Extended Programming and Design:

A Language and Toolset for Integrating Requirements, Architecture, Design and Implementation when Developing Complex Software Systems

A Thesis
Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Science
with a
Major in Computer Science
in the
College of Graduate Studies
University of Idaho
by
Sanjeev Shrestha

Major Professor: Daniel Conte de Leon, Ph.D.
Committee Members: James Alves-Foss, Ph.D.; Axel W. Krings, Ph.D.
Department Administrator: Frederick Sheldon, Ph.D.

December 2015

Authorization to Submit Thesis

This thesis of Sanjeev Shrestha, submitted for the degree of Master of Science with a Major in Computer Science and titled **"Extended Programming and Design: A Language and Toolset for Integrating Requirements, Architecture, Design and Implementation when Developing Complex Software Systems,"** has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ Date: _____
Daniel Conte de Leon, Ph.D.

Committee members: _____ Date: _____
James Alves-Foss, Ph.D.

_____ Date: _____
Axel W. Krings, Ph.D.

Department Administrator: _____ Date: _____
Frederick Sheldon, Ph.D.

Abstract

Complex software systems are developed by engineers with diverse backgrounds using different software development methods such as traditional, formal, and agile. Increasing software complexity coupled with the lack of holistic system models, ineffective communication between engineers, and the lack of up-to-date system documentation increases the likelihood of faults and failures of system security and safety. This thesis, introduces EXMPLRAD: a language for specifying software project artifacts such as informal requirements, design and architectural descriptions, and source code. This thesis also present SyModEx2, an expert tool that analyzes EXMPLRAD specifications and verifies the absence of hidden dependencies between system components. A case study of applying these technique and associated toolset to the SEL4 micro-kernel's Interprocess Communication architecture is presented.

Acknowledgements

I would like to thank my advisor, Dr. Daniel Conte de Leon for his support, encouragement, and wonderful edits.

I would also like to thank my committee members, Dr. Jim Alves-Foss and Dr. Axel W. Krings for their valuable input on my thesis.

I would like to thank all of my professors and teachers who have helped to impart the knowledge which has helped me along in my academic career.

I would also like to thank my friends and family who have been supportive of my endeavors.

I would like to thank Anirudh Bandari for making this LaTeX thesis template.

This thesis work was made possible with the financial support of a State of Idaho IGEM grant with the Center for Secure and Dependable Systems at the University of Idaho.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Problem Statement, Proposed Solution, and Thesis Outline

Software systems even though widely used, may present errors due to inadequate understanding of the system and its environment, hidden component dependencies, and poor, inadequate, or inconsistent documentation. Hence, we need techniques and toolsets which help solve these problems across different software development methodologies, identify specification and implementation related issues, and help discover hidden dependencies within a software system.

In this introductory chapter, Section 1.1 describes problems that maybe encountered in various software development strategies at different phases of the software development life-cycle. Secondly, Section 1.2 provides an overview of our proposed techniques and toolset for an implementation-oriented formalization of a system model. This techniques and toolset may be used to help discover safety and security related vulnerabilities. The techniques and toolsets may also be used to validate the system implementation against its stated requirements by enforcing traceability relationships between various model components, located at different abstraction levels. Finally, Section 1.3 provides an outline of the remaining structure for this thesis document.

1.1 Current Problems with Different Software Development Strategies

Currently, software intensive systems are developed by applying a software development process model. These software development methods are: a) Traditional methods, b) Formal methods and c) Agile methods. The practice of specifying requirements, creating requirement specification documents, discussing possible software designs that satisfy the requirements, authoring design diagrams and documents, developing the software and writing user and technical documentation, is typically viewed as applying a traditional software development method [42, 66, 68, 73, 76, 81, 82].

A good introduction to Traditional and Agile methods is presented in Meyer's book [66]. This book provides a compendium of comparisons and criticisms between the two approaches and suggestions regarding the introduction of Agile methods into a software development organization.

One criticism faced by traditional software development methods is their inability to adequately handle requirement changes during the life-cycle of the system [36, 55, 66, 78, 88].

A customer may not have all the information regarding the process workflow of the system [3, 66, 74, 78, 88]. Hence, the problem that lies herein is that, the customer may not be self-aware of what they need from a software system during the preliminary stages of development. During iterations, it's frequent that we hear the client saying, "can we add this small feature", and "can we make this small change". These "small" changes are rarely "small" and often software designs need to be reviewed and redesigned to accommodate these changes [66, 68]. Thus, the resulting software is "patched" which in-turn leads to poor software quality and software bugs [42, 66, 68, 73].

Formal methods have been successful in finding various faults in software before they actually result in errors [23, 56]. However, the learning curve for using formal methods is very high and a fair amount of expertise is required before being able to apply them successfully [22–24, 36, 60]. Formal software development methods are widely used to create highly reliable software programs where software failure can lead to critical or disastrous outcomes during the execution of the program. Due to high level of expertise required, any documentation done during these iterations can only be understood by an expert user [24, 36]. Another problem that exists, is being able to correctly derive the connection between the formal software model and the actual working system [3].

Agile methods provide practices for developing software systems which focus on customer priorities, team performance, sustainable pace of development, and test driven development [36, 55, 66, 79]. Agile methods provide techniques on how to handle changing requirements, and avoid "documentation" [66]. However, these solutions have been subject to different criticisms. While it is true that evaluating the full nature of a software program at preliminary stages is hard, the Agile idea to avoid requirement specification and focus on event-based "user stories" is hardly the best solution [3, 55, 79]. Even though the full requirements may not be known and are subject to constant change, shunning the engineering practice of requirement analysis can prove to be detrimental rather than useful [66]. Another problem with Agile is terming documents as "waste" [66]. While it is a valid argument that heavy documentation is a burden rather than advantage for some businesses; in certain situations, the idea of considering all documentation as waste is an engineering hazard rather than a good practice [55, 66, 79]. More information regarding traditional, formal, and agile development methods is provided in Section 2.2.

Hence, assimilating all this information from the above discussion and from Meyer's book [66], we have derived a list of problems present in the software development methods:

1. **Inability to specify requirements clearly**: The "general" nature of requirements in traditional approaches make it hard to clearly express them in the requirement specification document [82]. This results in confusion on the developer's side and may further result in project failure. On the other hand, formal methods express requirements in a mathematical form which becomes hard for untrained developers to understand [60]. Finally, in Agile, written requirements are rejected and they are expressed as event based entities, termed "user stories", which focus on functional requirements forgetting the importance of non-functional requirements such as security and safety [66]. These specific event based stories fail to take into consideration the general nature of a software module. Hence, our goal is to find an approach which makes it simple for all project stakeholders, to incorporate and express requirements and abstraction levels granularly.

   This thesis, proposes Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD) which is capable of describing all system specifications separated at various abstraction levels.

2. **Ad-hoc organization of specifications**: During the requirements analysis phase, the focus is on figuring out the requirements [82]. A step further down this approach yields the information that seemingly different features may have common components shared between them. While UML diagrams do capture a snapshot of this relationship during the project life-cycle, they do not adapt well as the software system changes [34, 66]. A framework which captures this information dynamically and creates the system model can help a developer visualize the system. Even in modern Agile methods the relationship between different user stories is not properly analyzed. The relationship and grouping between different user stories can possibly result in finding missing scenarios which can help avoid critical software mistakes and omissions [21].

   For better organization of specifications, we need a centralized system model that can capture and archive specifications as they are created and changed. EXMPLRAD collects all the specification components present in the system and stores them as facts in a Prolog based knowledge-base.

3. **Changes in software are hard to manage**: A common problem during project development is that functionality changes are requested during or after the creation of the original functionality [36, 66]. Archiving the changes is hard and cumbersome which has lead the Agile model to skip documenting the change procedure altogether. This idea has more detrimental side-effects than advantages [66]. It leads to confusion and lack of knowledge when new functionality is being developed on top of the current functionality. Hence, this can lead to loss of time for analyzing the current procedural workflow of the system and result in project failure or program errors later during the project life-cycle.

   This thesis proposes performing changes in the corresponding specification definition written in EXMPLRAD language. Hence, any changes made will be promptly reflected and correct information will be promptly available to all project stakeholders.

4. **Hard to find a balance between over-documentation and no-documentation**: While traditional and formal approaches strongly focus on creating documentation at each project life-cycle stage; the Agile approach says no to documentation [66, 74]. The reason they provide is that documentation doesn't accompany the project deliverable so they consider a waste of time and resources. The Lean software method goes as far as tagging documentation as a kind of "waste" [66]. Hence, we need a documentation approach that enables documentation that is easy to read, simple to change, and can be stored and retrieved effortlessly [73, 81].

   While our approach does try to include documentation in the form of specification definitions in EXMPLRAD. It does not focus on solving this problem as a whole. As the problem more specifically relates to qualitative factor, we need an approach which would make documentation efforts more effective and usable.

5. **Incorporate testing in all phases of project life-cycle**: Testing is one of the primary mechanism to check the correctness of software functionality [42, 68, 73, 81]. Testing exists in many different flavors such as, unit testing, integration testing, exhaustive testing, and coverage testing. While traditional software development methods focus on testing at the final stages of development, test Driven Development proposes to first create the tests and then create the functionality to satisfy these tests [66]. However, with this new rise in interest for testing, the problem of testing between the specification components and the functionality still remains to be solved [23, 24]. Modern static and dynamic tools

are available for checking the software code. A mechanism is still needed to verify the correctness between the requirements, architecture, design and implementation components of a system[23, 24].

Testing is essential in complex software intensive applications for providing correct functionality and should be carried out in each stage of the project life-cycle. However, our approach focuses on mapping of specification level components and enabling the linkage between higher level components and tested components.

6. **Accurately derive connection between specification and functionality**: Traditional and formal software development approaches focus on specification while Agile methods center their attention on implementation details [23, 55, 66, 68, 81]. However, a common problem that arises while creating these systems is when establishing the relationship between these design components and code modules [23, 24].

   EXMPLRAD includes attributes to help model the parent-child relationship between specification and functional code components. EXMPLRAD also uses the Complete and Unique verification to verify the presence of hidden dependencies which could later cause errors in the system.

1.2 An Overview of the Proposed Solution

Numerous entities such as requirement artifacts, architectural overviews, high-level design documents and lower level implementation details are generated during the project development life-cycle. The current problem is that these details are fairly disconnected from each other which may result in specifications being missed out [24]. Also, no adequate mechanism exists for indexing and retrieving these documents, which can result in unforeseen problems while implementing new enhancements in the system. For solving these problems, we propose to aid project stakeholders with tools for visualizing the relationships between various model components, and automatically finding the missing relationships. These tools must also help in storing project related artifacts in a user-friendly way and provide justification for the correctness of the model [25].

Different software modeling frameworks described in Section 2.2 require different levels of documentation. Even though Agile followers prescribe the "No documentation" rule, this is a recipe for disaster; so some amount of minimal documentation is needed. The "No doc-

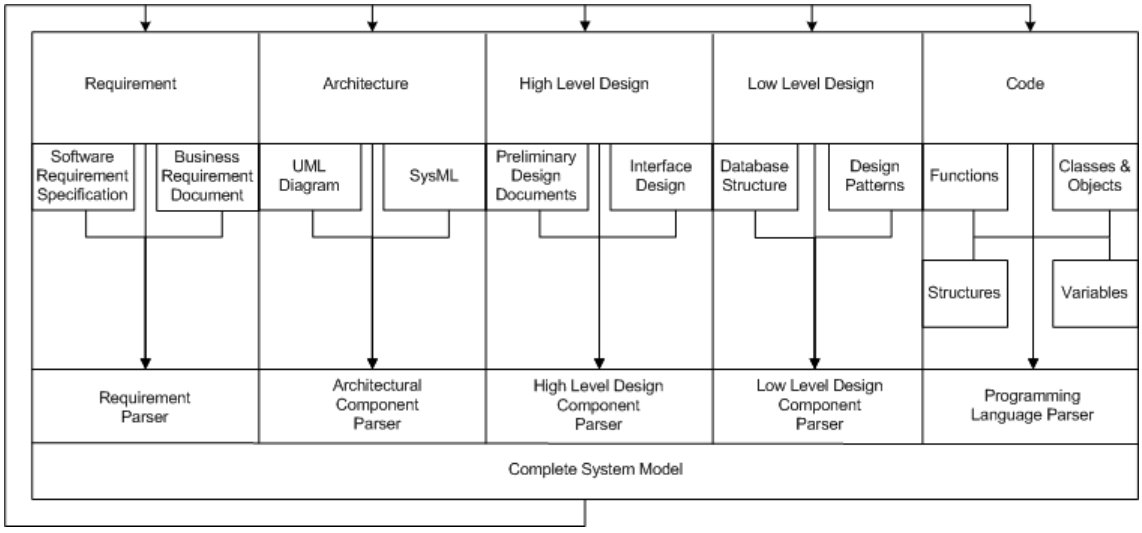


Figure 1.1: Project Artifacts Needed to Generate a System Model

umentation" rule arose because the documentation artifacts created were rarely used during the software evolution period [66]. Our approach combines the documentation methodology with the coding methodology. As time has passed on, the focus has shifted from gathering requirements in requirement specification phase to merging the requirement specification with the coding and testing phase [66, 81]. Our methodology does exactly this and combines the documentation and design phases with the coding phase. Similar to coding procedures, all the design ideas are coded and annotated in-line with the source code.

We have created a specification language Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD) for specifying the project artifacts such as requirement specification, architectural model, design documents and guidelines.

We have also created an expert system that has the ability to analyze this language along with C language constructs. This system has been named *SyModEx2* based on it predecessor *SyModEx* created by Conte de Leon et.al. [23, 24]. Figure 1.1 illustrates the project artifacts that SyModEx2 uses to create the system model. Our expert tool has the capability to extract information from these existing specifications, expressed in EXMPLRAD. This all encompassing system model can be further used to justify the relationships between the software components present in the system.

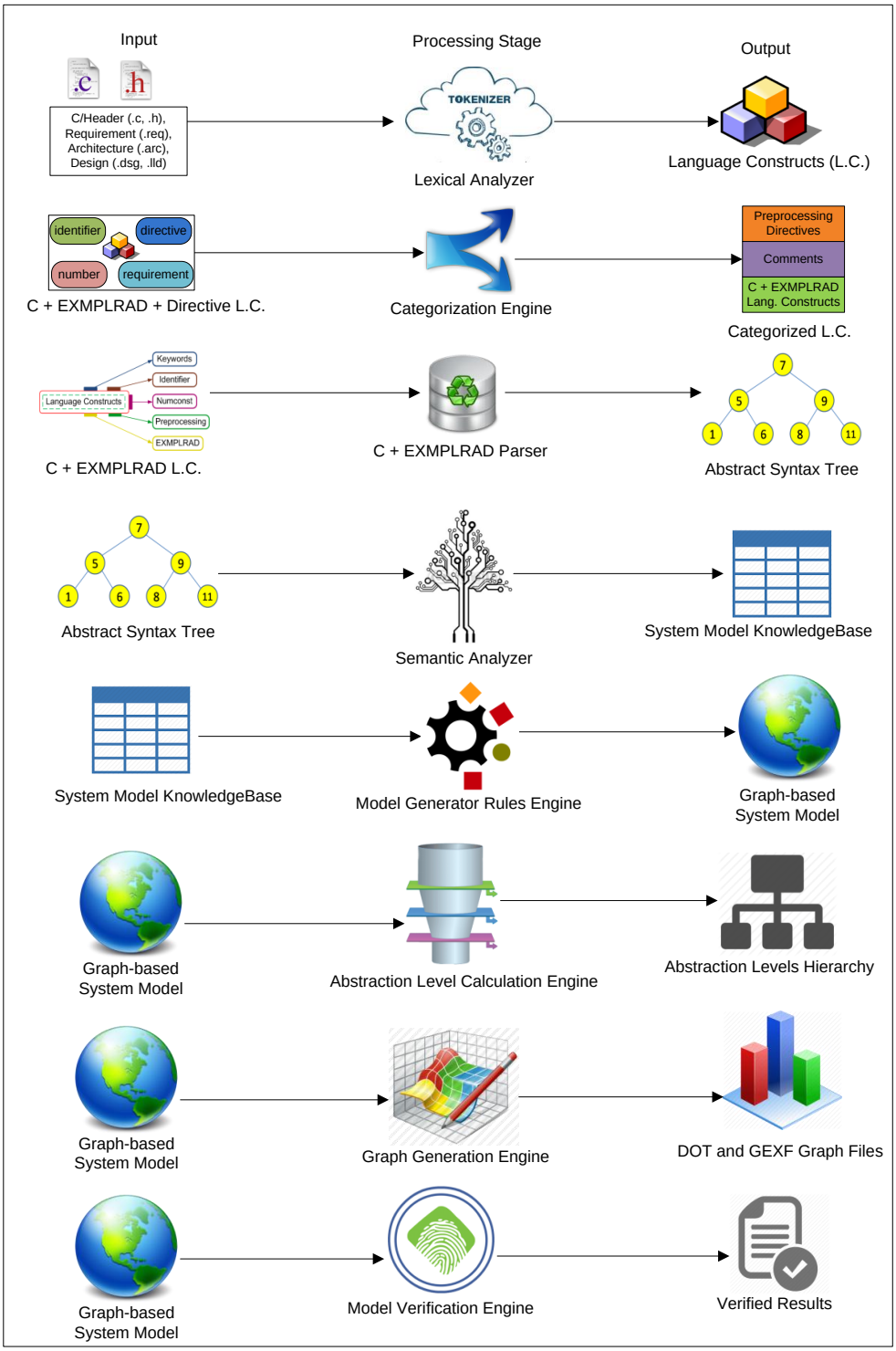


Figure 1.2: Processing Stages of SyModEx2

The information processing mechanism of SyModEx2 expert system is presented in Figure 1.2. In Figure 1.2, the left side is composed of input, which is processed by the processing mechanism shown in middle and the corresponding output generated is shown in the right side.

Firstly, SyModEx2 compiles a list of compatible source files for a given project. Each file is then processed by our **Lexical Analyzer** to output a list of language constructs composed of C, C preprocessing directives and EXMPLRAD literals. Language constructs are language tokens which represent a semantic piece of information. This list is passed to the **Categorization Engine** which separates each of these language constructs according to their category of language. The C and EXMPLRAD language constructs are processed by our **C + EXMPLRAD Parser** to create an Abstract Syntax Tree (AST). The **Semantic Analyzer** analyzes this AST and creates a raw Prolog knowledge-base. This Prolog knowledge-base is processed by our **Model Generator Rules Engine** to output the system model of the overall software system. This system model is used by the **Abstraction Level Calculation Engine** for generating the abstraction level for each component in the system model. The system model is also used by the **Graph Generation Engine** to generate the DOT language [38] and GEXF [47] graph files, which can be processed further for getting visual output. Finally, the system model is processed by the **Model Verification Engine** to check for Complete and Unique verification between system components. Further discussion about each processing mechanism is outlined in Section 1.3.

1.3 Outline of this Thesis

The thesis structure is presented as follows:

Chapter 2 provides background information on software engineering methodologies, functional and non-functional requirements of the system, overview of XSB Prolog, and Definite Clause Grammar (DCG) predicates.

Chapter 3 introduces the need for an extended language, the abstraction levels present in any software system, the syntax and attributes associated with the language along with the advantages of using the techniques and toolset proposed.

Chapter 4 discusses the **Lexical Analyzer** for scanning language constructs in Section 4.2, the **Categorization Engine** for preprocessing source code in Section 4.3, the **C and EXMPLRAD Parser** for generating the AST in Section 4.5, the **Semantic Analyzer** for analyzing the code structure in Section 4.6, the **Model Generator Rules Engine** for generating the system model in Section 4.7, the **Graph Generation Engine** for generating graphs in Section 4.8 and the benefits and limitations of using SyModEx2 in the remaining sections.

Chapter 5 gives an in depth overview of the abstraction level differences present in a system and why manual labeling proves to be a hard task. This chapter also discusses the relationship between different model components, removal of recursive links from the system model and presents an algorithm for the **Abstraction Level Calculation Engine** to automatically generate the abstraction levels for these components in Section 5.8. This chapter also present visualization techniques used to graph the generated abstraction levels along with the benefits and limitations of using this method.

Chapter 6 presents why formal models are important. It further describes the motivation behind the Complete and Unique verification mechanisms and how the **Model Verification Engine** was implemented individually for each of them in Section 6.4 and Section 6.5 respectively. This chapter also describes the visualization strategy used for Complete and Unique verification along with the advantages and disadvantages of using this technique.

Chapter 7 provides an overview of the SEL4 micro-kernel, focusing on the Interprocess Communication (IPC) mechanism. The mapping done from the SEL4-IPC documentation into Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD) and the resulting system model along with results for Complete and Unique verification of components is also presented.

Chapter 8 describes related research work done in the field of traceability for different software methods. It also describes the similarities and differences to our own research work.

