicle domain. In *Engineering of computer based systems, 2006. ecbs 2006. 13th annual ieee international symposium and workshop on* (pp. 9–pp).

Hansen. (2005). Software-defined features. In *Hansen report on automotive electronics. a business and technology newsletter, vol 18, no 6, july/august 2005.*

Hardung, B., Kölzow, T. & Krüger, A. (2004). Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th acm international conference on embedded software* (pp. 203–210).

Haskins, C. (2011). A historical perspective of MBSE with a view to the future. In *Incose international symposium* (Vol. 21, pp. 493–509).

Hayes, J. H., Dekhtyar, A., Sundaram, S. K., Holbrook, E. A., Vadlamudi, S. & April, A. (2007). REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, *3*(3), 193–202.

Hegny, I., Hummer, O., Zoitl, A., Koppensteiner, G. & Merdan, M. (2008). Integrating software agents and IEC 61499 realtime control for reconfigurable distributed manufacturing systems. In *Industrial embedded systems, 2008. sies 2008. international symposium on* (pp. 249–252).

Hevner, A. & Chatterjee, S. (2010). *Design science research in information systems*. Springer.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105.

Hill, K., Menk, D. & Swiecki, B. (2016). *Just How High-Tech is the Automotive Industry?* [Internet web page]. Center for Automotive Research (CAR). Retrieved 11 July 2016, from `http://www.cargroup.org/?module=Publications&event=View&pubID=103#sthash.T5d2ik2u.dpuf`

Ho, A. (2015). What is the most dangerous field in computer science? Retrieved 14 December, 2015, from `https://www.quora.com/What-is-the-most-dangerous-field-in-computer-science-Why`

Hoffman, K., Basson, A. H. & le Roux, A. (2014). Towards Alternatives for Agent Based Control in Reconfigurable Manufacturing Systems. In *Enabling manufacturing competitiveness and economic sustainability* (pp. 237–242). Springer.

Hollan, J., Hutchins, E. & Kirsh, D. (2000). Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *7*(2), 174–196.

Holloway, C. M. (1997). Why engineers should consider formal methods. In *Digital avionics systems conference, 1997. 16th dasc., aiaa/ieee* (Vol. 1, pp. 1–3).

Hutchins, E. (1995). *Cognition in the Wild.* MIT press.

IBM. (2016). *Rational Rhapsody.* Inc.

IEC. (2013). IEC 61499-1: Function blocks-part 1 architecture. *International Standard, First Edition, Geneva*, *1*.

IEEE. (1990). IEEE Standard Glossary of Software Engineering Terminology. *Office*, *121990*(1), 1.

ISaGRAF, I. T. (2010). *The ISaGRAF Workbench.* Retrieved 29 May, 2016, from `http://www.isagraf.com/index.htm`

ISO. (2015). *ISO Standard 15288:2015 Systems and software engineering - System life cycle processes.* Retrieved 29 June, 2016, from `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=63711`

ISO. (2016). *ISO Standard ISO-26262 Road Vehicles - Functional Safety.* Retrieved 27 June, 2016, from `https://www.iso.org/obp/ui/#iso:std:iso:26262:-9:ed-1:v1:en:sec:intro`

ISO/IEC. (1997). Information Technology. Software Lifecycle Processes. *International Standards Organisation (ISO) and International Electrotechnical Commission (IEC).*

Jamro, M. (2014). SysML modeling of POU-oriented unit tests for IEC 61131-3 control software. In *Methods and models in automation and robotics (mmar), 2014 19th international conference on* (pp. 82–87).

Kamen, E. (1999). Chapter 8: Ladder Logic Diagrams and PLC Implementations. *Industrial Controls and Manufacturing1999, Academic*, 141–163.

Kamsties, E., Berry, D. M., Paech, B., Kamsties, E., Berry, D. & Paech, B. (2001). Detecting ambiguities in requirements documents using inspections. In *Proceedings of the first workshop on inspection in software engineering (wise'01)* (pp. 68–80).

Khaitan, S. K. & McCalley, J. D. (2014). Design techniques and applications of cyberphysical systems: a survey.