Lastly, Chapter 9 presents a summary of findings and the future work that needs to be done.

Chapter 2

Background and Overview

One goal of this project is to provide project stakeholders with a technique and toolset which can help them merge the specification and programming disciplines. This thesis strives to achieve that goal by introducing a new specification language, EXMPLRAD and SyModEx2 an analysis tool for verifying the relationship between crucial system components.

2.1 Chapter Outline

This chapter, firstly gives an overview of various traditional and modern Agile software development methodologies practiced in the software industry. Then it discusses the functional and non-functional properties that characterize a software system. This is followed by a short overview of XSB Prolog and our motivation for using it to develop the SyModEx2 tool. The chapter concludes with a description of the Definite Clause Grammars techniques present in XSB Prolog, which was used for writing the Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD) and C language grammar.

2.2 Traditional, Formal, and Agile Software Development Methods

Software intensive systems in the present world are inherently complex. These systems are deployed in various different sectors such as business, defense, education, energy, health-care, and research.

During creation of software systems, a life-cycle methodology is required for proper development. This methodology is defined in terms of a software engineering model, which is then used to implement the software system [81]. Many different alternative software engineering models have been proposed by the software engineering community. Some of the more traditionally notable ones that were often used to engineer software systems are the Waterfall Model, the Spiral Model and the Iterative Design Model [42, 73, 76, 81, 82].

1. **Traditional Software Development Methods** : The classical waterfall model involves a series of clearly defined process activities during the software development life-cycle. Fundamental tasks involved while creating the software system are requirement definition, system and software design, coding and unit testing, integration and system

testing, operation and maintenance of the developed software [42, 66, 73, 74, 76, 81, 82]. Some major criticisms for the waterfall model are made in terms of non-adaptability to frequently changing requirements, reduced number of testing iterations, inability of the developed software to fully match it's functional specification, large number of project artifacts created which were rarely used later in project lifetime or may be considered as waste as they are not part of project deliverable [66, 78, 81].

The spiral model for software development is a risk-driven process model for engineering software projects [42, 73, 78, 81, 82]. The spiral model dynamically guides the software design team to adopt various process modeling patterns such as waterfall, evolutionary prototyping or incremental development based on different risk patterns identified during requirement evaluation [73, 81]. In a sense, this software engineering model self-adapts according the risk scenario encountered during software engineering.

The more widely adopted iterative design model interleaves between the requirement specification, code development and testing [73, 78, 81]. In each iteration, we get an incremental version of the software which has more added features than it's previous counterparts. Each build of software released during this phase is known as an "incremental" build [73, 81]. Similar to classical waterfall model, it goes through the requirement gathering, design, coding and deployment but the phase lifetime is shorter than of actual waterfall model [82]. Each software build is termed as a "prototype" which has some added functionality than it's previous software version [73, 81].

2. **Formal Software Design Methods** : Formal methods are mathematically rigorous techniques which are used for design, development, and verification of software intensive applications [14, 69]. Formal methods are written in well-formed mathematical statements and formal verifications are deductions performed on these statements using inference logic [69]. Formal methods are important because they provide a means for examining the whole state space of a system model which cannot be accomplished by testing alone.

   Mistakes made in high assurance and critical software systems can have catastrophic consequences [23, 24, 58, 90]. Safety and security are a major concern while creating such systems. Hence, for the purpose of minimizing such critical errors, formal methods are the preferred approach.

Using formal methods for designing complex software systems is labor intensive and thus an expensive endeavor [14]. Therefore, it is infeasible to check each and every component relationship in great detail. Thus, project stakeholders need to prioritize on the behavior exhibited by crucial system components [14].

Formal methods are used for formal specification of requirements, code generation, assertion checking for selected code components, human directed or automated theorem proving mechanisms [69]. Research endeavors based on purely formal approaches have reported that software developed using formal approaches are highly dependable and reliable which makes using these approaches more worthwhile [23–25].

3. **Agile Software Development Methods** : Agile methods tend to focus on fast paced and reliable software development practices [21, 41, 66, 74, 78, 81]. The "buzz" word in today's software development practice is using one of the "Agile" methods [66, 74, 78]. Some of the major principles introduced by Agile software development methodology's Manifesto are [41, 66]:

   (a) Focus placed centrally on the customer.

   (b) Self-organizing teams.

   (c) Software development carried out at a sustainable pace.

   (d) Develop only needed software components (code and unit test cases).

   (e) Accept changes during the development iteration.

   (f) Iteratively develop components and freeze the requirements during coding.

   (g) Create test case for each scenario and do not move to other development cycle before fixing all issues in this current software build.

   (h) Requirements are depicted by scenarios or user stories.

   Agile methods can be broadly categorized into four different variants namely Lean Software, Extreme Programming, Scrum, and Crystal [66, 81]. Lean software is focused on reducing waste while designing a software system [66]. "Waste" in Lean is defined as anything that is not delivered to the consumer. It makes certain that the software project strives to achieve what matters most to the customer and not diverge into any artifacts which can act as a distraction and make the development team stray from it original goal [81].

One of the original agile approaches was the Extreme Programming technique introduced in the late nineteen-nineties [10]. This methodology is based on the idea of incrementally adding functionality to the software and then if needed refactoring it for simplicity [21, 66, 81]. This process model is generally followed by a small group of software developers who work in pairs and maintain a close connection with the customer. The main guiding principle behind extreme programming is take the good practices in software engineering and use them to the extreme point [21, 66]. For example, if code review is a good approach, it's extreme would be to program in pairs so that code is constantly reviewed and checked for errors. Some essential techniques described in extreme programming are user stories, code refactoring, open workspaces, pair programming, test driven development, and continuous integration [66].

Scrum is by far the most popular methodology in the Agile paradigm [66, 78]. Scrum focuses more on the organizational aspects of software engineering and borrows principles from extreme programming for developing the specific software components. The main idea that Scrum focuses on is to freeze requirement specifications during short software development iterations. Change is inevitable during development but accepting changes during an iteration can have disruptive side-effects [78]. Scrum addresses this problem by imposing a rule to accept the change without disrupting the current iteration which is often termed as "closed-window rule" [66]. Any priority items can then be addressed in the next "sprint" because the development sprint is fairly short. Some common practices followed by scrum agilists are sprint planning, closed-window rule, tasks decomposed as event scenarios or "user stories", daily fifteen minute *stand-up* meetings to discuss project status, task board and burndown chart to access speed of project execution, sprint review for an overview on the good and bad that happened during the sprint [78].

Crystal refers to an array of methods where the projects are characterized according to their criticality and size [20, 66]. Crystal focuses on crystal clear interaction between the team members. Its favors open communication techniques such as open office layout for driving down costs and helps solve lack of communication problems [20]. Crystal relies on principles such as clear communication, frequent product delivery, reflective improvement, personal safety for sustainable project deadlines, focus on single task, easy and assured access to expert team members [66]. Similar to the Lean method, Crystal doesn't guide through the steps that need to be followed while designing the software. It is more of a

combination of good software development guidelines which are to be considered closely when designing the product [66].

2.3 Functional and Non-Functional Properties of a Software System

The functionalities provided by a system are defined by its functional requirements [8, 61]. Functional requirement define the functions of a working system and its elements. It is defined by a set of inputs, the processing mechanism (as inherent in a state machine) and a set of outputs. Functional requirements can be technical details such as calculations, processing tasks and other specific service which characterize what a system is supposed to do. Sommerville and Kotonya [82] define functional requirements as:

> *"These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements also explicitly state what the system should not do."* [82].

Chung et.al define non-functional requirements as:

> *"In software system engineering, a software requirement that describes not what the software will do, but how the software will do it, for example, software performance requirements, software external interface requirements, design constraints, and software quality attributes. Non-functional requirements are difficult to test; therefore, they are usually evaluated subjectively."* [15].

Security and safety are some of the important non-functional requirements in highly complex critical software systems [48]. Somerville also states that

> *"These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services."* [82].

Dependability properties which can also be referred to as the non-functional requirements of highly critical computing systems have some important "emergent properties" such as safety and security [8]. Again, this leads us to the reasoning that to achieve these desirable properties a system needs to be correct. Thus, we can infer a software model is correctly implemented iff:

1. No other behaviors apart from the ones mentioned in the specification are possible.
2. The functionality provided and inferred from the spec has been correctly implemented.
3. The requirement and the given constraints are complete and nothing more is required to implement the system correctly.

A system, whether small or large, comprises of manageable building blocks or sub-components. In a software program these software components are namely called functions or procedures or methods. In a project, when this work breakdown is performed, each entity is called a task [81]. Similarly, while decomposing a system, we find that a system is composed of modules and sub-modules integrated properly together to process some useful information. All these work components are collectively referenced as *work products* by Software Engineering Institute (SEI) [87].

Conte de Leon and Alves-Foss [24] define work product sections as independent unit of work product which have some semantic meaning associated with it. The ability to describe the relationship between different work product sections is referred to as traceability of work product sections [24]. In this thesis, the work product sections are referred to as "model component". The model discussed in this thesis gives an approach to represent different work product sections and also define the relationship between them. Making the use of formal traceability relations we can find the discrepancies that are introduced when designing a complex system and make necessary amends accordingly.

2.4 An Overview of XSB Prolog

Prolog is a logic-based language with support for knowledge representation. Officially, the first version of Prolog was developed at University of Marseilles, France in the early 1970s [84]. It was designed as a tool for PROgramming in LOgic. The programmer provides the facts and defines rules for reasoning with these known facts. This information is then used by a Prolog engine to decide whether a proposed fact can be logically derived from the knowledge-base [13, 19, 89]. Hence, Prolog helps us discover or verify new facts or conclusions from the provided old ones.

XSB Prolog is one of the available Prolog platforms and is currently being developed at Stony Brook University [77, 85, 86, 89]. XSB combines features of classic Prolog with tabling [77, 85, 86, 89]. The idea behind tabling is that the Prolog engine keeps track of all the answers returned and if a predicate call is made again, the engine will use the previously

computed answer to satisfy the query. The tabling feature is also referred to as lemmatization or memoization [86]. XSB Prolog is an industrial strength Prolog system for Linux and Windows platforms. XSB Prolog implements some novel features such as full SLG resolution or tabled evaluation, fully multi-threaded engine with thread-shared static code, constraint handling for tabled programs, interfaces to various popular languages such as C, Java, Perl and support for Higher-Order Logic (HiLog) programming [77].

XSB Prolog provides support for primitive data types such as integers, floating point numbers, atoms, variables, compound terms, and lists. SyModEx2 was developed in XSB Prolog because it has tabled resolution of predicates which makes it faster to derive answers from the facts. Built in libraries related to ISO-style file handling, list manipulation, string refactoring and task management were also used.

XSB Prolog also made it easier to express the C language grammar in it's BNF format without having to rewrite for any left or right recursive rules [89]. XSB Prolog's tabling mechanism ensures that the predicates are traversed deterministically avoiding any infinite loops. If we had used any other Prolog we would have to refactor the grammar and some other predicates; while XSB Prolog supports them in their close to Backus Naur Form (BNF).

2.5 Definite Clause Grammar for Expressing Grammar Rules in XSB Prolog

Definite Clause Grammars (DCGs) are an extension of context free grammars (CFGs) which are used to describe formal and natural languages. The procedural meaning of a grammar rule is that it takes in an input sequence of tokens, matches some portion of that list with the appropriate grammar rule, and produces the remaining list of tokens which are matched similarly.

An example DCG rule

```
a(Y) --> b(Y).
```

is expanded as

```
a(Y,Xi,Xo) :- b(Y,Xi,Xo).
```

Here, `Y` is the input to the DCG rule, `Xi` is the input list of terminals which needs to parsed and `Xo` is the output list that results are partial processing of `Xi`.

Listing 2.1: DCG Fragment for C99 Grammar in XSB Prolog

```
1
2 enumerator(enumeratorDef(identifier(E),CE)) -->
3     [identifier(E,_)], [assign], constantExpression(CE).
```

Using Prolog to evaluate DCGs for parsing is termed as *"recursive descent parsing"*. Even though recursive descent algorithms are easy to implement, they do suffer from a number of problems such as:

1. Time taken to parse is exponential to size of input [11, 65]
2. May not terminate for certain CFGs [11, 65]

By using XSB Prolog's inherent tabling features we overcame both of these shortcomings. This is the reason why we used XSB Prolog to create a parser for the C99 grammar.

A sample DCG fragment depicting a part of the C99 Grammar is presented in Listing 2.1. In Listing 2.1, the enumerator predicate on the right hand side is one of the non-terminals in the given grammar rule. The `enumeratorDef(identifier(E),CE)` is a structure which is appended to the Parse tree if the left hand side has been successfully satisfied. Here, `E` and `CE` are Prolog variables which will be further substituted. In this particular instance, the variable E will be replaced by the `identifier(E,_)`'s value. Here, both `[identifier(E,_)]` and `[assign]` are terminals. The `constantExpression(CE)` is a non-terminal which will be successively expanded by XSB Prolog's engine.

Chapter 3

Contribution 1: EXMPLRAD : Extended Language for System Modeling and Implementation

Designing a software system from scratch is a complicated process. During the process of software development it is plausible that this process may have some discrepancies. An emerging area of concern is the proper linking and archiving of generated project specification documents. From the traceability viewpoint, keeping project specifications updated and creating links between the specification and code components can be very beneficial. Proper traceability mechanisms will further ensure that the system constraints have been properly fulfilled. This thesis is a step forward towards reaching this goal. Our solution is to create a specification language which will help us relate software specifications with source code entities. Performing analysis operations on the formal model created by referencing this specification is yet another task we undertake in this thesis. This chapter describes our approach for creating Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD).

3.1 Chapter Outline

This chapter presents the EXMPLRAD specification language, which can be used to describe the properties of a system. This chapter also presents the hierarchical foundation that makes up a system model with it's corresponding properties. This chapter also describes five general abstraction levels that are present in any system model and how they associate with each other.

Subsequently, the format for providing the specifications in EXMPLRAD is discussed. We describe our technique which will help project stakeholders specify any type of requirements using EXMPLRAD. The general markup and format is presented along with their corresponding attributes and meanings. The meaning and importance of these fields to create the Prolog knowledge-base is presented as well.

Finally, in this chapter, a list of advantages that is a result of using such a language is presented and how their use will serve for better mapping of traceability links is shown. A list of limitations are also compiled at the end of the chapter.

3.2 The Need for an Extended Language

A common problem present while designing complex systems today is the ability to properly include and enumerate the project related documents that are generated during various stages in the product life cycle [18, 23]. As discussed earlier in Chapter 1, even though there are numerous documents that are created during the project's lifespan the proper management and review of these documents is always a never-ending problem [63, 67]. Another common problem that developers usually face is the fact that they may not have an in-depth knowledge related to the system [25]. There are numerous reasons for this, however, a common one being that the developer working on the system might be new to the job and have very little knowledge about the complete system. In this case, it is very easy to come up with a scenario in which the developer makes a mistake because he doesn't have a complete understanding of the system [55, 79, 81].

For solving this problem, we propose a language, Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD) which aims to make it easier for the different product stakeholders to write, organize, and consolidate all the product related information in one place. We can even further this approach by providing the users with link fields to map the corresponding component with a designated sub-module's detailed description which may contain implementation details as well as module dependency information. With the help of one common language users can then model the whole system and perform a full analysis related to the inner workings of the system, perform various data mining operations which gives a whole picture of the system and automatically generate documentation relevant to both non-technical project owners and technical developers. Even now a portion of these methodologies are being effectively used; a common example being comment techniques used in Java to write about the method purpose and structure which can later be exported as JavaDocs and other similar documentation tools. The proposed approach is a generalization of this useful technique where rather than generating the project documentation we create a knowledge-base that contains full specification and implementation details about the software system.

3.3 General Hierarchical Model of System Abstractions

During product engineering phase, a system is separated into different components according to their abstraction levels [72]. Generally, a requirement is a broad description of what the system

must implement and it's corresponding lower levels contain information regarding how such requirements are implemented and any dependencies that may be within a system [82]. Hence, we can say that these various abstractions are the actual building blocks for making the system. These substantial information blocks provide crucial data to fully understand the system model. This project provides project stakeholders a simple yet powerful platform to actually embed this abstraction levels information so that we can store as well as perform analysis regarding the implementation details of the system.

Abstractions present in the system have been broadly categorized into five levels:

1. **Requirement :** EXMPLRAD can be used to describe about functional requirements. The requirements specification should give information regarding the services which need to be implemented. It should also describe it's dependencies as well as under which category it falls under. The requirement can also have sub requirements which are then separated into different abstractions using our Abstraction level calculation engine described in Chapter 5. The requirements are defined by using the *"requirement"* keyword followed by the requirement name.

2. **Architecture :** The architecture hierarchy describes the high level architectural overview of the system. It serves as a guideline which should be followed by the developers to develop the system. It gives a formal structural representation of the system and discusses various behaviors that may occur within a model. The architectural components present in the model are elaborated using the *"architecturalComponent"* keyword followed by the name of the component.

3. **High Level Design :** The high level design abstraction gives the high level description of the registered component. It gives some idea of how the system will be implemented without going into details how modules are made up in themselves. High level design descriptions describe various interactions that would occur in a system. A simple example of this would be a Data Flow Diagram which shows how data flows through the system from one component to the other component. The high level design components are written using the *"designComponent"* keyword followed by the component's name.

4. **Low Level Design :** Low level design is similar to the high level design abstraction. The only difference that occurs between these components is the level of detail encompassed

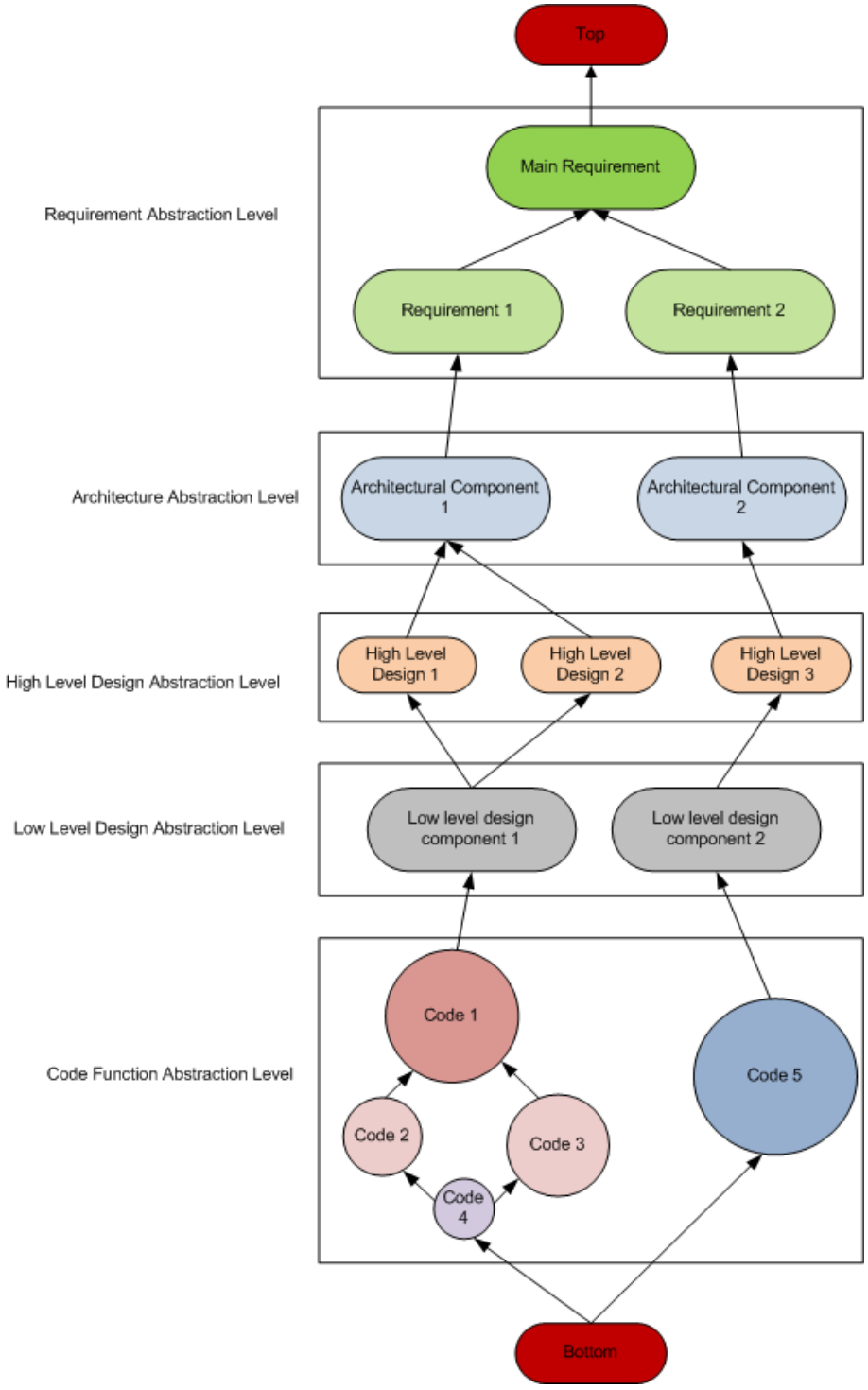


Figure 3.1: A Sample Software System Example

by them. The low level design describes the actual components which interact to form the overall system. They provide a connection between the theoretical model and the

practical implementation that occurs within a system. The component is introduced using the *"lowLevelDesignComponent"* keyword followed by the name of the component.

5. **Code :** The code level abstraction depicts the call graph relationship that occurs within the functions present in the system. This abstraction gives the implementation outline of the project model. It helps give developers an idea of modularity and call relationships between different functions. This technique also helps users identify the functions which reside at the higher level and those which are the basic building blocks for the code. Hence, this gives the most granular view of the system. The keyword *"codeFunction"* is used to provide extra information and attach analysis checkpoints to a particular function. Similar to previous components it is followed by a name which is the actual function name being processed. Thus, the keyword provides important meta data regarding the method which are not necessarily present in the function definition.

Figure 3.1 illustrates an example system. The *Top* and *Bottom* components are used to verify the Unique and Complete formal constraints between two system components, explained in detail in Chapter 6. The size of the bubble covered correspond to the size of code the method signature contains. The primary specification, the *MainRequirement*, is connected to the virtual *Top* component. As seen in Figure 3.1, the requirements abstraction level consists of one *MainRequirement* and two sub requirements, *Requirement1* and *Requirement2* respectively. The requirements branch out to their corresponding architectural designs. The first architectural component *ArchitecturalComponent1* comprises of two design alternatives, *HighLevelDesign1* and *HighLevelDesign2* respectively. The second architectural component *AchitecturalComponent2* consists of one design alternative *HighLevelDesign3*. The lowest abstraction levels are the *LowLevelDesignComponent1* and *LowLevelDesignComponent2* which are connected to the *Code1* and *Code5* method signatures. Finally, the bottom most code level abstractions are connected to the virtual *Bottom* component.

3.4 Language Syntax for Specification of Abstractions in EXMPLRAD

Our goal for developing a new language has been to facilitate the practice of including the project specification entities such as requirements, architecture, high level design, and low level design and merge them together with the actual code. We developed a simple approach to achieve this goal by creating a language on top of the C grammar with minimal side effects to the C99 grammar. The language structure has been inspired from many already present

languages namely JavaDoc Annotations, JSON and PHP.

Listing 3.1: An Example of the Proposed EXMPLRAD Model

```
/*xpp

requirement MainRequirement{
  ID: "req01"
  Type: "SystemReq"
  Description: "All requirements must implement this req"
}

requirement Requirement1{
  ID: "req02"
  Parent: "MainRequirement"
}

requirement Requirement2{
  ID: "req03"
  Parent: "MainRequirement"
}

architecturalComponent ArchitecturalComponent1{
  ID: "arch01"
  Parent: "Requirement1"
  Description: "A sample architectural component"
}

architecturalComponent ArchitecturalComponent2{
  ID: "arch02"
  Parent: "Requirement2"
}

designComponent HighLevelDesign1{
  ID: "design01"
  Parent: "ArchitecturalComponent1"
  Description: "a sample high level design"
}

designComponent HighLevelDesign2{
  ID: "design02"
  Parent: "ArchitecturalComponent1"
}

designComponent HighLevelDesign3{
  ID: "design03"
  Parent: "ArchitecturalComponent2"
}

lowLevelDesignComponent LowLevelDesignComponent1{
  ID: "lowLevelDesign01"
  Type: "LowLevelDesign"
  Parent: "HighLevelDesign1"
  Parent: "HighLevelDesign2"
  Description: "A sample low level design"
  uniquelyImplements: "Requirement1"
  ImplementorFunction: "Code1"
}

lowLevelDesignComponent LowLevelDesignComponent2{
  ID: "lowLevelDesign02"
```

```
58    Type: "LowLevelDesign"
59    Parent: "HighLevelDesign3"
60    ImplementorFunction: "Code5"
61    completelyImplements: "HighLevelDesign3"
62  }
63
64  xpp*/
65
66  void Code1(){
67    Code2();
68    Code3();
69  }
70
71  void Code2(){
72    Code4();
73  }
74
75  void Code3(){
76    Code4();
77  }
78
79  void Code4(){
80  }
81
82  void Code5(){
83    return;
84  }
```

An example of the proposed EXMPLRAD model for the software system illustrated in Figure 3.1, is presented in Listing 3.1. Listing 3.1 shows the syntactic structure of EXMPLRAD. This system model presented in Listing 3.1 is continually referenced in our thesis for the sake of understanding.

In EXMPLRAD, users can write component descriptions and other related metadata in different files or in the same file ending with extensions such as *.c, *.lld, *.arc, *.dsn, *.req . The language structure for EXMPLRAD is fairly similar to JSON where the field and corresponding field values are separated using a colon. Users can distinguish between two fields by either giving a space after the field value of by providing a new line to separate out the sections similar to the Python language. Similar to PHP's **"<?php "** and **"?>"**,each of these small "definitions" must occur between the **"/*xpp"** and **"xpp*/"** tags. The advantage of this structure is that they can be embedded as C comments in the project code.

3.5 Attributes for Specifying Relationships in EXMPLRAD

EXMPLRAD is composed of a number of different attributes which make it easier for the developers and end-users to provide information about the component in context. We strive to make EXMPLRAD more useful by making it simple and flexible as much as possible.

Simplicity is provided to EXMPLRAD users in terms of:

1. Providing the language users with relevant and clearly demarcated field names.
2. Avoiding long and confusing descriptions regarding each field.

Flexibility is provided to EXMPLRAD users in terms of:

1. Order of the fields doesn't matter nor does the actual written order of the component descriptions.
2. Each provided field is not mandatory except for `ID` providing the ability to *"addin"* the meta-data after the field has become relevant enough or is available for the client.

A short description regarding each attribute in EXMPLRAD is given below.

1. **ComponentName :** The *"ComponentName"* attribute defines the given name of the component. For the specification level model components, it is the name of the component that is described after their corresponding component class. For the implementation model i.e. the code portion these component names correspond to the actual function names used in the project. Currently, they need to be unique and are used to perform the analysis operation using SyModEx2.
2. **ID :** The *"ID"* of a component in the overall system is a unique identifier in the system model.
3. **Description :** Given *"Description"* attribute is used to give a detailed description about the component. It should provide all the details that are currently known about the component. This attribute serves as a documentation about the component in context.
4. **Parent :** The *"Parent"* attribute is used to indicate the "Implements" relationships between two distinct and separate abstraction level components. This relationship provides the answer to the question "Which component implements which other component ?". The parent attribute is used to generate the traceability links for the system model.
5. **Type :** Attribute *"Type"* describes the type of component. It relates to the nature or level of the component. A simple example of this would be for a project, there is a MainRequirement of type *"MainReq"* and there can be two sub requirements such

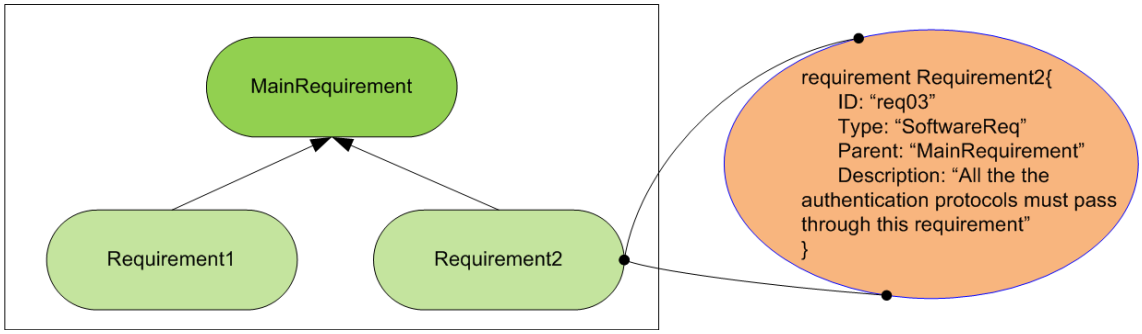


Figure 3.2: A Zoomed In View of Requirement2 (adapted from Figure 3.1)

as *“HardwareRequirement”* and *“SoftwareRequirement”* of type hardware and software respectively.

6. **completelyImplements :** This attribute is used to indicate that the Complete implementation analysis needs to be performed between this component and the referred component. More information relating to Complete implementation can be found in Section 6.3.

7. **uniquelyImplements :** This attribute is used to indicate that the Unique implementation analysis is to be performed between this component and the referred component. More information relating to Unique implementation can be found in Section 6.3.

8. **ImplementorFunction :** This attribute is used to indicate a link between a low level design component and a functional C code function. The link is established based on name of selected implementor component.

A zoomed-in view of the *Requirement2* work product section (wps) is shown in Figure 3.2. The figure shows how each wps has a one to one mapping with a wps definition. It also shows how each wps definition contains important information regarding the wps. A collection of all this significant data is used to create a knowledge-base about the system.

3.6 Advantages of using EXMPLRAD

The advantages of using EXMPLRAD are enumerated below:

1. Give a common platform to describe the system and generate a holistic model of the system.

2. All levels of project stakeholders can be involved in creating a system's model. This combined effort would foster co-ordination and increased product knowledge.

3. Would help new project implementors by providing reference of how a higher level and lower level components were developed and how they would interact with each other.

4. Attributes are clearly demarcated in their definitions which would avoid confusion as well as duplication.

5. Would promote a culture of documentation in the organization and could be used to create project documentation artifacts.

6. Easily extensible and more attributes can be added as per the needs with different features.

7. Ability to perform formal analysis on the generated model which has been written in EXMPLRAD.

3.7 Scope and Limitations of EXMPLRAD

The objective of designing EXMPLRAD is for expressing system specifications as part of the project code. Our rationale is that by using this technique it would be easier for project stakeholders to include and change the specifications as the system undergoes changes. When the system goes through any requirement changes then the system model is automatically changed as well. Also, EXMPLRAD can be used to create a formal system model which is used for the Unique and Complete verification discussed in Chapter 6.

Even though EXMPLRAD does try to be a simple and flexible language. We cannot determine it's usability without performing user acceptance experiments.

Chapter 4

Contribution 2: SyModEx2 : A System Specification and Source Code Analysis Toolset

The previous chapters have discussed why developers need a language which can outline specifications without performing any changes to the actual code structure. However, developers also need an analysis tool which can process EXMPLRAD and develop a complete system model. For performing a level of verification, developers need a tool which can process this model and check for any missing dependencies between important system components. This chapter discusses our expert tool SyModEx2.

4.1 Chapter Outline

The second contribution of this thesis is a toolset which can be used to find missing dependencies that can occur between artifacts in software intensive systems. This chapter, discusses the specific tools used for creating SyModEx2 along with the information processing mechanism used to create this analysis tool.

Firstly, this chapter describes the lexical analysis procedure in SyModEx2. Then it moves on to describe the parsing technique used for parsing the language constructs list and generating the Abstract Syntax Tree (AST) as per the C99 Grammar. This chapter also describe what changes were made to the C99 grammar to accommodate Extended Modeling and Programming Language with Requirements, Architecture and Design Integration (EXMPLRAD) grammar. We discuss the problems we encountered while preprocessing the C source code and the approaches used to overcome these problems.

This chapter also describes how the semantic analysis was carried out over the generated AST and how important and relevant information was mapped from the AST to create a complete knowledge base. This knowledge base is a collection of predicates which are then processed to output the input model which is then further processed to output the complete system model. This output model contains information to perform checking analysis for different components.

Lastly, this chapter discusses advantages and limitations using SyModEx2.

4.2 Scanning Language Constructs and Lookahead: The Lexical Analyzer

Lexical analysis is the very first task any compiler must overtake [65]. In this step, a lexical analyzer reads the actual source code, usually as a continuous stream of characters and differ-

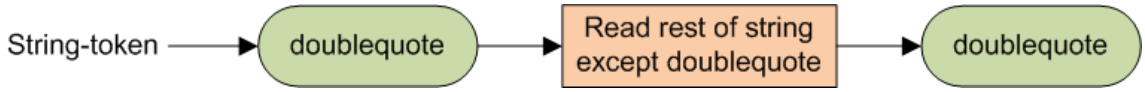


Figure 4.1: Syntax Diagram for String Language Construct

entiates them into meaningful units called tokens or lexemes correspondingly [1, 64, 65, 71, 80]. Also, a lookahead function is used to correctly identify and classify the lexemes. A common method for creating lexical analyzers is to create a finite state machine, either a DFA (Deterministic Finite Automata) or an NFA (Non-deterministic Finite Automata) [1, 65]. Another common method to identify lexemes is by the use of regular expressions. As we know, regular expressions are equivalent to a DFA or an NFA [64].

This thesis references tokens or groups of tokens as language constructs. These language constructs are entities which belong to a certain language grammar. An example of a C preprocessing language construct is the `#include<stdio.h>` which represents a file-include directive.

The *Lexical Analyzer* performs lexical analysis on *.c, *.h, *.req, *.arc, *.dsn, and *.lld file formats. We have created and compiled an external executable in C#, named "makedb.exe", which creates a list of all the files that need to be parsed. This external executable is called using the subprocess creating predicate *spawn_process* defined in XSB Prolog's script writing utilities library [85]. A list of all the files that need to be processed is then compiled in a Prolog database and then loaded statically into the memory for performing the lexical analysis and parsing operation.

SyModEx2 automatically reads each supported file and opens them as an ISO Stream [89]. Using an ISO Stream is important for us because we need to have the facility of peeking to and from through the file for implementing the lookahead functionality [65, 89]. Using XSB Prolog's character peeking facility, the lookahead function is able to correctly predict the type of language construct each character belongs to. This technique helps us segregate between characters and assign them to a proper data type (such as integer, keyword, string, etc.). The read file stream is processed as a list of ASCII character language constructs. After the differentiation of all the characters present in the file is done, we have a list of lexemes which can be processed further for parsing [80]. Figure 4.1 gives the syntax diagram for string identification from input source files.

This technique is very different from traditional compiler building approaches in Prolog. An important reason for this is that Prolog inherently doesn't provide regular expressions. Also

creating a DFA would be a viable option, however the sheer number of cases to handle would be time-consuming and could result in problems. Even though XSB Prolog does provide a built-in lexical analyzer, it is not powerful enough to parse all C constructs [85]. Therefore, we decided to hand-code a lexical analyzer from the ground-up by creating predicates which take the current character and the lookahead character(s) into consideration and categorize them appropriately [80]. Thus, by using this method we successfully generated a list of correctly identified lexemes. An added benefit of this approach is that it makes it easier to debug and pinpoint any problem that occurs during lexical analysis by issuing errors for unidentified characters or undefined lexemes.

4.3 Preprocessing C Source Code: The Categorization Engine

Actual C source code is a mix of two languages: the C language and the preprocessor language [45]. Preprocessing is necessary because it adds facilities which are lacking in the original C language [45]. The C preprocessor enhances expressibility through file inclusion mechanism (`#include`), macros (`#define`) and static conditionals (`#if`, `#ifdef`) to capture variability. The C preprocessor is not aware of C structure and operates only on individual language constructs [45].

Using the C preprocessor, developers can overcome some limitations of C, by introducing added features such as iterators, meta-programming or even source code refactoring [71]. However, this flexibility comes at a steep price: it makes it extremely difficult and tedious to perform style preserving source to source transitions such as refactoring on C code [71]. Any step to combine these two grammars, results in a grammar which is very large, contains ambiguities and is very different from the C grammar [71].

Attempts have been made for a one time processing mechanism which could lexically process and parse the whole source code all at once. Such attempts to combine both the C grammar along with the preprocessor grammar have been with mixed results. Tools such as Yacfe [45] and SuperC [71] show while this merge is possible, it is not fully complete. They mostly report incompleteness in terms of macro evaluation. As macro evaluation is subject to both expansion as well as replacement, it becomes a hard task to integrate it into the C99 grammar rules. Our solution to this problem was to develop the *Categorization Engine* which separates the preprocessing directives and comments from the C and EXMPLRAD source code.

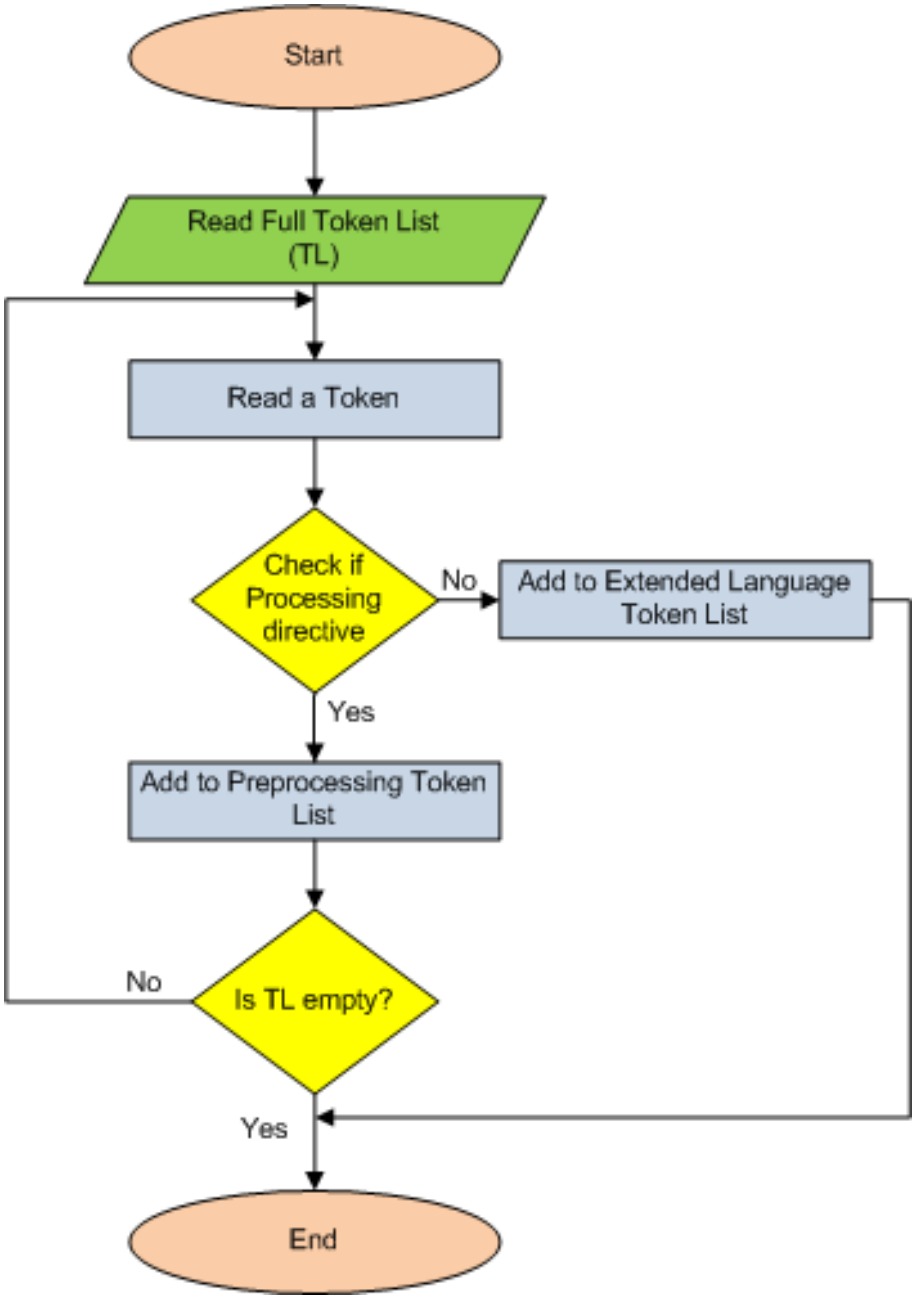


Figure 4.2: Categorization into C and Preprocessing Language Constructs

Looking at the structure of preprocessing directives in C, we can say that they have some very distinct properties which can be used for identification. Some of these properties are outlined below:

1. All C preprocessor directives start with a # sign

2. Each C preprocessing directive must end by a new line character. The only method to

Listing 4.1: SyModEx2 Source for Separating C Code and C Preprocessing Directives

```
1
2  isPredefined(directive(_,_)).
3
4  categorizePreDefineTokens([OriginalTokensH|OriginalTokensT],
       PreDefineIn, PreDefineOut,ELCTokensIn, ELCTokensOut):-
5  (
6  isPredefined(OriginalTokensH)
7  ->
8  (
9    append([OriginalTokensH],PreDefineIn,PreDefine ),
10   categorizePreDefineTokens(OriginalTokensT,PreDefine,
         PreDefineOut,ELCTokensIn, ELCTokensOut)
11 );
12 (
13   append([OriginalTokensH],ELCTokensIn,ELCTokens ),
14   categorizePreDefineTokens(OriginalTokensT,PreDefineIn,
         PreDefineOut,ELCTokens, ELCTokensOut)
15 )
16 ).
17
18 categorizePreDefineTokens([], P,POut, E,EOut):-
19   reverse(P,POut),
20   reverse(E,EOut).
```

extend the preprocessing directives definition is by using a backslash symbol. In this case, the last line is identified as the line ending the new line character.