Kleines, H., Detert, S., Drochner, M. & Suxdorf, F. (2008). Performance aspects of PROFINET IO. *Nuclear Science, IEEE Transactions on*, *55*(1), 290–294.

Krüger, I. H., Nelson, E. C. & Prasad, K. V. (2004). *Service-based software development for automotive applications* (Tech. Rep.). SAE Technical Paper.

Kukafka, R., Johnson, S. B., Linfante, A. & Allegrante, J. P. (2003). Grounding a new information technology implementation framework in behavioral science: a systematic analysis of the literature on IT use. *Journal of biomedical informatics*, *36*(3), 218–227.

Lee, E. A. (2008). Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on* (pp. 363–369).

Lee, J., Bagheri, B. & Kao, H.-A. (2015). A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, *3*, 18–23.

Leffingwell, D. (1997). Calculating your return on investment from more effective requirements management. *American Programmer*, *10*(4), 13–16.

Leitao, P., Karnouskos, S., Ribeiro, L., Lee, J., Strasser, T. & Colombo, A. W. (2016). Smart Agents in Industrial Cyber–Physical Systems. *Proceedings of the IEEE*, *104*(5), 1086–1101.

Li, F.-L., Horkoff, J., Borgida, A., Guizzardi, G., Liu, L. & Mylopoulos, J. (2015). From Stakeholder Requirements to Formal Specifications Through Refinement. In *Requirements engineering: Foundation for software quality* (pp. 164–180). Springer.

Lindgren, P., Lindner, M., Lindner, A., Vyatkin, V., Pereira, D. & Pinho, L. M. (2015). A real-time semantics for the IEC 61499 standard. In *Emerging technologies & factory automation (etfa), 2015 ieee 20th conference on* (pp. 1–6).

Lindland, O. I., Sindre, G. & Solvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE software*, *11*(2), 42–49.

Loia, V. & Vaccaro, A. (2011). A decentralized architecture for voltage regulation in Smart Grids. In *Industrial electronics (isie), 2011 ieee international symposium on* (pp. 1679–1684).

Lutz, R. (1993, Jan). Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on* (p. 126-133). doi: 10.1109/ISRE.1993.324825

Mäder, P., Gotel, O. & Philippow, I. (2009a). Enabling automated traceability maintenance through the upkeep of traceability relations. In *Model driven architecture-foundations and applications* (pp. 174–189).

Mäder, P., Gotel, O. & Philippow, I. (2009b). Motivation matters in the traceability trenches. In *Requirements engineering conference, 2009. re'09. 17th ieee international* (pp. 143–148).

Maletic, J., Collard, M. L., Marcus, A. et al. (2002). Source code files as structured documents. In *Program comprehension, 2002. proceedings. 10th international workshop on* (pp. 289–292).

Martins, L. E. G. & Gorschek, T. (2016). Requirements engineering for safety-critical systems: A systematic literature review. *Information and Software Technology*, *75*, 71–89.

Marwedel, P. (2010). Embedded and cyber-physical systems in a nutshell. *DAC. COM Knowledge Center Article*, *20*(10).

Matisse. (1894). *Woman Reading*. Musée National d'Arte Moderne Centre Georges Pompidou, Paris.

Maury, J.-P. (1992). *Newton: understanding the cosmos*. Thames & Hudson.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*(4), 308–320.

McKinsey. (2014). What's Driving the connected car? Retrieved 29 June, 2016, from `http://www.mckinsey.com/industries/automotive-and-assembly/our-insights/whats-driving-the-connected-car`

Meyer, B. (1995). From process to product: Where is software headed? *IEEE Computer*, *28*(8), 23.

Microsoft. (2016). Microsoft Visual Studio Team Foundation Server 2016. Retrieved 19 June 2016, from `http://blogs.objectsharp.com/post/2012/02/22/Requirements-Traceability-in-Visual-Studio.aspx`

Mitschke, Andreas and Loughran Neil and Josko Bernhard and Häusler Stefan and Dierks Henning. (2010). *Definition and exemplification of RSL and RMM* [Internet web page]. CESAR. Retrieved 11 December 2015, from `http://www.cesarproject.eu/fileadmin/userupload`

Modbus, I. (2004). Modbus application protocol specification v1. 1a. *North Grafton, Massachusetts (www. modbus. org/specs. php).*

Monin, J.-F. (2012). *Understanding Formal Methods*. Springer Science & Business Media.