A simple example is presented below :

```
#define DEBUG_BREAK(a) \
        if ((a)) \
       __asm int 3;
```

3. In each C preprocessing directive, the `#` symbol is followed by a keyword. For example in `"#pragma"` where `pragma` is the keyword.

Using these rules, we created rules which will look at all such cases and map out the preprocessing directives and categorize them as preprocessing language constructs. Removing the `#include` and `#define` preprocessing language constructs should have not effect on the actual source code as they are separated from the source code. The conditional compilation directives `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` and `#endif` pose more of a problem as they can change the actual source code dramatically. However, embedding the C preprocessor grammar alongside the C99 is very difficult [45, 71]. Hence, we have categorized the conditional compilation, diagnostics, line control and pragma preprocessing language constructs and separated them out

Listing 4.2: SyModEx2 Source for Separating C Code and Comments

```
1
2  isComment(comment(_)).
3
4  categorizeCommentTokens([OriginalTokensH|OriginalTokensT],
       CommentsIn, CommentsOut,ELCTokensIn, ELCTokensOut):-
5  (
6  isComment(OriginalTokensH)
7  ->
8  (
9    append([OriginalTokensH],CommentsIn,Comments ),
10   categorizeCommentTokens(OriginalTokensT,Comments,
         CommentsOut,ELCTokensIn, ELCTokensOut)
11 );
12 (
13   append([OriginalTokensH],ELCTokensIn,ELCTokens ),
14   categorizeCommentTokens(OriginalTokensT,CommentsIn,
         CommentsOut,ELCTokens, ELCTokensOut)
15 )
16 ).
17
18 categorizeCommentTokens([], C,COut, E,EOut):-
19   reverse(C,COut),
20   reverse(E,EOut).
```

in a list. This procedure is also depicted in Figure 4.2 for a clearer insight to the *Categorization Engine*.

An excerpt Prolog code for the *Categorization Engine* used while categorizing C preprocessing directives is shown in Listing 4.1. All preprocessing directives are identified as `directive(X,Y)`, where X corresponds to the directive entry and Y corresponds to the line number where the preprocessing directive was encountered. The directive entry contains all the characters which are present in the preprocessor definition. The Prolog predicate *categorizePreDefineTokens* in Line 4 checks if each entry from the list of language constructs, contains the preprocessing directive entry. The checking operation is performed with the help of the predicate *isPredefined*. If the selected language construct is a preprocessing directive it is appended to a preprocessing directive list otherwise the language construct is added to the C language constructs list. At the end of the list, after all the language constructs has been consumed by the *categorizePreDefineTokens* predicate, both the preprocessing language construct list and the C language constructs list is reversed so that the language constructs read first are actually on the top of the list.

Similar predicates for the categorization of comments and C99 language literals has been presented in Listing 4.2. The predicate for categorizing the comments is termed as *catego-*

rizeCommentTokens shown in Line 4. The *isComment* predicate, defined in Line 2, is used to check if a language construct is a comment or not.

4.4 Embedding the EXMPLRAD Grammar into the C99 Grammar

After systematic categorization of the source code using the *Categorization Engine*, the next step is to parse these output lexemes using a combined *C and EXMPLRAD Parser*. The discussion in this section details how we merged the C and EXMPLRAD grammars and created a

Listing 4.3: Excerpt of the C99 Grammar (adapted from C99 Specification [54])

```
1 (6.9) translation-unit:
2   external-declaration
3   translation-unit external-declaration
4
5 (6.9) external-declaration:
6   function-definition
7   declaration
```

Listing 4.4: EXMPLRAD Grammar Integrated with C99 Grammar in BNF

```
1  <translation-unit> ::=
2    <external-declaration>
3    | <translation-unit> <external-declaration>
4
5  <external-declaration> ::=
6    <function-definition>
7    | <declaration>
8    | <extended-lang-definition>
9
10 <extended-lang-definition> ::=
11   "/*xpp" elDef "xpp*/"
12
13 <elDef> ::=
14   <elDefItem>
15   | <elDef> <elDefItem>
16
17 <elDefItem> ::=
18   "requirement" <identifier> "{" <attributeList> "}"
19   | "architecturalComp" <identifier> "{" <attributeList> "}"
20   | "designComp" <identifier> "{" <attributeList> "}"
21   | "lowLevelDesignComp" <identifier> "{" <attributeList> "}"
22   | "codeFunction" <identifier> "{" <attributeList> "}"
23
24 <attributeList> ::=
25   <attributeItem>
26   | <attributeList> <attributeItem>
27
28 <attributeItem> ::=
29   "Description" ":" <stringLiteral>
30   | "Type" ":" <stringLiteral>
31   | "Parent" ":" <stringLiteral>
32   | "uniquelyImplements" ":" <stringLiteral>
33   | "completelyImplements" ":" <stringLiteral>
34   | "ImplementorFunction" ":" <stringLiteral>
```

Listing 4.5: SyModEx2 Source for C99 Grammar with EXMPLRAD Hook using DCG

```
1  program(AST) -->
2      cProgram(AST).
3
4  cProgram(fullCProgram(AST)) -->
5      translationUnit(AST).
6
7  translationUnit(translationUnit(AST)) -->
8      externalDeclaration(Unit),restTranslationUnit(Unit,AST).
9
10 restTranslationUnit(Unit,Units) -->
11     externalDeclaration(Unit1),restTranslationUnit(
           multipleTranslationUnit(Unit,Unit1),Units).
12 restTranslationUnit(Unit,Unit) -->
13     [].
14
15 externalDeclaration(externalDeclaration(Decl)) -->
16     declaration(Decl).
17 externalDeclaration(externalDeclaration(FunctDef)) -->
18     funtionDefinition(FunctDef).
19 externalDeclaration(externalDeclaration(ELDef)) -->
20     extendedLangDefinition(ELDef).
```

unified parsing mechanism. The grammatical structure for EXMPLRAD is very simple and straightforward. As we have described earlier in Section 3.4 the format of the EXMPLRAD grammar closely matches that of the JSON Grammar [28]. The grammar rules for starting the parse operation in the C99 language are outlined in Listing 4.3. The "hook" for adding the entry point for the EXMPLRAD grammar was identified in the higher level of the C grammar. The "hook" point selected for embedding the EXMPLRAD grammar was the *external-declaration* non-terminal presented in Listing 4.3 in Line 5.

After analyzing the C99 grammar, we found that the main entry point for C grammar was the *translation-unit* non-terminal [54]. This grammar rule extends to any function definitions or any other declaration such as variable or function declarations. Due to the small and simple structure of the EXMPLRAD grammar, the most intuitive point to add EXMPLRAD's grammar rules would be at the *external-declaration* level.

This section now describes the details of the grammar for the EXMPLRAD. As discussed in Section 3.4, the language starts with */*xpp* and ends with *xpp*/* tags. Inside these tags users can define one or more system model components. Each model component must have one or more attributes defining the properties of the component. Listing 4.4 shows an excerpt of this extended grammar which integrates C99 with EXMPLRAD grammar in Backus-Naur-Form (BNF).

The SyModEx2 source code developed for the C99 grammar with the EXMPLRAD grammar is shown in Listing 4.5 and Listing 4.6. In the source code in Listing 4.5, `program` predicate is the initial predicate called for parsing and it's parameter `AST` represents the Abstract Syntax Tree generated using XSB Prolog. Similar to the BNF grammar shown in Listing 4.4, the `translationUnit` predicate is recursively made up of `externalDeclaration` and `translationUnit`. The `externalDeclaration` predicate is composed of three other predicates, namely `declaration`, `funtionDefinition` and `extendedLangDefinition`. Similarly, in Listing 4.6, the `extendedLangDefinition` predicate provides the grammar definition for the EXMPLRAD language. The `elDefItem` predicate is used parse the EXMPLRAD entries in the predefined abstraction level. Also, the `statementCC` predicate is used to enumerate the different attributes present in an abstraction level.

Listing 4.6: SyModEx2 Source for EXMPLRAD Grammar using DCG

```
1
2  extendedLangDefinition(ReqDef) -->
3      [divide],[times],[keyWord(xpp)],elDef(ReqDef),[keyWord(
           xpp)],[times],[divide].
4
5  elDef(elDef(ReqDef)) -->
6      elDefItem(R1), restElDefItem(R1,ReqDef).
7
8  restElDefItem(R1,R2) -->
9      elDefItem(R3), restElDefItem(elDef(R1,R3),R2).
10 restElDefItem(R,R) -->
11     [].
12
13 elDefItem(component(identifier(ReqName,N),BlockItemListCC))
       -->
14     [keyWord(requirement)],[identifier(ReqName,N)],[leftcurly
           ],blockItemListCC(BlockItemListCC,'req'),[rightcurly].
15
16 elDefItem(component(identifier(ArchCompName,N),
       BlockItemListCC)) -->
17     [keyWord(architecturalComponent)],[identifier(
           ArchCompName,N)],[leftcurly],blockItemListCC(
           BlockItemListCC,'archComp'),[rightcurly].
18
19 elDefItem(component(identifier(DesignCompName,N),
       BlockItemListCC)) -->
20     [keyWord(designComponent)],[identifier(DesignCompName,N)
           ],[leftcurly],blockItemListCC(BlockItemListCC,'
           designComp'),[rightcurly].
21
22 elDefItem(component(identifier(LowLevelDesignCompName,N),
       BlockItemListCC)) -->
23     [keyWord(lowLevelDesignComponent)],[identifier(
           LowLevelDesignCompName,N)],[leftcurly],blockItemListCC
           (BlockItemListCC,'lowLevelDesignComp'),[rightcurly].
24
25 elDefItem(component(identifier(FunctionName,N),
       BlockItemListCC)) -->
```

```
26      [keyWord(codeFunction)],[identifier(FunctionName,N)],[
           leftcurly],blockItemListCC(BlockItemListCC,'
           codeFunction'),[rightcurly].
27
28  blockItemListCC(blockItemListCC(StatementVarCC),N) -->
29      statementCC(StatementVarCC1,N),restBlockItemListCC(
           StatementVarCC1,StatementVarCC,N).
30
31  restBlockItemListCC(StatementVarCC,StatementVarCCs,N) -->
32      statementCC(StatementVarCC1,N),restBlockItemListCC(
           blockItemListCC(StatementVarCC,StatementVarCC1),
           StatementVarCCs,N).
33  restBlockItemListCC(StatementVarCC,StatementVarCC,_) -->
34      [].
35
36  statementCC(elComp(Prefix,'ID',N,Var),Prefix) -->
37      [identifier('ID',N)],[colon],[stringLiteral(Var)].
38  statementCC(elComp(Prefix,'Description',N,Var),Prefix) -->
39      [identifier('Description',N)],[colon],[stringLiteral(Var)
           ].
40  statementCC(elComp(Prefix,'Parent',N,Var),Prefix) -->
41      [identifier('Parent',N)],[colon],[stringLiteral(Var)].
42  statementCC(elComp(Prefix,'Type',N,Var),Prefix) -->
43      [identifier('Type',N)],[colon],[stringLiteral(Var)].
44  statementCC(elComp(Prefix,'CompletelyImplements',N,Var),
       Prefix) -->
45      [identifier('completelyImplements',N)],[colon],[
           stringLiteral(Var)].
46  statementCC(elComp(Prefix,'UniquelyImplements',N,Var),Prefix)
       -->
47      [identifier('uniquelyImplements',N)],[colon],[
           stringLiteral(Var)].
48  statementCC(elComp(Prefix,'ImplementorFunction',N,Var),Prefix
       ) -->
49      [identifier('ImplementorFunction',N)],[colon],[
           stringLiteral(Var)].
```

4.5 Parsing the Integrated EXMPLRAD and C99 Source Code

C99 definition of *typedef-name* and *IDENTIFIER* language constructs make the C99 grammar context sensitive and usually extra coding effort is required to correctly parse using the C99 grammar [30, 49, 54]. In our case, we do not have to perform any of these checks because of Prolog's non-determinism functionality. The non-determinism functionality ensures that a correct tree is always selected and returned after processing the input language constructs [89]. Hence, we can write the C99 grammar integrated with the EXMPLRAD grammar in it's close to BNF form [54].

Another common problem that occurs while developing parsers is due to the fact that the C99 grammar is left recursive [45, 71]. Since we know that Prolog does depth first search in order to find "solutions", the parser may get stuck in an infinite loop while processing the lexemes. A common solution to solve this problem is found by using the left factoring techniques which

Listing 4.7: Excerpt of Prolog Predicates for the SyModEx2 Semantic Analyzer

```
1
2 functionEncoder(fullCProgram(translationUnit(X)),Name,
     FileName):-
3      functionEncoder(X,Name,FileName).
4
5 functionEncoder(multipleTranslationUnit(X1,X2),Name,FileName)
     :-
6      functionEncoder(X1,Name,FileName),
7      functionEncoder(X2,Name,FileName).
```

have been described in the Compiler Construction Handbook by Kenneth C. Louden [65]. Our approach uses XSB Prolog's tabling mechanism to handle the left recursive grammar rules and hence avoid any infinite recursions [77, 86, 89].

The ambiguities present in the C99 grammar such as the dangling else problem can be resolved using the traditional techniques explained in [1]. However, since our parser is hand-written in XSB Prolog, the non-determinism feature combined with the tabling mechanism takes care of this problem as well. The C99 and the EXMPLRAD grammar rules are represented as Definite Clause Grammar (DCG) rules in Prolog. More information related to use of DCG's in Prolog for expressing grammar rules was presented in Section 2.5.

For the purpose of porting the grammar rules, and to check if no ambiguities were present, we used an all-purpose parsing system known as "Gold Parsing System" [31]. Gold parser generates a DFA and LALR(1) parser based on the language construct recognition scheme used and the grammar rules supplied [31]. The language constructs were recognized using the regular expressions defined in the Lex specification file by Jutta Degener [29]. Similarly, the C grammar rules written by Jutta Degener were derived from the Yacc specification file and cross checked with the ISO C99 grammar specifications [30, 54]. After that, the EXMPLRAD grammar was integrated to the C grammar rules and checked for consistency. These grammar rules were then ported to their corresponding DCG Prolog predicates forms and thus the *C and EXMPLRAD Parser* was created.

After the categorization of the lexemes has occurred, the end result consists of lexemes which are all compatible with the C99 and EXMPLRAD Grammar rules. The output language constructs list is passed from the *Categorization Engine* on to the *C and EXMPLRAD Language Parser*. Each language construct is consumed by the grammar rules defined in XSB Prolog's DCG format and an Abstract Syntax Tree (AST) is recursively generated. The full AST is created when there are no further language constructs left to consume.

Listing 4.8: Example C Source Code Excerpt from Listing 3.1

```
1
2 void Code5(){
3   return;
4 }
```

4.6 Semantic Analysis of Abstract Syntax Tree to Generate the KnowledgeBase

Once the parser generates the AST, the semantic analysis of the source code needs to be performed. In this phase, the Semantic Analyzer mines the needed information out of the AST using rules which are capable of extracting data from the provided AST. These rules are recursive in nature and they help remove a layer of structure for each provided AST structure. Using Prolog predicates the AST can be traversed and search operations can be performed for specific patterns.

An example of such a Prolog predicate is given in Listing 4.7. Here, the predicate `functionEncoder` consists of 3 parameters: the first parameter corresponds to the Abstract Syntax Tree's structure which is matched with the pattern given, the second parameter `Name` is related with the name of the destination file where the analysis results will be written, and the third parameter `FileName` refers to the name of the source file which is used extensively during file manipulation operations. The right hand side of the Prolog predicate recursively calls the predicate passing the result of pattern matching operation performed by the first parameter of the left hand side clause. Hence, during each successful recursive call, one of the outer layers of the AST is removed.

Consider a simple example C source code given in Listing 4.8 extracted from Listing 3.1. The Semantic Analyzer recursively removes this upper level information and searches for corresponding function definitions and calls. The Semantic Analyzer is capable of extracting the function names, function parameters, and the function calls performed by the calling function. Using this data we create a knowledge-base with all the information necessary to generate a system model. During the semantic analysis phase, we also extract the model information from the EXMPLRAD specifications.

4.7 System Model Generation by Analysis of Prolog Knowledge Base: The Model Generator Rules Engine

The compiled Prolog knowledge base contains all of the information needed to create a complete model of the system. The knowledge base is composed of information such as function

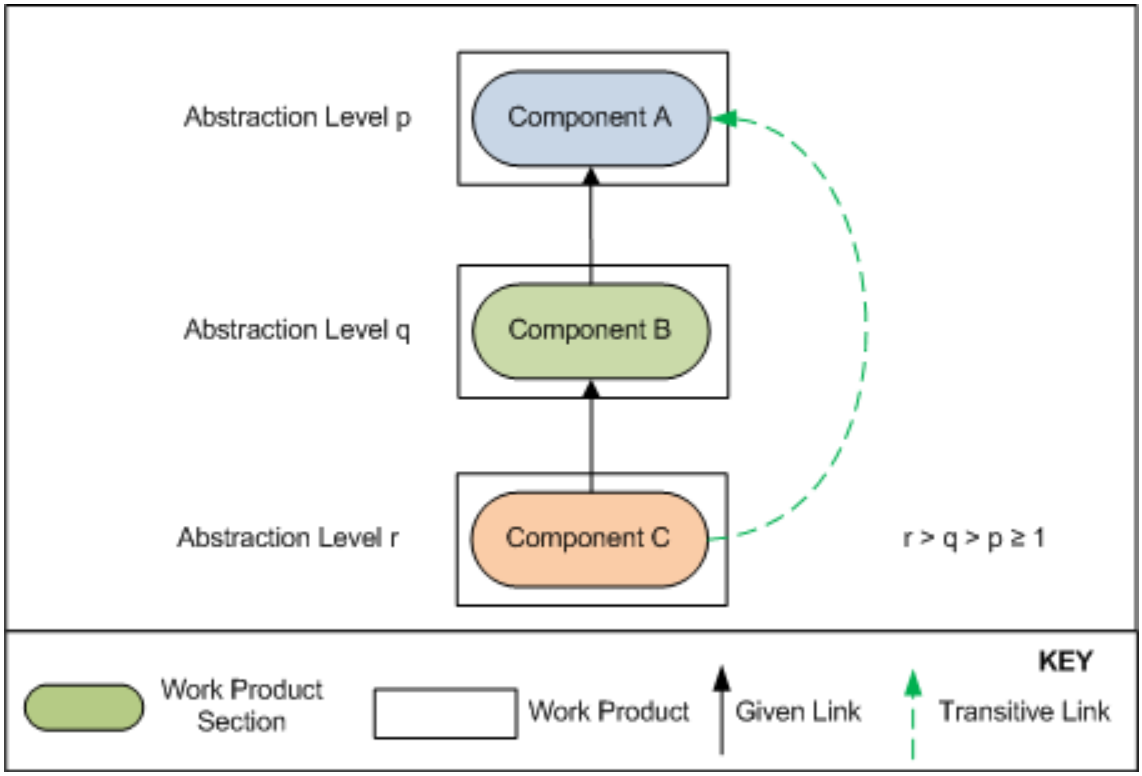


Figure 4.3: Illustration of Presence of Transitive Links (adapted from [24])

definitions, method calls, requirement definitions, component description, design artifacts, and architectural models. This data is loaded dynamically into the XSB deduction engine which is then processed by the *Model Generator Rules Engine*.

The *Model Generator Rules Engine* is used to generate the initial system model which is in common format and which makes the system ready for other types of analysis as will be discussed later in Chapter 6. The first step is to create the nodes from all relevant components. Nodes are structures which represent the actual components or entities in the system such as requirements, architectural components, design components or functions. After all the nodes have been identified, SyModEx2 makes a list of all the links present in the system. The links are extracted from the parent-child relationship between specification level components and C function calls for C code functions. After this has been done, SyModEx2 removes any recursive links present in the system model and then calculate the abstraction levels for each of the nodes present in the model. SyModEx2 removes recursive function calls because they introduce cycles in the model and then the hierarchy of the components is not properly defined. The *Abstraction Level Calculation Engine* is used to derive these levels and will be described further in Section 5.8, *"An Algorithm for the Automated Generation of Abstraction Levels"*.

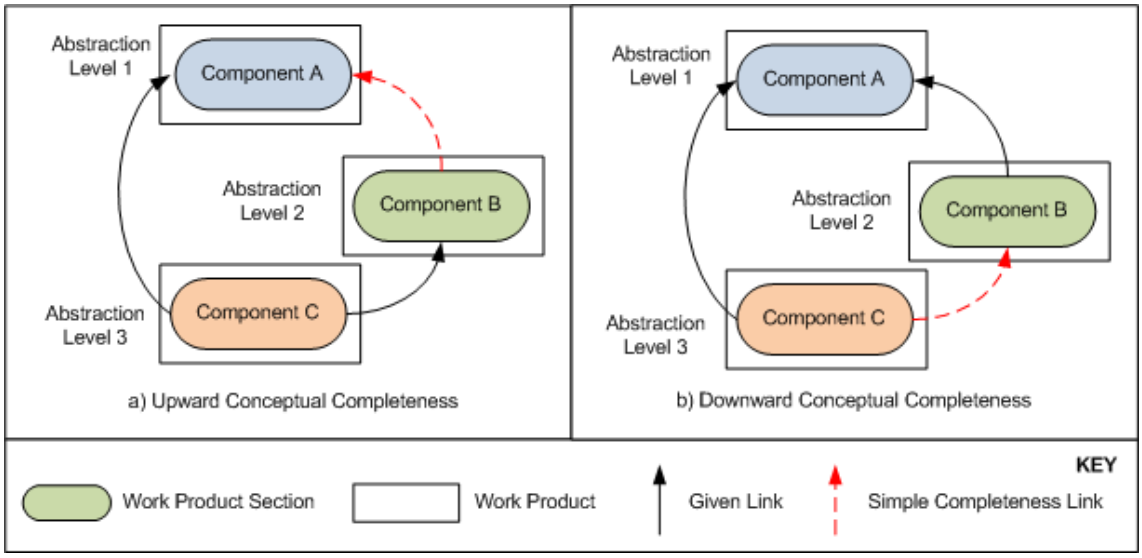


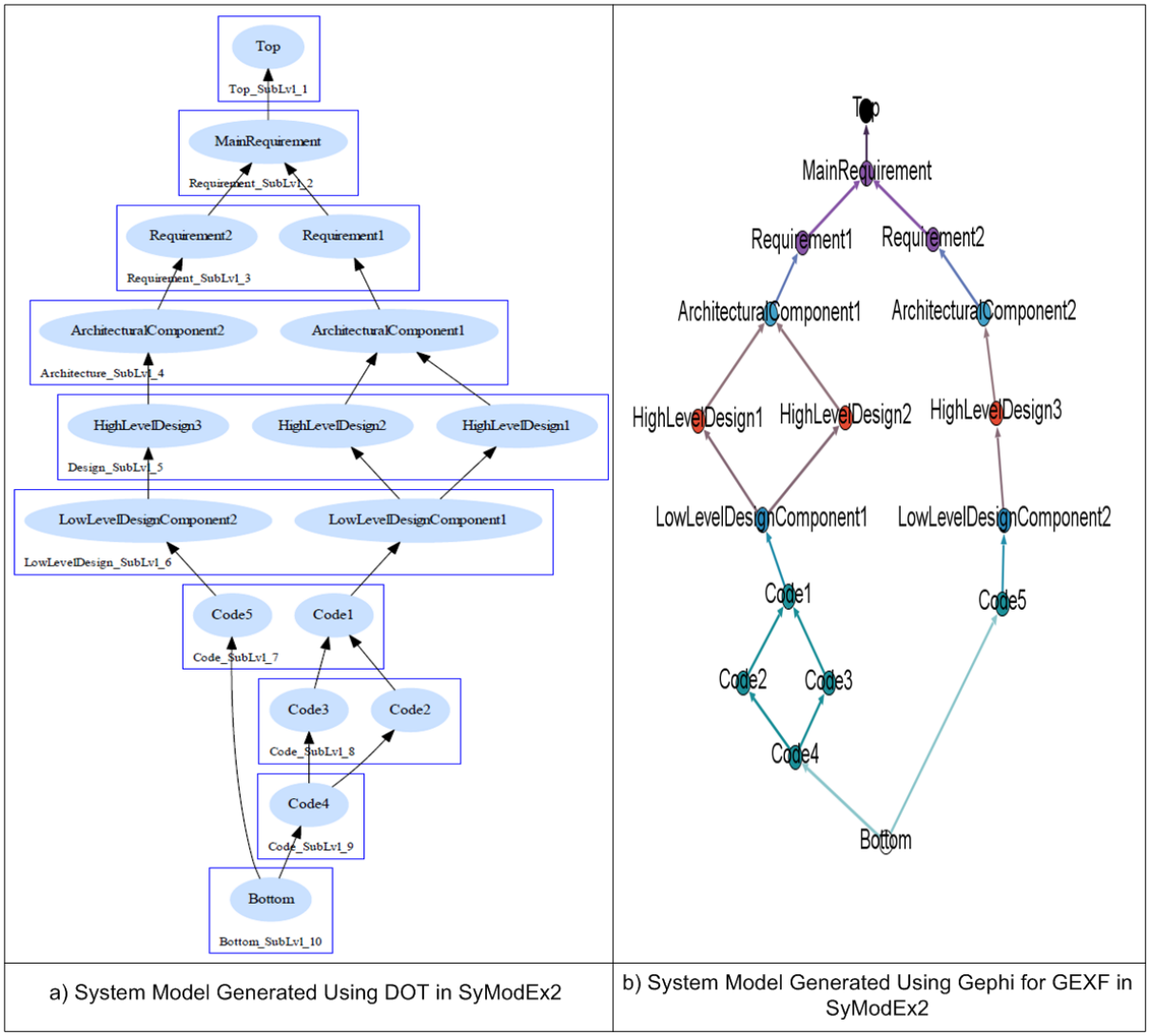
Figure 4.4: Illustration of Presence of SimpleCompleteness Links (adapted from [24])

The input model is then further processed by the *Model Generator Rules Engine* to generate "Transitive" and "SimpleCompleteness" links which can be inferred from the system's "Given" links. These links denote the implementation links present in the model. All these links are represented by the "Implements" attribute in the edges present in the system. "Implements" links are described by Conte de Leon et.al. in [23, 24].

"Transitive" links are those links which can be derived from simple transitive relationships between components. A simple example of this relationship is shown in Figure 4.3. Here, the service provided by *Component C* is used by *Component B* and the service offered by *Component B* is in turn used by *Component A*. Thus, we can conclude that the preliminary service offered by *Component C* is crucial for the working of *Component A*. Hence, a connection exists from *Component C* to *Component A* which is defined by the "transitive" property.

SimpleCompleteness links are generated on the basis of conceptual completeness principle described by Conte de Leon and colleagues in [23, 24]. The conceptual completeness principle can be separated into two cases, namely: *Upward Conceptual Completeness* and *Downward Conceptual Completeness*. Figure 4.4 can be used to describe both of these cases.

Consider three distinct work components *Component A*, *Component B* and *Component C*. For upward conceptual completeness, if there is a link from *Component C* to *Component A* and a link from *Component C* to *Component B* and they are located in distinct abstraction levels then a completeness link is present between *Component B* to *Component A*.



a) System Model Generated Using DOT in SyModEx2

b) System Model Generated Using Gephi for GEXF in SyModEx2

Figure 4.5: Visualization of a System Model using GraphViz DOT Tool and Gephi

Similarly, for downward conceptual completeness illustrated on the right side of Figure 4.4, if there exists a link from *Component C* to *Component A* and a link from *Component B* to *Component A* and they are located in distinct abstraction levels then a completeness link is present from *Component C* to *Component B*.

4.8 Visualization of SyModEx2 System Models: The Graph Generation Engine

SyModEx2 includes built-in capabilities for automated graph generation using the *Graph Generation Engine*. The generated graphs adhere to either the GEXF [47] or the DOT language [44] standard. DOT language is a text-based graph description language which is capable of describing the nodes and their directed or undirected relationships [44]. DOT language is partic-

ularly useful for automatically drawing non-overlapping graphs using the appropriate placement for each node. DOT language can be processed by the GraphViz DOT tool [38] to output a number of standard file formats such as JPG, PNG, SVG, and PDF [44]. We use the DOT language to write the hierarchical relationship between different abstraction levels. We also use the GraphViz DOT tool to show implementation links between different components. Graphs are generated in both grayscale and color formats. We can use the GrapViz DOT tool or Gephi [9, 46] to visualize the graphs.

Another graphing file format we use is GEXF format. GEXF is an XML-based graph language which has the ability to represent a large number of nodes and edges with their corresponding attributes [47]. We used GEXF as an alternative graphing language because of some of DOT language's deficiencies. Some problems with DOT are that it's attributes are not fully recognized by Gephi [46], filtering and selection capabilities are not available and the graphs generated are very bulky or even unreadable in some cases.

Gephi is a graph visualization tool which has the ability to color, make layout changes, rank, cluster and filter the graph [46]. GEXF is a supported language which is supported by Gephi. In Gephi, developers can add plug-ins for coloring and managing layout changes [9]. Gephi also supports filtering of nodes and edges according to a certain attribute or a list of attributes connected by a logical operator [9, 46]. This makes it ideal for visual analysis of large datasets as certain desired subgraphs can be pruned from view. Gephi also has a plug-in for automatically creating an interactive web-viewer called SigmaJS [52] which has the ability to search and list out sets of connected nodes.

Examples of a visual model generated from DOT and Gephi are shown in Figure 4.5. Other graphs related to the hierarchy of abstraction levels and the Complete and Unique verification are presented in their corresponding chapters, Section 5.9 and Section 6.6 respectively.

4.9 SyModEx2 Testing and Results

For the purpose of testing SyModEx2, we ran our tool through the c test files located inside the "gcc-5.2.0/gcc/testsuite/c-c++-common/asan" folder. The folder contained 68 files out of which we had to comment out 39 instance of C Extensions in 19 files. After commenting out the C Extensions, the files were parsed successfully by the SyModEx2 tool. Our overall objective is to parse all the test-cases in the gcc testsuite which will be done in the future.

4.10 Advantages of using SyModEx2

The advantages of using the SyModEx2 toolset are:

1. Enables the specification of a system using both C language and EXMPLRAD.

2. A complete model of the system is automatically generated that includes requirements, architecture, high level design, low level design and code.

3. Can show the missing dependencies between model components using Complete and Unique verification discussed later in Chapter 6.

4. Enables the a graphical representation of the system model which may help in understanding the overall system model.

4.11 Scope and Limitations of SyModEx2

SyModEx2 was designed for parsing EXMPLRAD source code. The C language was used because it is used in creating embedded systems. In this thesis, other languages are left for future work due to time constraints. Although there are other kinds of relationship present in a software system, our goal was to trace all the "Implements" links in the software system.

SyModEx2 has the following limitations:

1. SyModEx2 currently parses C99 preprocessing directives such as `#include` and `#define`. It cannot parse other preprocessing conditional statements for C such as `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` and `#endif`.

2. SyModEx2 has the advantage that any problems during lexical analysis phase and parsing phase are promptly reported. However, it just says "No" when it cannot parse the source code and it thus becomes hard to pinpoint the source of the error.

3. Properly parses C99 and EXMPLRAD literals. However, it still doesn't recognize built-in GCC functions such as `__builtin_expect`, `__builtin_LINE`, `__builtin_inf` to name a few. Similarly, it also doesn't support 60 C extensions such as `__asm__`, `__inline__`, `__attribute__`.

Chapter 5

Contribution 3: Automated Detection of Abstraction Levels

One goal of this project is to create a technique which can outline the implementation relationship between different system components and provide a mechanism for finding out hidden dependencies between crucial system components. One of the objectives for achieving this goal, is to create a complete system model which allows for verification of implementation links between system components. This proposed solution creates the system model by syntactic and semantic analysis of the C source code and EXMPLRAD specification definitions. Once the system model is developed, the abstraction levels need to be derived from the implementation relationship between components. The abstraction levels are used during the Complete and Unique verification as well as for generating Transitive and SimpleCompleteness links. This chapter describes a technique for calculating the abstraction level of each model component automatically.

5.1 Chapter Outline

The third contribution of this thesis is the automated detection of System Abstraction Levels. The abstraction levels are first defined in a System Model. Secondly, this chapter shows why manual determination techniques are hard as well as error prone. Thereafter, this chapter discusses the parent-child implementation relationship between different components. Section 5.5 also present our hypothesis that there are strong links between the specification level model components and lower level code components. Section 5.6 discusses our approach to embed the C call graph into our system model. Section 5.7 describes how we tag different recursive links. Then, Section 5.8 discuss our method to automatically calculate and generate the abstraction level information using the "Abstraction Level Calculation Engine". Section 5.9 describes the method for visual representation of abstraction levels. Section 5.10 presents the advantages of using our technique to auto generate abstraction levels. Finally, Section 5.11 discuss the scope and limitations of our technique.

5.2 Abstraction Levels in a System Model

A system is composed of many different kinds of modules and sub-modules. Each module and it's underlying sub-module has different functionalities. Each level signifies the difference in

view of the system. How the components interact with each other at different levels is very important for complete understanding of the system [23–25]. The higher level is more concerned and focused on the upper level details of the system [23]. The lower level call graph is more relevant to the programmer or test engineer working on the system [4]. With the use of this tool we can also check for indirect cycles that are present in the system. How this recursion affects the overall system can be of interest to a programmer.

The Upper specification levels present a higher or broader view of the system. The main categories for all these specification level components are the Requirements, Architecture, High Level Design, Low Level Design and Code. These general levels can have lower level sub components such as sub requirements, sub-level architectures, sub design models that integrate together with higher level parent components.

The lower level represents the lower level functionalities that represent the working mechanism of the system. The functions and function calls to other basic functions are used to represent the functionalities provided by the system. All the head level functions represent the exposed functionalities that the system provides. The intermediate functions are used to make different choices to further lower level functionalities. The lowest level function blocks basically consist of basic functions that are used to invoke some hardware or software routines [6, 73]. These are the core mechanisms which are actually used to achieve some goal or task.

5.3 Manual Detection of Abstraction Levels

Software intensive systems are complex in nature. For the purpose of making software design easier, we divide the software into different abstraction levels. As discussed in Section 5.2, each abstraction level presents a different view of the system. Hence, it is natural to categorize not only software but specifications into different abstraction levels.

However, categorizing a software system into different abstraction levels is a hard task. Each project stakeholder has his own view of the system being designed. This view may be too broad or too detailed according to the level of expertise possessed by the project's stakeholder.

The labeling of abstraction levels is done based upon the views of different level of project stakeholders. The requirement, sub-requirements and user stories are based on the perception of the Business Analyst [81]. While the architectural flow of the system is developed by System Architect [81]. The different alternative designs for a particular implementation are based upon

the designs created by the Principal Engineer. Finally, the implementation task is carried out by the Developer [21].

Currently the process of differentiating into different abstraction levels is done by different project stakeholders. This manual process separates the system into different views and each view may be incomplete or error-prone. Thus, developers and analysts need an automated approach which can derive the system abstraction level dynamically based upon the component interrelationship information provided. Using such a methodology, changes made to any view of the system will also result in changes to the system model.

5.4 The Implementation Relationship between System Components

One of the major challenges in software engineering is the realization of relationship between various different components [73, 76]. These components are not necessarily all software code but may consist of different document artifacts such as software requirement specification document, architecture guidelines, and prototype design models [4, 6]. These entities help in correct implementation of the software being designed and serve as a road map for the software developer.

Even though these elements are different and contain variable levels of information, they do have some relationship between themselves. These links between the different abstraction level components varies according to the context. A simple example would be the *"fileContains"* relationship which shows the artifacts contained in the file such as a requirement definition and its corresponding design criteria. SyModEx2 currently checks for the functional correctness between software requirements and its implementation model. Hence, we are only concerned with the *"Implements"* links that exist between components.

5.5 The Implementation Relationship between Low-Level Design and Code Components

The low-level design abstraction level is the lowest level view of the system model. It contains the most granular categorization of higher level components. Low Level Design components are also important in our model because they describe the implementing functions for any particular low level design. This *"ImplementorFunction"* links the lower level components with the implementor function.

Functions are the primary mechanisms to achieve some programmatic task in the C language. If the program is modular then each function is used to achieve one particular functionality. Hence, each functionality can be described as a service whose execution results in some meaningful work being done. Functions have the capability of calling other lower level functions. A *head function* is such a function which is exposed as a service that is run using some higher level invocation mechanism. On the other hand, *basic or fundamental functions* are those lower level functions which do not call any other functions. These basic functions act as a building blocks for providing the overall critical service or functionality provided by the system.

5.6 Embedding the C Function Call Graph in the System Model

A *call graph* is a directed graph which depicts the calling relationships between different functions in a software system [42, 81]. Each node in a call graph represents a function and each edge represents that a caller method calls a callee method. Any cycles present in the call graph represent recursive function calls. A section describing how recursive function calls are handled is presented later in Section 5.7.

The function calls present in the system can be represented in an ordered manner using a call graph. The call graph, often used for debugging purposes, shows the caller and callee relationship between various functions. These related functions represent how the system works to achieve a certain goal. Function calls performed by a calling function are at a lower level than the caller. All the called functions are generally at the same abstraction level.

The *Semantic Analyzer* discussed in Section 4.6, recognizes all function calls performed at different levels using the C99 grammar as a reference and actively decodes them from the Abstract Syntax Tree at the definition of each included function. These function calls are stored in the format `functionCall(FileLocation, LineNumber, Callee, Caller)` in the Prolog Knowledge-base. The `FileLocation` variable indicates the location of the parsed file. The `LineNumber` variable is used to indicate where the function call is encountered within the file. The `Callee` and `Caller` variables represent the called function and the calling function respectively.

This `functionCall` predicate is further processed by the *Model Generator Rules Engine* to create a given implementation link which is stored in the system model as `link(FromID, ToID, From, To, LinkType, LinkGeneratedBy)`. The `FromID` and `ToID` represent the NodeID for

each component in the link. The `From` and `To` represent the component name for each component in the link. The `LinkType` represents the type of link that is used. In our case, we use the "Implements" keyword to specify the implementation relationship between function calls. Finally, the `LinkGeneratedBy` field represents how the link was generated either "Given", "Transitive" or "GenBySimpleCompleteness".

Listing 5.1: SyModEx2 Source for Finding and Marking Recursive Links

```
1
2  removeRecursiveFunctions([HeadFunctList|TailFunctList]):-
3    removeRecursiveFunction(HeadFunctList),
4    removeRecursiveFunctions(TailFunctList).
5
6  removeRecursiveFunctions([]).
7
8  removeRecursiveFunction(Node):-
9    findChains(Node,Node,ChainList),
10   removeLastFunctions(ChainList).
11
12 removeLastFunctions([HeadFunctionList|TailFunctionList]):-
13   reverse(HeadFunctionList,HeadFunctReversedList),
14   removeLastFunction(HeadFunctReversedList),
15   removeLastFunctions(TailFunctionList).
16
17 removeLastFunctions([]).
18
19 removeLastFunction([To|[From|_]]):-
20   usermod:functionCall(X,Y,From,To),
21   retract(usermod:functionCall(X,Y,From,To)),
22   assert(usermod:functionCallRecursive(X,Y,From,To)).
23
24 getFunctionCallLink(From,To,[From|[To]]) :-
25   usermod:functionCall(_,_,From, To),!.
26
27 getFunctionCallLink(From,To,[From|List]) :-
28   usermod:functionCall(_,_,From, Intermediate),
29   getFunctionCallLink(Intermediate,To,List).
30
31 findChains(Source,Destination,ChainList):-
32   findall(SingleInstance,getFunctionCallLink(Source,
       Destination,SingleInstance),ChainList).
```

5.7 Finding and Marking Recursive Links in the System Model

Software applications may use recursion to implement a certain functionality. Recursion is used in software design when it's application is fairly straightforward and the number of steps to implement it are minimal. Recursion can be of two types, namely, **direct** and **indirect** recursion. Direct recursion is usually introduced by a programmer to achieve some specific functionality. Indirect Recursion may be introduced for the same purposes. However, it could be that indirect recursion may have been introduced unknowingly.

Our *Abstraction Level Calculation Engine* specifically does not support recursion because with recursive functions hierarchy cannot be defined since recursive links create a cycle within the model. When a cycle is created SyModEx2 cannot properly distinguish between the caller and the callee function. In this case, the hierarchy shifts as SyModEx2 changes the caller of the function.

Listing 5.2: Example of Recursive C Source Code (adapted from Listing 3.1)

```
1  void Code5(){
2    Code5();
3  }
4
5  void Code2(){
6    Code3();
7  }
8
9  void Code3(){
10   Code4();
11 }
12
13 void Code4(){
14   Code1();
15 }
16
17 void Code1(){
18   Code2();
19 }
```

SyModEx2 uses a simple approach to remove the recursions present in a fully functional call-graph. The predicates used to remove the recursions are shown in Listing 5.1. First, SyModEx2 calculates all the functions present in the system model and passes it to the predicate `removeRecursiveFunctions` for processing. Each function is processed individually by the predicate `removeRecursive-Function`. Predicate `removeRecursiveFunctions` calculate a list of recursive chains present for each particular function using the `findChains` predicate. Each chain is individually processed by the predicate `removeLast-Function` and here the recursive function call is retracted and a new recursive function call i.e. *"functionCallRecursive"* is asserted to make the system model complete again. Hence, the traceability information is not lost but just changes form. After using this approach the model can then be further processed to generate the abstraction level information for each node.