Morel, G., Panetto, H., Zaremba, M. & Mayer, F. (2003). Manufacturing enterprise control and management system engineering: paradigms and open issues. *Annual reviews in Control*, *27*(2), 199–209.

Mousavi, A., Yang, C.-W., Pang, C. & Vyatkin, V. (2014). Energy efficient automation model for office buildings based on ontology, agents and IEC 61499 function blocks. In *Emerging technology and factory automation (etfa), 2014 ieee* (pp. 1–7).

Naur, P. & Randell, B. (1969). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.

Norman, D. A. (2013). *The design of everyday things: Revised and expanded edition*. Basic books.

Noy, N. F., McGuinness, D. L. et al. (2001). *Ontology development 101: A guide to creating your first ontology*. Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880.

Nunamaker Jr, J. F., Chen, M. & Purdin, T. D. (1990). Systems development in information systems research. *Journal of management information systems*, *7*(3), 89–106.

nxtControl GmbH. (2016a). *The nxtStudio Development Environment.* Retrieved 8 January 2016, from `http://www.nxtcontrol.com/en`

nxtControl GmbH. (2016b). *The nxtStudio nxtHIM visualisation toolset for SCADA.* Retrieved 30 July 2016, from `http://www.nxtcontrol.com/en/visualisation/`

OMG. (2016a). *The Object Management Group Systems Modeling Language v1.3.* Retrieved 21 May, 2016, from `http://www.omgsysml.org/`

OMG. (2016b). *OMG XML Metadata Interchange (XMI) 2.5.1).* Retrieved 21 May, 2016, from `http://www.omg.org/spec/XMI/2.5.1/`

Omoronyia, I., Sindre, G., Stålhane, T., Biffl, S., Moser, T. & Sunindyo, W. (2010). A domain ontology building process for guiding requirements elicitation. In *International working conference on requirements engineering: Foundation for software quality* (pp. 188–202).

Ortel, M., Malot, M., Baumgart, A., Becker, J. S., Bogusch, R., Farfeleder, S., . . . others (2013). Requirements Engineering. In *Cesar-cost-efficient methods and processes for safety-relevant embedded systems* (pp. 69–143). Springer.

Osswald, S., Matz, S. & Lienkamp, M. (2014). Prototyping automotive cyber-physical systems. In *Proceedings of the 6th international conference on automotive user interfaces and interactive vehicular applications* (pp. 1–6).

Penzenstadler, B. & Eckhardt, J. (2012). A Requirements Engineering content model for Cyber-Physical Systems. In *Ress* (pp. 20–29).

Pesce, M. (2011). Software Takes On More Tasks in Today's Cars. Retrieved 29 June, 2016, from `http://www.wired.com/2011/04/the-growing-role-of-software-in-our-cars`

Pooley, R. & Stevens, P. (1998). Using uml.

Pretschner, A., Broy, M., Kruger, I. H. & Stauner, T. (2007). Software engineering for automotive systems: A roadmap. In *2007 future of software engineering* (pp. 55–71).

Rajan, A. & Wahl, T. (2013). *CESAR: Cost-efficient Methods and Processes for Safety-relevant Embedded Systems* (No. 978-3709113868). Springer.

Rajkumar, R. R., Lee, I., Sha, L. & Stankovic, J. (2010). Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th design automation conference* (pp. 731–736).

Ramesh, B. & Jarke, M. (2001). Toward reference models for requirements traceability. *IEEE transactions on software engineering*, *27*(1), 58–93.

Ramesh, B., Powers, T., Stubbs, C. & Edwards, M. (1995). Implementing requirements traceability: a case study. In *Requirements engineering, 1995., proceedings of the second ieee international symposium on* (pp. 89–95).

Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M. & Törner, F. (2013). Increasing Efficiency of ISO 26262 Verification and Validation by Combining Fault Injection and Mutation Testing with Model based Development. In *Icsoft* (pp. 251–257).

Regnell, B., Svensson, R. B. & Wnuk, K. (2008). Can we beat the complexity of very large-scale requirements engineering? In *Requirements engineering: Foundation for software quality* (pp. 123–128). Springer.

Riedl, M., Zipper, H., Meier, M. & Diedrich, C. (2014). Cyber-physical systems alter automation architectures. *Annual Reviews in Control*, *38*(1), 123–133.

Royce, W. W. (1970). Managing the development of large software systems. In *proceedings of ieee wescon* (Vol. 26, pp. 328–388).

Runeson, P. & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, *14*(2), 131–164.

Ryan, A. J. (2000). An approach to quantitative non-functional requirements in software development. In *Systems engineering–a key to competitive advantage for all industries: proceedings of the 2nd european systems engineering conference (eusec 2000), munich* (p. 73).

Ryan, M. J., Wheatcraft, L. S., Dick, J. & Zinni, R. (2015). On the Definition of Terms in a Requirements Expression. , *25*(1), 169–181.

SAE. (2016a). The ARP4754A Standard - Guidelines for Development of Civil Aircraft and Systems . Retrieved 26 June, 2016, from `http://standards.sae.org/arp4754a`

SAE. (2016b). EIA Standard 632 - Processes for Engineering a System. Retrieved 29 June, 2016, from `http://standards.sae.org/eia632/`

Saldivar, A. A. F., Li, Y., Chen, W.-n., Zhan, Z.-h., Zhang, J. & Chen, L. Y. (2015). Industry 4.0 with cyber-physical integration: A design and manufacture perspective. In *Automation and computing (icac), 2015 21st international conference on* (pp. 1–6).

Sandhu, G. (2015). Analysis of Modeling Techniques used for Translating Natural Language Specification into Formal Software Requirements. *International Journal of Computer Applications*, *113*(1).

Schoffelen, J., Claes, S., Huybrechts, L., Martens, S., Chua, A. & Moere, A. V. (2015). Visualising things. Perspectives on how to make things public through visualisation. *CoDesign*, *11*(3-4), 179–192.

Scientific, A. (2016). Atlas Scientific ENV-RGB v1.6 Solid-State Camera data sheet. Retrieved 29 May, 2016, from `https://www.atlas-scientific.com/_files/_datasheets/_probe/ENV-RGB.pdf`

Shahid, M., Ibrahim, S. & Mahrin, M. N. (2011). An Evaluation of Requirements Management and Traceability Tools. *World Academy of Science, Engineering and Technology, WASET*.

Siewiorek, D. P. & Narasimhan, P. (2005). Fault-tolerant architectures for space and avionics applications. *NASA Ames Research http://ic. arc. nasa. gov/projects/ishem/Papers/Siewi*.

Simon, H. A. (1996). *The Sciences of the Artificial 3rd ed.* MIT Press.

Sinha, R., Roop, P., Shaw, G., Salcic, Z. & Kuo, M. (2016). Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks. , 59-68.

Siricharoen, W. V. (2013). Infographics: The new communication tools in digital age. In *The international conference on e-technologies and business on the web (ebw2013)* (pp. 169–174).

Sjøberg, Dag IK and Dyba, Tore and Jorgensen, Magne. (2007). The future of empirical methods in software engineering research. In *2007 future of software engineering* (pp. 358–378).

Spanoudakis, G., Zisman, A., Pérez-Minana, E. & Krause, P. (2004). Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, *72*(2), 105–127.

Standish, G. (1994). The chaos report. *The Standish Group*.

Stephenson, D. (2006). Boeing: The Airplane Doctors. Retrieved 14 December, 2015, from `http://www.boeing.com/news/frontiers/archive/2006/august/ts_sf09.pdf`

Stojanovic, N. D., Stojanovic, L. & Stuehmer, R. (2013). Tutorial: personal big data management in the cyber-physical systems-the role of event processing. In *Proceedings of the 7th acm international conference on distributed event-based systems* (pp. 281–288).