A simple recursive C source code, which has been adapted from Listing 3.1, is presented in Listing 5.2. There is a recursive relationship for functions `Code5` and `Code1`. For the purpose of brevity, the accompanying requirements, architecture and design are not shown. Running SyModEx2 on this project generates the system model illustrated in Figure 5.1. Here, the black

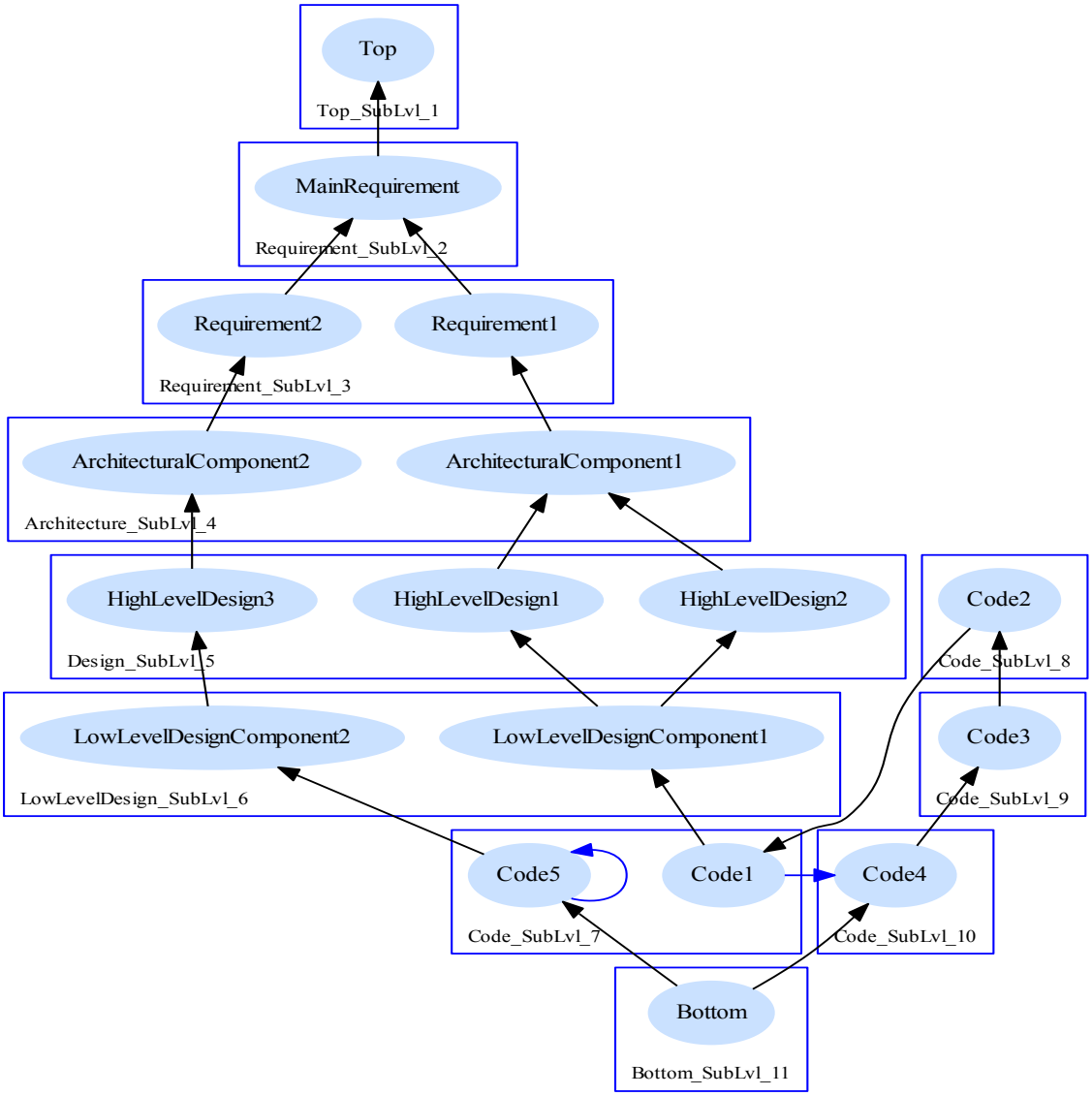


Figure 5.1: System Model for Recursive C Functions Presented in Listing 5.2

links represent normal given links and the blue links represent the recursive given links. From Figure 5.1, it can be clearly seen that:

- Function Node `Code5()` has a **direct** recursive link.
- Function Node `Code1()` and `Code4()` have an **indirect** recursive relationship between them.

5.8 Automated Determination of Abstraction Levels

The *Abstraction Level Calculation Engine* proposed is adaptive in nature, meaning that it automatically adapts to the changes performed on the input model. Any changes that affect the model can easily be visualized and reported to the user. Each hierarchy in the model is generated by using the main hierarchies, namely the requirements, architecture, high level design, low level design and code followed by a number which denotes the abstraction level of the included components.

The general working mechanism of the algorithm is given in Algorithm 1.

Algorithm 1 Automated Abstraction Level Calculation Algorithm

```
1:  procedure CALCULATEABSTRACTION( fullNodeList )
2:      L ← [Top, Req, Arch, HighLvlDesign, LowLvlDesign, Code, Bottom]
3:      counter ← 0
4:
5:      for absLevelType ← L do
6:          filteredNodesList ← getNodesOfType(fullNodeList, absLevelType)
7:          headNodesList ← getHeadNodes(filteredNodesList)
8:          writeLevelInfo(headNodesList, counter)
9:          counter ← counter + 1
10:
11:         while 1 do
12:             subAbsLvlList ← getImmediateChildren(headNodesList)
13:             if !empty(subAbsLvlList) then
14:                 resultList ← getUniqueChildrenList
15:                 writeLevelInfo(resultList, counter)
16:                 counter ← counter + 1
17:                 headNodesList ← resultList
18:             else
19:                 break
```

In Algorithm 1, the list L in Line 2 refers to the general abstraction levels. The variable *counter* in Line 3 denotes the abstraction level calculated for each system model component. The first component (the *Top* component), has the abstraction level 0 because this is the uppermost abstraction level possible in the system model. *Top* is considered the highest abstraction level because it doesn't have any other model component as it's parent. The *Top* node connects to all the head nodes (nodes which do not have any parent). Typically, most of the higher level requirements are associated with the *Top* component. Unlike other nodes, the *Top* node doesn't contain any other sub components and is the highest ranking node.

Firstly, Algorithm 1 extracts an abstraction type value from the list L and store it in the variable *absLevelType* on Line 5. The algorithm generates a *filteredNodesList* for each general abstraction level which corresponds to the *absLevelType* variable. The predicate *getNodesOfType* in Line 6, calculates the filtered list using the *fullNodeList* and *absLevelType*. After that,

Algorithm 1 calculates all the nodes which do not have any parent components, these nodes are termed as "head" nodes. Algorithm 1 writes the head nodes with the counter information as the calculated abstraction level. Thereafter, Algorithm 1 increases the *counter* value by one.

Secondly, Algorithm 1 calculates the abstraction level for each of the sub components. The idea behind this is as follows: calculate a list of all the lower level children components for the head nodes. This list containing immediate children of the head nodes is designated by the *subAbsLvlList* and the predicate used to retrieve this list is the *getImmediateChildren* in 1, Line 12. If this calculated list is empty Algorithm 1 can *break* from the *while* loop and calculate nodes for another abstraction level. If *subAbsLvlList* is not empty then, Algorithm 1 retrieve a sub list of all the unique immediate children i.e, the *resultList* variable. The *resultList* variable contains those unique nodes which are not called later by some other function further down the execution cycle. Further, SyModEx2 saves all the information in the model knowledge-base. The *counter* is then increased by the value of 1 and the resulting list is assigned as the new *headNodesList*. The process is reiterated until all the nodes have been assigned an abstraction level value, after which the algorithm terminates.

5.9 Visualization of the Abstraction Levels Hierarchy

The different levels of hierarchy between the abstraction levels are generated using SyModEx2 toolset. The generated graph depicts the order of abstraction levels calculated using our automated *Abstraction Level Calculation Engine*. The DOT language [44] graph is automatically created for each project analyzed using GraphViz DOT tool [38] for the image output. An example of such a graph is presented in Figure 5.2 for the example system shown in Figure 3.1 of Section 3.3.

5.10 Advantages of the Presented Approach and Algorithm.

The advantages of this algorithm are stated below:

1. Automatically generates the different abstraction level of components in a model.
2. Basis for performing Complete and Unique verification between system components in SyModEx2.
3. Any changes made to the system are observed immediately as computed levels may change and the corresponding abstraction levels for the components will change accordingly.

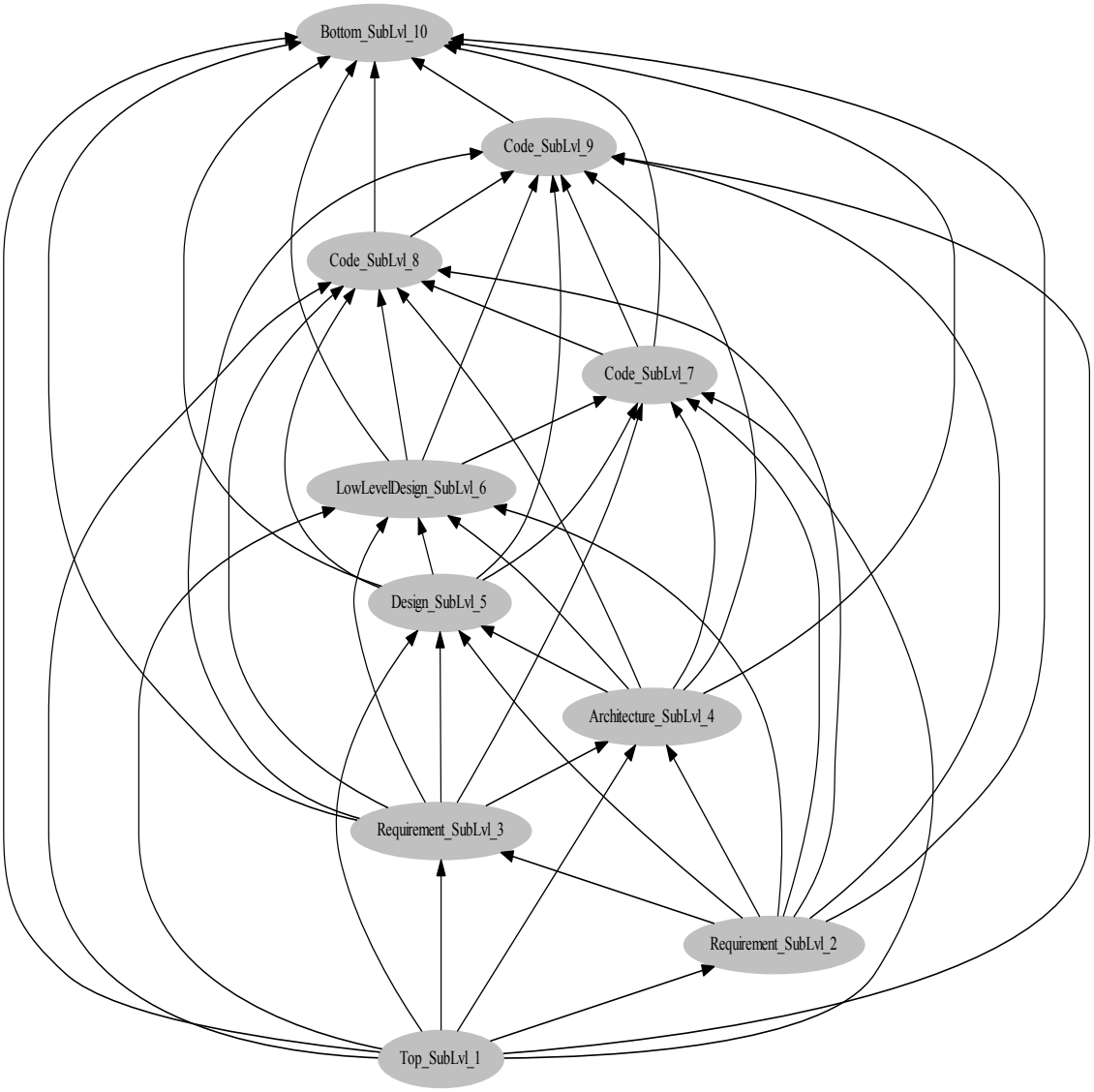


Figure 5.2: Hierarchical Order of Abstraction Levels for an Example System Model

4. Project stakeholders just need to outline the implementation relationship occurring between specification components.

5.11 Scope and Limitations of the Presented Approach and Algorithm.

The presented abstraction level calculation algorithm has been designed for automating the process of identification of abstraction levels between system components. It relieves different project stakeholders from the responsibility of classifying the hierarchy levels of the components. This algorithm maintains the fact that components called later during the program cycle are further below in the system model. The motivation for this is that the primary functions

are called more frequently during the lower phases of the program execution. As discussed in Section 5.6, the algorithm derives the relationship between code level components using the C call graph relationship. However, developers must specify the relationship between specification components using the parent child relationship. The algorithm will not derive the abstraction levels if these links are not present.

Chapter 6

Contribution 4: Semi-Automated Verification of Complete and Unique Implementation Between System Components

Developing complex software intensive systems is hard and error-prone. Stakeholders need a methodology which can aid us in outlining the specification along with the implementation details and help us check the validity of implementation with a given specification. The EXMPLRAD helps project stakeholders specify specifications alongside the functional code components. Similarly, the SyModEx2 is used to create a system model which can be checked for errors. This chapter presents an algorithm to outline any missing dependencies between selected system components.

6.1 Chapter Outline

This chapter presents the fourth contribution of this thesis. This chapter introduces and describes a semiautomatic procedure for verifying the Complete and Unique implementation between components proposed by Conte de Leon and Alves-Foss in [23, 25]. Firstly, Section 6.2 give a brief introduction about the theory behind the Complete and Unique verification mechanism. Secondly, Section 6.3 give an overview of the two formal verification techniques, the Complete and Unique implementation techniques. Thereafter, this chapter describes how the chaining mechanism from graph theory is utilized to implement the "Model Verification Engine" and then verify the Complete and Unique implementation dependencies between system components [25].

The proposed verification technique is based on the work performed by Conte De Leon and colleagues in [23, 25]. By using this methodology, many of the unforeseen and unaccounted problems can be resolved by limiting the design of the system which are proven to implement both functional as well as non-functional requirements.

6.2 Need For a Holistic Formal Implementation Model of a System

Most current software development techniques do not provide a means for verifying the correctness and accuracy implementation relationships between provided specification and code components [25]. This problem exists because current techniques do not impose the creation and maintenance of essential traceability links, which are crucial for establishing the relationship between various different system modules and their corresponding specifications [22, 55, 79].

Development of critical and high assurance software intensive systems are characterized by well-known engineering and design techniques which help in analysis and development of modules and sub-modules [42, 73, 81]. However, a problem that still exists is that the formal analysis and attestation of security procedures for components used inside the system is hard to find and not commonly used in practice [25]. It has been found that by using formal validation mechanisms the functional correctness of the system can be guaranteed to a great degree [24]. Nevertheless, a common problem here is that the formal model parameters are not bound properly to the actual implementation model of the system.

Using the approach described in Section 4.7, this thesis creates a system model which contains all the necessary information to enable model property verification. This highly detailed and structured model also enables developers to perform Complete and Unique verification operations on the system and helps to automatically identify the inter-component dependencies.

6.3 Definitions of Complete and Unique Implementation Between Model Components

This section provides formal definitions for Complete and Unique implementation between model components or work product sections (WPS). These definitions have been given by Conte de Leon et. al in [23, 25]. The description they provide serve as guidelines for carrying out the implementation of the Unique and Complete formal model verification.

The following formal definitions are provided by Conte de Leon and Alves-Foss for Complete and Unique implementation of a system.

> **Definition 1. Complete implementation between work product sections:** *"A wps b (implementor) completely implements a wps t (implementee) if and only if every chain <C,implements> in* Γ*, whose minimum element is Bottom and whose maximum element is t contains b."* [23, 25].

Complete implementation of a component relies on the fact that if a component "b" completely implements another component "t" at a different abstraction level then there can be no other alternative paths, from "Bottom" to "t" through which component "b" is not included. While performing this verification, if there are routes between the bottom level abstraction and the implemented "t" which miss out the implementor "b" then the implementation is in-

complete. Complete implementation helps us ensure that the dependency of a component is properly defined.

> **Definition 2. Unique implementation between work product sections:** *"An implementor wps b uniquely implements an implementee wps t if and only if every chain <C,implements> in Γ, whose maximum element is Top and whose minimum element is b contains t."* [23, 25].

Unique implementation assures that a dependency relationship between the Top level component and a component "b" contains the component "t". It ensures that if there is a functional component then all routes have the critical component needed for proper functioning of the system. A common example of such a component can be the central authentication mechanism in a login system. The authentication module is to be called every time a user tries to login using different login interfaces.

6.4 Verification of Complete Implementation

In graph theory, a directed path in a directed graph is an ordered sequence of links which connect an ordered sequence of nodes [26]. Calculating a path from one node to another node becomes important to infer answers from the Prolog knowledge base created using the Model Verification Engine [13]. Each path is referred to as a "Chain" from "Source" to "Destination". The Model Verification Engine generates a sub-graph from the original graph denoted by the system model, by creating a list of all "Chains" from "Source" to "Destination".

The *Model Verification Engine* is thus a combination of the inference mechanism, the knowledge base and the input data fed to the engine. The verification results generated by the *Model Verification Engine* is further based on the chains created during the execution of the engine.

In a forward chaining approach, searches are performed on facts to derive a conclusion about these facts [13, 26, 84]. For forward chaining, the data values in working memory are matched against the rule definition in the predicate rule-base. Each successful execution instance outputs a path from a given source node to the given destination node. In this work, this path is represented using a list of nodes. This list contains nodes which are consecutively connected to one another. A collection list containing all these paths is then created.

Formally verifying Complete implementation can be done by checking if the required searched node is a member for each instance of a given path. If each path instance contains the searched

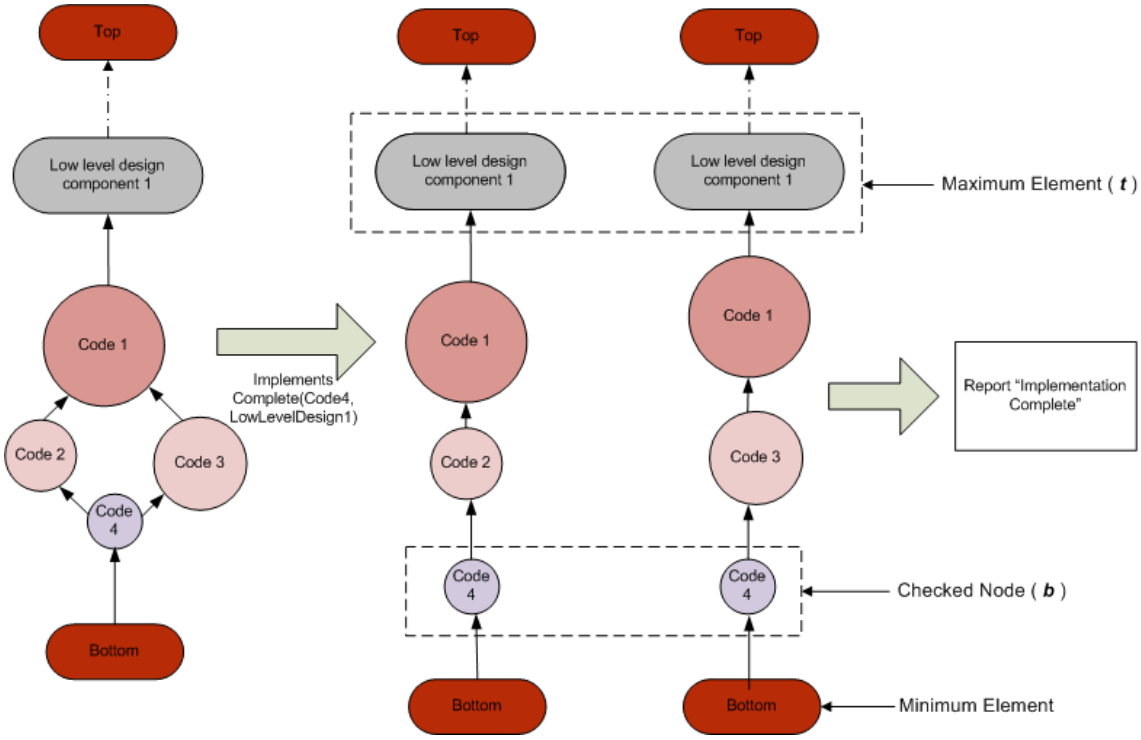


Figure 6.1: Verification of Complete Implementation using Forward Chaining

node then the provided search node completely implements the given higher level node. Hence, this check reveals if any other dependencies are missed by a higher level component.

A small example of the Complete implementation between model components *Low level design component 1* and *Code 4* is presented in Figure 6.1. Here, in Figure 6.1, the simple graph is separated into two distinct chains. These chains are checked to see if *Code 4* component is present in all of the chains where the maximum element is *Low level design component 1* and minimum element is *Bottom*. The dashed line between *Low level design component 1* and *Top* represent that there are other components between these system components.

6.5 Verification of Unique Implementation

A backward chaining mechanism works by providing conclusions to drive the given facts [13, 26, 84]. If these facts exist then the assumed conclusion is marked as true. This method is also referred to as a goal driven approach [13]. For chaining backwards, the engine matches a goal in the knowledge base against conclusions of rules in the rule-base. When one of the goal matches, this can produce other goals and the cycle continues until it outputs a path from source node to destination node.

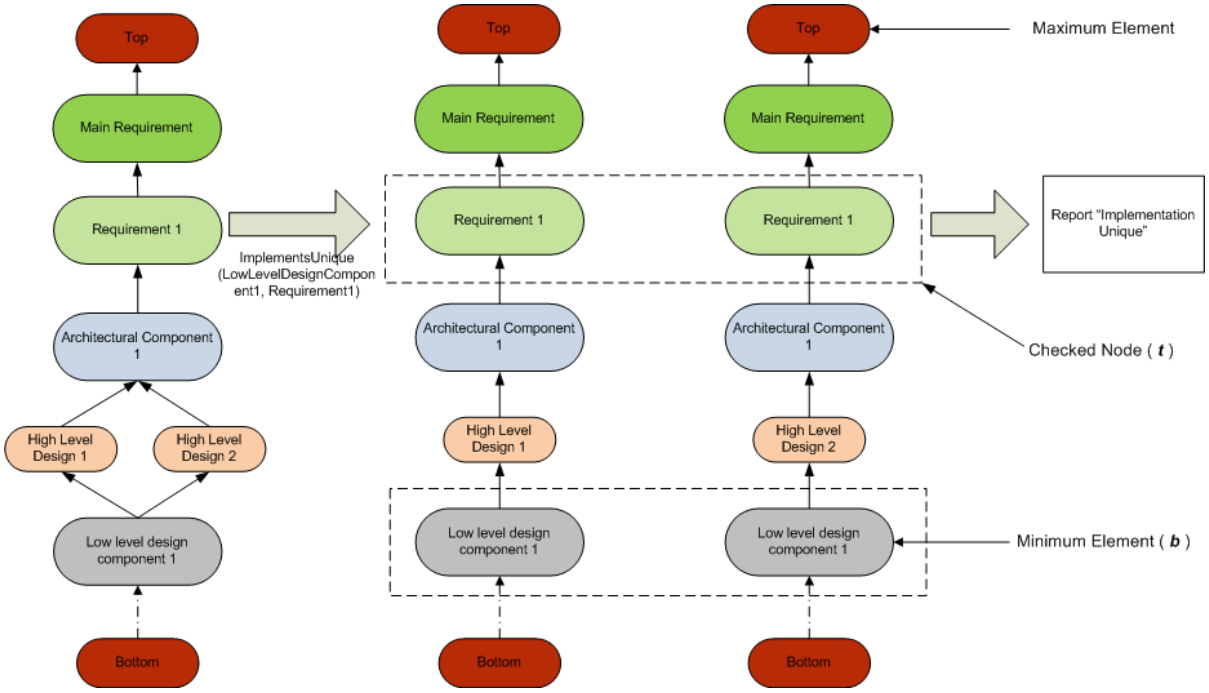


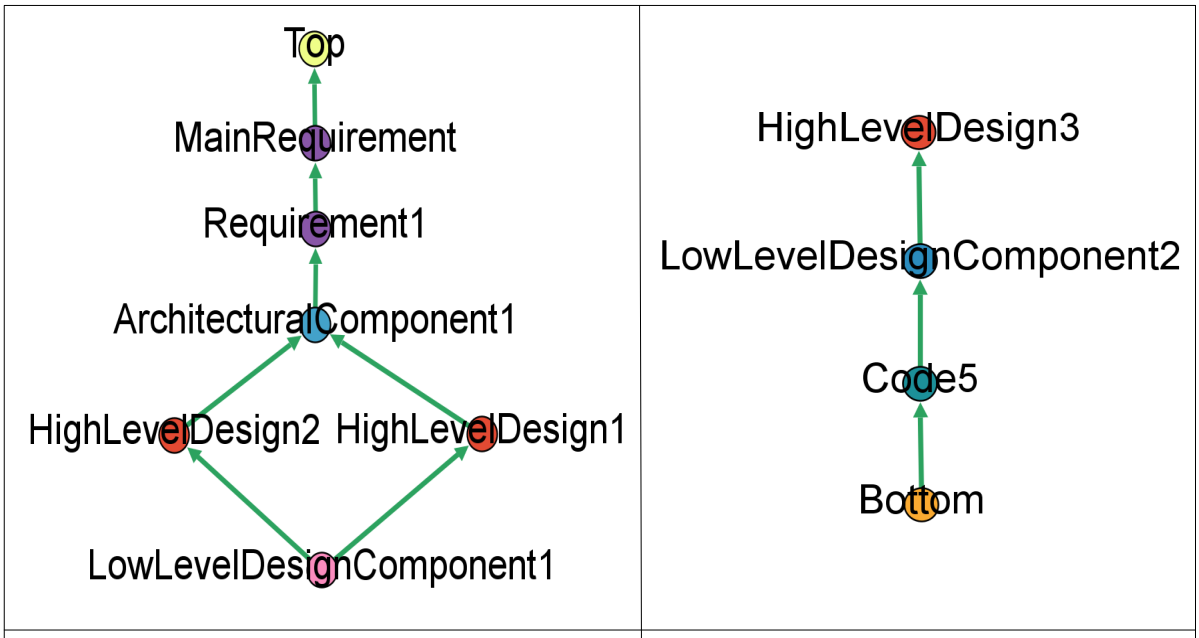
Figure 6.2: Verification of Unique Implementation using Backward Chaining

Similar to the above approach, the Model Verification Engine creates a list of paths which move from a given source to destination point. Each path is then individually examined to check if the searched node is a member of the given directed path. If the engine encounters a false result, then it concludes that the implementation is non-unique. This check verifies that the lower level dependency uniquely implements the higher level abstraction and nothing else. Anything else implemented by the system can be analyzed and checked to see if this has any unknown side effect on the system.

An example of the Unique implementation between model components *Low level design component 1* and *Requirement 1* is illustrated in Figure 6.2. Here, in Figure 6.2, the simple graph is separated into two distinct chains. These chains are checked to see if *Requirement 1* component is present in all of the chains where the maximum element is *Top* and minimum element is *Low level design component 1*. The dashed line between *Low level design component 1* and *Bottom* represent that there are other components between these system components.

6.6 Visualization of Complete and Unique Verification Results

The Complete and Unique verification are performed on any two model components to see if any missing dependencies exists between them. Even though we do get text based results



a) Unique Implementation between LowLevelDesignComponent1 and Requirement1 Drawn Using Gephi

b) Complete Implementation Between LowLevelDesignComponent2 and HighLevelDesign3 Drawn Using Gephi

Figure 6.3: Visualization of Complete and Unique Implementation Sub-Graphs Using Gephi

stating whether the components do satisfy the Complete or Unique property; some visual aid would definitely help to pinpoint any missing links exiting between the selected components. Hence, we have developed a graph which is typically a subgraph of the whole system model and contains links which show the "Given" and "Missing" links in the model. All the "Given" links are represented by the "Green" arrowheads and the "Missing" links are represented by the "Red" arrowheads. An example of the Complete and Unique analysis is shown in Figure 6.3 for the example system model described in Figure 3.1 of Section 3.3.

6.7 Advantage of Using the Complete and Unique Verification Techniques

Complete and Unique verification between system components provide crucial information about which components are properly connected and there are no "missing" links between important components. They provide a strong proof that if there is a one to one connection between components then there are no hidden dependencies between these components. The Complete verification can more readily capture the relationship between important functional level components. The Unique verification can help justify the relation between higher level specification components because the specifications are connected more closely to each other.

Typically, there are high level functions which provide some kind of service which are made up of other smaller "primitive" functions. Even though the Complete and Unique verification in these components may fail, it is plausible because they provide a combination of different functionalities. Hence, with the use of Complete and Unique implementation we can pinpoint design flaws present in the software system and avoid any critical software errors.

Hence, the Unique and Complete checking mechanisms can be used easily to verify the dependency behavior of the system. These techniques will aid project stakeholders to find any missing dependencies which can obstruct the information flow of the critical system and helps in removing faults before they actually occur. These formal mechanisms should be rigorously applied to systems that provide critical functionality. This will increase security and safety aspects of the critical system.

6.8 Scope and Limitations of the Complete and Unique Verification Techniques

Complete and Unique verification can be applied to a number of different software systems. The verification is performed to check the implementation details of a system. They will not check if the functional component has been properly implemented or not. Hence, proper quality testing of the software product must be done to remove these issues. Even though Complete and Unique verification are powerful methods they may not be able to avoid all errors in the system. This is because there might still be errors in the specification or implementation. Hence, it makes sense to actually make this approach a part of a bigger tool suite with code checking functionality as well. Also, one main dependency for running the Complete and Unique verification is the system model. If the generated system model is absent or incomplete this technique will become ineffective.

Chapter 7

Case Study : Complete and Unique Implementation Verification of the SEL4-IPC Subsystem

Software intensive systems are prone to errors and faults. In current years, there has been a lot of advancement in field of software error detection by the use of static and dynamic source code analysis. However, the community still lacks tools to pinpoint errors between software specifications and system implementation. The approach mentioned in our thesis is a step towards solving this problem. The previous chapters discussed how to express the specification using EXMPLRAD and analyze the system using SyModEx2. This chapter introduces the SEL4 microkernel and the application of our technique and toolset for analyzing this software system.

7.1 Chapter Outline

In this chapter, Section 7.2 gives an introduction to the SEL4 microkernel. This chapter also describes how a microkernel differs from a monolithic kernel. This chapter also discusses why SEL4 is a significantly better achievement, in terms of safety and security, from its other microkernel counterparts. Section 7.3 then focus our attention to give a general introduction to the IPC mechanism in the SEL4 microkernel. After that, Section 7.4 discusses in detail how the common authentication mechanism is implemented in the SEL4 microkernel using the capability rights model. Section 7.5 discusses the proposed system architecture for the IPC model in SEL4 using EXMPLRAD. This chapter then discusses the application of SyModEx2 for verifying the Unique and Complete implementation consistency between SEL4-IPC component pairs and show the obtained results.

7.2 Introduction to the SEL4 Microkernel

A kernel plays a crucial part in modern operating systems. A kernel is responsible for managing the input/output requests from applications or system software. A kernel is written in a protected region of memory so that other processes may not overwrite the kernel code and data structures [56, 57].

A kernel may be further categorized into two broad categories :

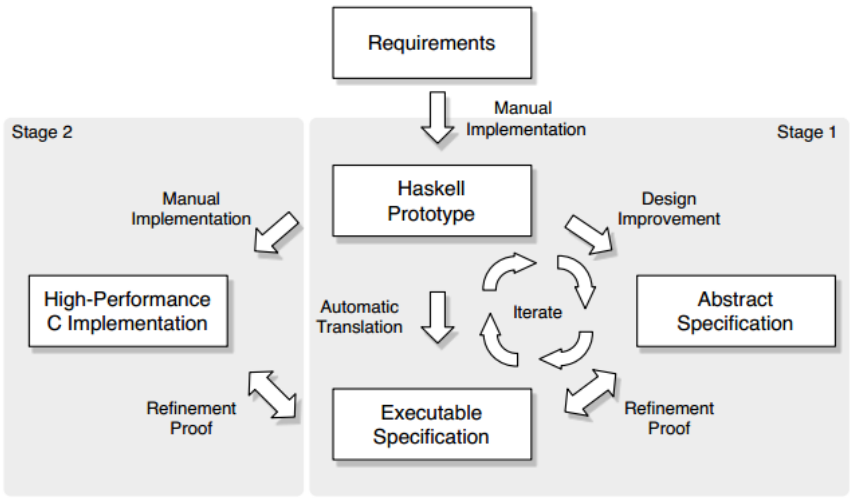


Figure 7.1: SEL4 Design Procedure (adapted from [57])

1. **Monolithic Kernels** : For a monolithic kernel, the OS services and the main kernel thread run alongside each other, so they reside in the same memory region [75]. This technique provides easier and robust hardware access. The main disadvantage of such kernels is that they introduce close dependencies between system components [57]. From a high assurance view-point it is harder to verify a monolithic kernel because of it's tightly coupled architecture between different module components. A popular example of the monolithic kernel is the LINUX operating system.

2. **Micro Kernels** : A microkernel is a small piece of software or code that contains all the necessary functions and features necessary to interface with the hardware devices and implement an operating system [56, 57, 75]. The primary motivation for designing a microkernel is leaving as much as little in the system space and moving all the other tasks into the user-space. Typically, a microkernel is designed and configured for a particular device or platform. Usually, a microkernel is designed by defining a simple abstraction over the hardware, with a set of primitives which implement memory management, thread control, and interprocess communication facilities [56, 57]. All other OS services are run as user-level processes. SEL4 is one such implementation of a bare-minimum general purpose microkernel which has been enhanced for security [37, 39, 56, 57]. The main disadvantage of using a microkernel is it's additional communication overhead in comparison to monolithic kernels [56, 57].

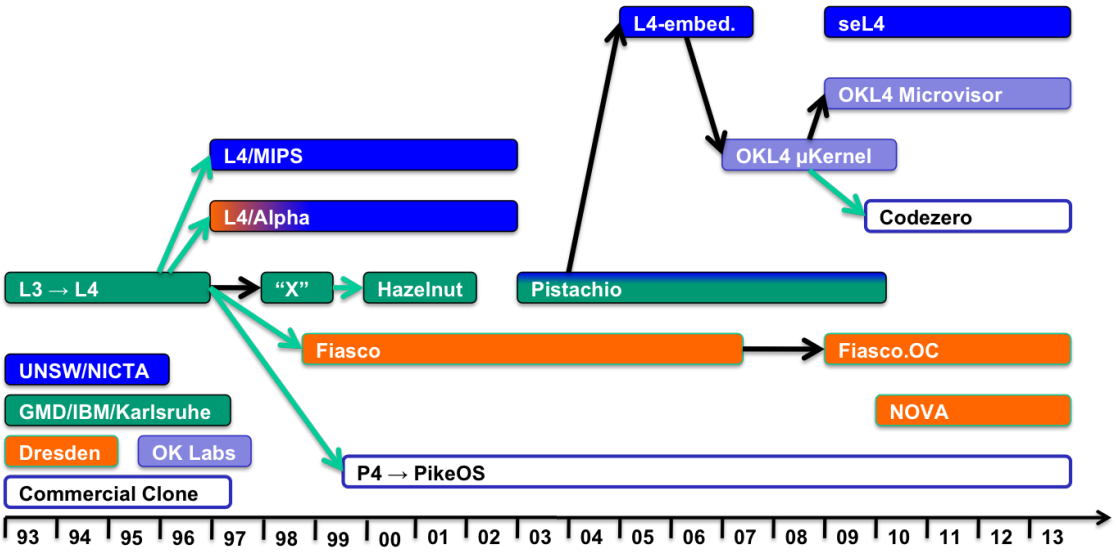


Figure 7.2: L4 microkernel Family Tree (adapted from [39])

Due to the small size of a microkernel, the trustworthiness problem becomes more manageable. With careful considerations in design the size of the kernel can be brought down to nearly 10,000 lines of code (10 kloc) [57]. SEL4 is one such instance of a microkernel which is still small in size and fully under the realm of full formal verification [39, 57]. SEL4 is primarily based on the design of the L4 microkernel family which is known for providing provably strong security mechanisms while also taking into consideration performance. Klein and colleagues [56, 57] stated that SEL4 has been checked for functional correctness. An implication of the SEL4 is that the kernel is formally proved to be free from security vulnerabilities such as code injection attacks, buffer overflows, NULL pointer access, memory leaks, non-termination problems, unchecked user arguments, and arithmetic exceptions [56]. The purpose of creating the SEL4 microkernel is to create industrial grade and fully-verified kernel with low complexity which would make formal verification possible while also maintaining the performance aspects of the kernel.

SEL4 is a third-generation microkernel which is similar to other microkernel systems such as Nova [83] and Coyotes [27]. Similar to L4 it has characteristic designs similar for thread spaces, virtual address memory, IPC and capabilities for authorization. A brief description is each features is provided below:

1. **Interprocess Communication** : Interprocess communication (IPC) refers to the communicating mechanism that occurs between two processes or threads. In SEL4, IPC

occurs in two forms: a) asynchronous and b) synchronous [32]. Synchronous transmission occurs by using endpoints whereas asynchronous notifications occur with the help of asynchronous endpoints [32, 57]. Both `fastpath` and `slowpath` functionalities exist for the synchronous transmission form. Remote procedure calls (RPC) are performed using the synchronous IPC via the `reply` system call. During transmission it is also possible to send capabilities with the message to help perform some task [56]. Each send capability is associated with a unforgeable badge which determines the authority level and sending thread's identity [32]. Additional information regarding the SEL4-IPC mechanism is explained in Section 7.3.

2. **Authority Using Capability** : A capability model is used to confer authority over object instances [37]. Capabilities are stored in their own capability space made up of container objects known as Capability Nodes (CNodes) [12]. SEL4 consists of 7 system calls in which 6 of them require holding of a capability. These system calls are [32]:

   (a) *sel4_Send()*,

   (b) *sel4_NBSend()*,

   (c) *sel4_Call()*,

   (d) *sel4_Wait()*,

   (e) *sel4_Reply()*,

   (f) *sel4_ReplyWait()*,

   (g) *sel4_Yield()*

   Only *sel4_Yield()* doesn't require a capability and is used to yield to a scheduler. Even though the number of calls present is less, the kernel's API is composed of all the interfaces presented by all kernel object types [57].

3. **Kernel Memory Management** : Memory management for the SEL4 microkernel is explicit. All the kernel structures are either dynamically allocated first-class objects in the API or statically allocated memory during boot time [56]. These data structures may act as user-implicitly called small kernel services or in-kernel data structures [57]. A simple example of such structures are the CNodes, TCBs, and page tables.

4. **Virtual Address Spacing** : For the allocation of virtual address spaces, the virtual memory related kernel objects such as page tables, page directories, and frames are externally manipulated [32, 57]. These kind of address spaces do not have any defined structure. The distinction on whether the user-level system is para-virtualized monolithic OS or similar to an exokernel depends upon the map and unmap interfaces linked to the Frames and PageTables [57].

5. **Thread Management** : Threads are one of the most widely primarily used entity in SEL4. High level abstractions such as processes and virtual machines are a result of a combination between the capability nodes, the virtual address space associated with a thread, and the user policies enforced upon the entity [57].

6. **Exception Handling** : Each thread has an exception handler associated with it. Exceptions or faults that occur are propagated using synchronous IPC to the corresponding thread's exception handling code [57]. A simple example is if a page fault occurs then it is propagated to the thread's page fault handler.

The SEL4 microkernel is available for both ARM and IA32 microprocessor architectures [57]. For this thesis, we analyzed the IA32 code. We have derived the IPC and the IPC authentication mechanism EXMPLRAD definitions from the SEL4 reference manual [32].

7.3 General Interprocess Communication Mechanism (IPC) in SEL4

SEL4 provides an built-in message-passing communication medium for threads. Messages are transferred between objects by explicitly calling a capability to a designated kernel object [57]. Messages which are passed to either asynchronous or synchronous IPC endpoints are routed towards other threads [32, 57]. Each IPC message has a tag associated with it. This tag contains four distinct fields: a) **label**, b) **length of message**, c) **number of capabilities transferred** and d) **the *capsUnwrapped* field** [32]. No modifications are performed on the label field of the kernel and it may be used to specify the operation to be performed on the sent message. The message length and number of capabilities field specifies the amount of message registers and capabilities that the sender wants to transfer or the amount actually transferred. The final field *capsUnwrapped* is used on the receiver side to show the sequence in which the capabilities were received [32, 57].