Strasser, T., Rooker, M., Ebenhofer, G., Zoitl, A., Sünder, C., Valentini, A. & Martel, A. (2008). Framework for distributed industrial automation and control (4DIAC). In *Industrial informatics, 2008. indin 2008. 6th ieee international conference on* (pp. 283–288).

Strasser, T., Stifter, M., Andrén, F., De Castro, D. B. & Hribernik, W. (2011). Applying open standards and open source software for smart grid applications: Simulation of distributed intelligent control of power systems. In *Power and energy society general meeting, 2011 ieee* (pp. 1–8).

TC65, I. (1993). IEC 61131: Programmable Controllers-Part 3: Programming languages. *International Electrotechnical Commission (IEC).*

Tranoris, C. & Thramboulidis, K. (2003). Integrating UML and the function block concept for the development of distributed control applications. In *Emerging technologies and factory automation, 2003. proceedings. etfa'03. ieee conference* (Vol. 2, pp. 87–94).

USDOL. (2015). United States Department of Labor Safety and Health Topics - Robotics. Retrieved 14 December, 2015, from `https://www.osha.gov/SLTC/robotics/`

Van Lamsweerde, A. (2009). *Requirements engineering: from system goals to UML models to software specifications.* Wiley Publishing.

Vierhauser, M., Rabiser, R., Grunbacher, P. & Aumayr, B. (2015). A requirements monitoring model for systems of systems. In *Requirements Engineering Conference (RE), 2015 IEEE 23rd International* (pp. 96–105).

Vyatkin, V. (2009). The IEC 61499 standard and its semantics. *Industrial Electronics Magazine, IEEE*, *3*(4), 40–48.

Vyatkin, V. (2011). IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *Industrial Informatics, IEEE Transactions on*, *7*(4), 768–781.

Vyatkin, V. & Chouinard, J. (2008). On Comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations. In *6th ieee international conference on industrial informatics* (pp. 264–269).

Wadsworth, B. J. (1996). *Piaget's theory of cognitive and affective development: Foundations of constructivism* . Longman Publishing.

Weber, R. et al. (1997). *Ontological foundations of information systems*. Coopers & Lybrand and the Accounting Association of Australia and New Zealand Melbourne.

Wiegers, K. & Beatty, J. (2013). *Software requirements*. Pearson Education.

Wnuk, K., Regnell, B. & Schrewelius, C. (2009). Architecting and coordinating thousands of requirements–an industrial case study. In *Requirements engineering: Foundation for software quality* (pp. 118–123). Springer.

Wohlin, C. & Aurum, A. (2015). Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, *20*(6), 1427–1455.

Wolf, M. (2014). *High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing*. Newnes.

Wolff, C., Brink, C., Höttger, R., Igel, B., Kamsties, E., Krawczyk, L. & Lauschner, U. (2015). Automotive software development with AMALTHEA. *Practice and Perspectives*, 432.

Yan, J. & Vyatkin, V. V. (2011). Distributed execution and cyber-physical design of Baggage Handling automation with IEC 61499. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on* (pp. 573–578).

Ying, A. T., Murphy, G. C., Ng, R. & Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, *30*(9), 574–586.

Yoong, L. H., Roop, P. S. & Salcic, Z. (2012). Implementing constrained cyber-physical systems with IEC 61499. *ACM Transactions on Embedded Computing Systems (TECS)*, *11*(4), 78.

Yoong, L. H., Roop, P. S., Vyatkin, V. & Salcic, Z. (2009). A synchronous approach for IEC 61499 function block implementation. *Computers, IEEE Transactions on*, *58*(12), 1599–1614.

Zell-Ravenheart, Oberon. (2016). Living Dragons. Retrieved 15 July 2016, from `http://newpagebooks.blogspot.co.nz/2015/11/creature-of-month-living-dragons-by.html`

Zhabelova, G. & Vyatkin, V. (2012). Multiagent smart grid automation architecture based on IEC 61850/61499 intelligent logical nodes. *Industrial Electronics, IEEE Transactions on*, *59*(5), 2351–2362.