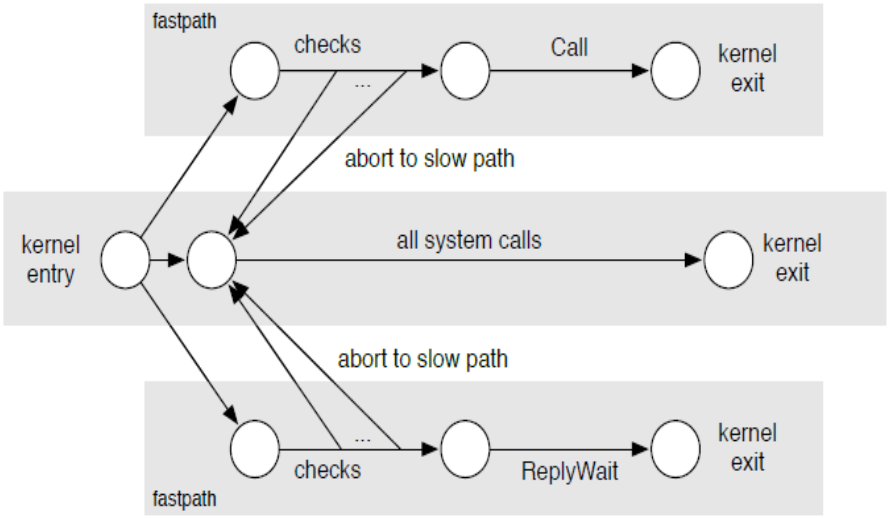


Figure 7.3: SEL4 IPC FastPath Mechanism (adapted from [57])

A small amount of data and capabilities are allowed to be transferred between two threads using synchronous IPC endpoints [57]. The transfer operation in such endpoints doesn't occur if the sender is not able to send the message or the receiver to receive the message. These endpoints can be configured to create a new capability with a badge attached to it. Whenever the SEL4 source thread sends a message to receiver via the endpoint, the badge is transferred to the receiving thread's badge register [32].

The synchronous IPC mechanism can be further categorized into the fastpath and the slowpath mechanisms. Fastpath is an enhancement in the SEL4 design that is used to perform fast IPC calls between threads. The fastpath mechanism focuses on the `Call` and `ReplyWait` system calls collectively referred to as kernel fastpath [57]. The first half of the fastpath code checks for rights and if the IPC request has the appropriate rights, it can utilize the fastpath mechanism otherwise it switches to the slowpath mechanism with a common function to handle to switch between other system calls and exceptions. The synchronous capability checked by the fastpath and slowpath functions is *cap_endpoint_cap*.

The functions used for fastpath mechanism are the `fastpath_call()` and `fast-path_reply_wait()`. The `fastpath_call()` function is used for the `Call` system call. The `fastpath_reply_wait()` function is used for the `ReplyWait` system call. The slowpath using IPC mechanism is the `slowpath()` function. The kernel fastpath case targets the case when the message will be send immediately and the control will be relinquished along with the message. The

fastpath technique often reorders the write sequences in the kernel and thus make it hard to find execution points in the system [57].

7.4 Authentication in SEL4-IPC using a Capability Model

One of the crucial security functions that the OS kernel provides is access control. Access control is used to impose security restrictions such as minimum level of authority, data integrity and confidentiality of information on all kernel services [57]. The SEL4 microkernel's primary objective is to provide a secure and safe environment for a guest operating system [32, 56, 57]. In SEL4, the access of a thread to other threads and page tables is controlled and restricted.

In the access control model, the part of the system state which is used to make access control decisions is known as *protection-state* [37]. The capabilities possessed by each thread explicitly represent the implementation-level protection state of the object in context. This state defines how the object can be accessed by a subject in terms of read and write operation. It also describes how the external subject may modify the protection settings of the object [37].

The way SEL4 provides authorization to it's various kernel objects such as threads is by using the capability model [37, 57]. A capability is defined as an unfakeable identifier that references a particular kernel object and describes the access rights that define what functions may be explicitly called [12, 37]. A capability resides inside an application's capability space (CSpace) [57]. A number of slots are present in the capability space which define a particular right. An application holds references to these slots and performs the required action [32]. Capability spaces are composed of kernel-managed capability nodes stored as a directed graph. Capabilities support move, copy, and send via IPC operations [32].

As discussed earlier, synchronous IPC endpoints have capabilities embedded inside their data structures. The endpoints can have capabilities such as Read, Write or Grant depending on type of end point created. If it is a `Call` endpoint then there must be Send and Grant capabilities which authorize the endpoint to transfer capability [57]. For `ReplyWait` endpoint we need to have a Receive and Send capability to receive the payload and send the acknowledgment about the message being properly received [57]. The functions used for checking if the endpoints contain the particular capability are :

1. **cap_endpoint_cap_get_capCanSend()**: Function which checks for capability required to send the object through the endpoint.

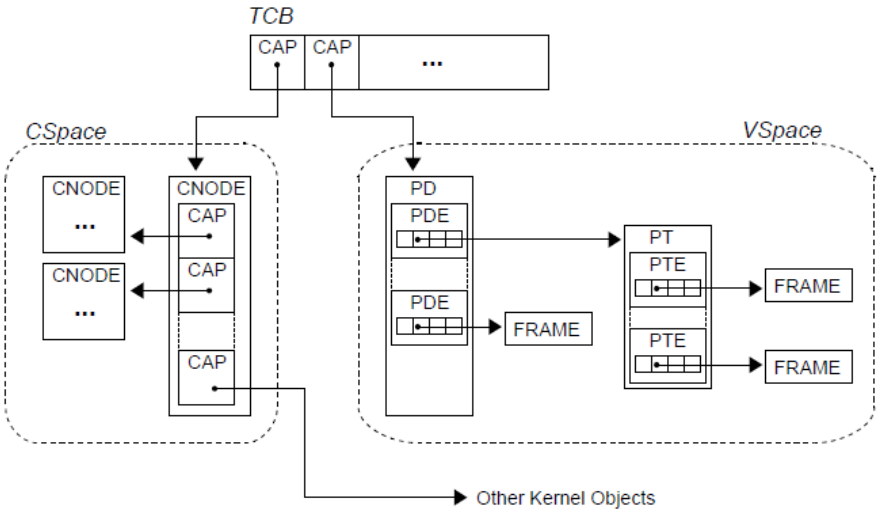


Figure 7.4: Capability Mechanism in SEL4 Threads (adapted from [57])

2. **cap_endpoint_cap_get_capCanReceive()**: Function which checks for capability required to read the incoming object through the endpoint.

3. **cap_endpoint_cap_get_capCanGrant()**: Function which checks for capability required to grant access to the right through the outgoing object i.e, used for transferring capabilities.

7.5 Specifying the SEL4-IPC Architecture Using EXMPLRAD

We applied EXMPLRAD for modeling the IPC Model and it's corresponding IPC authentication mechanism. We carefully read through the documentation and source code to separate the IPC related components present in the SEL4 microkernel. The presented model is our view of how the system has been designed by the SEL4 designers and developers. Figure 7.5 represents a part of the generated model.

Our primary objective was to separate the higher level abstractions present in the system. Our secondary objective was to divide the design model of SEL4 and reformat it into EXMPLRAD compatible format. As discussed earlier we have sub-divided our abstraction hierarchies to five different levels.

1. **Requirements Abstraction Level in SEL4-IPC** : The first abstraction level group represents the requirement specifications for the SEL4 microkernel system. The first one being the requirement level abstraction which is composed of the `MainRequirement`

followed by the `SEL4Software Requirement` which is further divided into the `SEL4IPC-Requirement`, `SEL4AuthRequirement` and the `SEL4OtherRequirement`.

The `MainRequirement` contains all requirement specifications for the SEL4 microkernel. Similarly, the `SEL4SoftwareRequirement` contains all the software related requirements for SEL4. All the features provided by SEL4 to the operating system are a part of the `SEL4SoftwareRequirement` requirement. Since we are modeling the IPC and the IPC authentication modules, we have added the necessary requirement definitions for these software components. `SEL4IPCRequirement` capture all the IPC related functionalities present in the SEL4 system. The `SEL4AuthRequirement` defines the access control model present in the SEL4 model which gives an overview of how authorization occurs between kernel objects. The `SEL4OtherRequirement` is the overarching requirement which captures all the other requirements present in the SEL4 system such as thread management, and memory management.

2. **Architectural Abstraction Level in SEL4-IPC** : The second abstraction level group belong to the architectural level hierarchies. The lower level requirements and the architectural components have a one-to-one mapping. The `SEL4IPCRequirement` is mapped to the `IPCArchModel`, the `SEL4AuthRequirement` is mapped to the `AuthModel` and finally the `SEL4OtherRequirement` is linked to the `OtherComponentsArchModel`. This mapping is clearly visible in the parent relationship defined in Figure 7.5.

   The `IPCArchModel` encompasses all the IPC related modules which exist in the SEL4 system. The `AuthModel` comprises of all the architectural designs of authentication mechanism of the SEL4. The `OtherComponentsArchModel` is an abstraction which contains the architectural specification for all the other software components present in the system which are not related to the IPC or IPC authentication.

3. **High Level Design Abstraction Level in SEL4-IPC** : The third general abstraction level group is the high level design of the SEL4 microkernel. Klein et.al. state the following in their article "Comprehensive Formal Verification of an OS Microkernel" which provide further justification for our high level design:

   > *"The first half of the fastpath checks whether the current situation falls within the optimised case. If so, the second half of the fastpath handles the system*

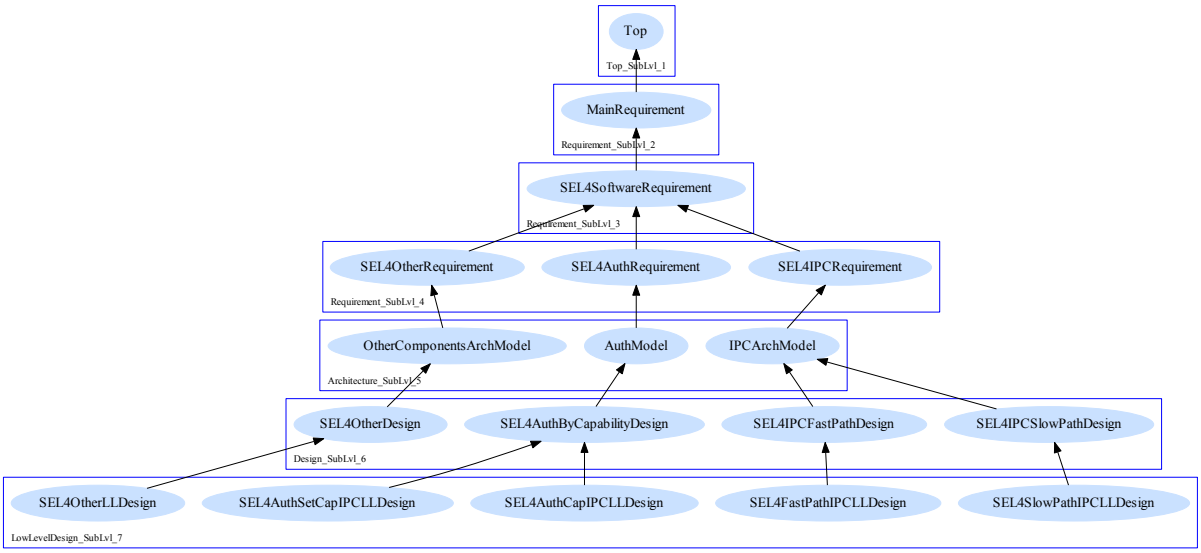


Figure 7.5: Snapshot of the System Model for the SEL4-IPC Design

call. If not, the fastpath calls back to the standard SEL4 system call entry point (sometimes called the slow path), which handles the more general case." [57].

Hence, the IPC components are divided into the `SEL4IPCFastPathDesign` and the `SEL4-IPCSlowPathDesign`. The `SEL4IPCFastPathDesign` refers to the fastpath code present in the kernel responsible to make fast system calls. The `SEL4IPCSlowPathDesign` refers to the general slowpath mechanism to make general system calls. All six present system calls in SEL4 can be called internally using the slowpath mechanism. Even the ones present in the fastpath mechanism can be called using the slowpath mechanism [32]. The fastpath mechanism acts as a special optimization case present in the SEL4 mechanism to optimize the amount of IPC calls made. This decreases the amount of system calls performed by the system in the SEL4 mechanism as the Reply and Wait calls have been merged as a single system call [57].

4. **Low Level Design Abstraction Level in SEL4-IPC** : The fourth general abstraction level group is the lower level design abstraction level which serves as an interacting layer between the high level design level and the functional code level. The first lower level abstraction component comprises of `SEL4FastPathIPCLLDesign`, `SEL4SlowPathIPCLLDesign`, `SEL4AuthCapIPCLLDesign` and `SEL4OtherLLDesign`. The `SEL4FastPathIPCLLDesign` is subdivided into `SEL4FastPathCallLLD` and `SEL4FastPath ReplyWaitLLD` representing the Call and the Replywait optimization case for the corresponding system calls.

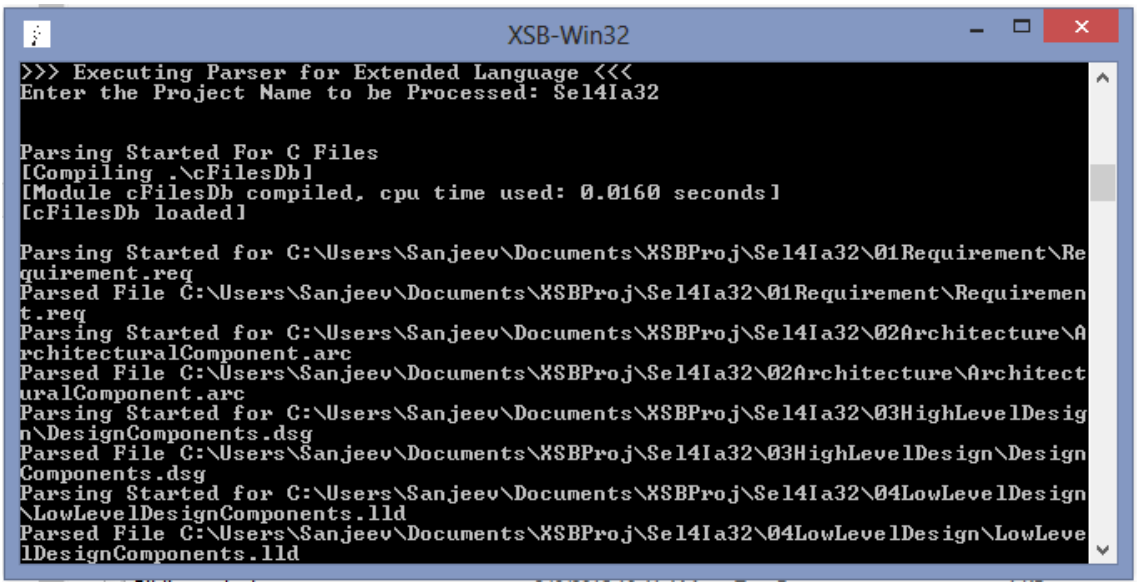


Figure 7.6: A Snapshot of SyModEx2 during Analysis of SEL4

`SEL4AuthCapIPCLLDesign` is further differentiated into `SEL4AuthCapIPCSendLLDesign`, `SEL4AuthCapIPCReceiveLLDesign`, `SEL4AuthCapIPCGrantLLDesign` and `SEL4AuthSetCapIPCLLDesign`.

Klein and colleagues state the following for the design of the fastpath mechanism:

> *"The fastpath targets the case of the Call and ReplyWait system calls where a message will be sent immediately and control will be transferred with the message."* [57].

Klein and colleagues also state the following which supports our design for the IPC authentication mechanism.

> *"The propagation of capabilities through the system is controlled by a take-grant-based model. SEL4 supports three orthogonal access rights, which are Read, Write and Grant."* [32]

7.6 Applying SyModEx2 and Model Statistics for the SEL4-IPC

Firstly, we processed the SEL4 source code with IA32 configuration using SEL4's built-in tool to output a full-blown preprocessed SEL4 microkernel source code for the IA32 model. All the include directives and macro expansions were performed by this tool and we got a C source code file which had some GCC C extensions in it.

Table 7.1: Number of Different Nodes in the SEL4-IPC Model

Abstraction Level	Number of Different Nodes
Top and Bottom	2
Requirements	5
Architectural Components	3
High Level Design Components	4
Low Level Design Components	10
C Code Functions	641
Total Number of Nodes	665

Table 7.2: Number of Different Edges Present in the SEL4-IPC Model

Edge Type	Number of Links Present
Given	1647
Transitive	16,694
SimpleCompleteness	189,544
Total Number of "Implements" Links	207,885

We applied SyModEx2 to verify the SEL4-IPC model. A sample run for running the SEL4 IA32 model using SyModEx2 is presented in Figure 7.6. In this figure, we may observe that only the name of the project needs to be supplied to SyModEx2. SyModEx2 then automatically searches for the compatible source files and starts to process them one at a time.

We derived the model statistics presented in Table 7.1 from the knowledge-base created by SyModEx2. We report a high number of SimpleCompleteness links being generated, showing the tight coupling between components. Table 7.2 illustrates this fact. Since, we only created the model for IPC and IPC related authentication mechanism there are a low number of specification level nodes.

After all the "Given" implementation links have been extracted using SyModEx2, we then move forward with the automated calculation of abstraction levels. The algorithm for this step is discussed in detail in Section 5.8. During the run of *Abstraction Level Calculation Engine*, any direct or indirect recursive links are removed from the call graph.This procedure is further explained in Section 5.7. When we run SyModEx2 on the SEL4 IA32 instance, the abstraction levels have the hierarchical order presented in Figure 7.7.

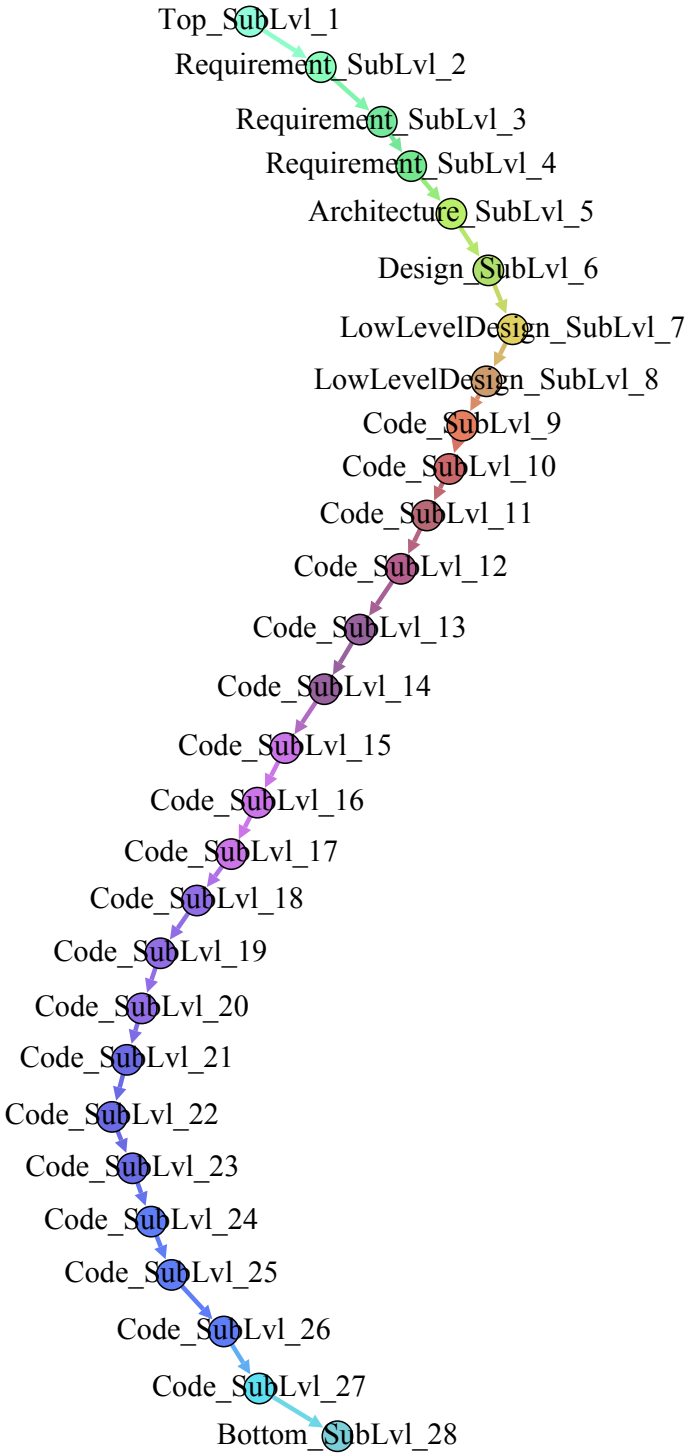


Figure 7.7: Calculated Abstraction Levels Hierarchy for SEL4-IPC Drawn Using Gephi

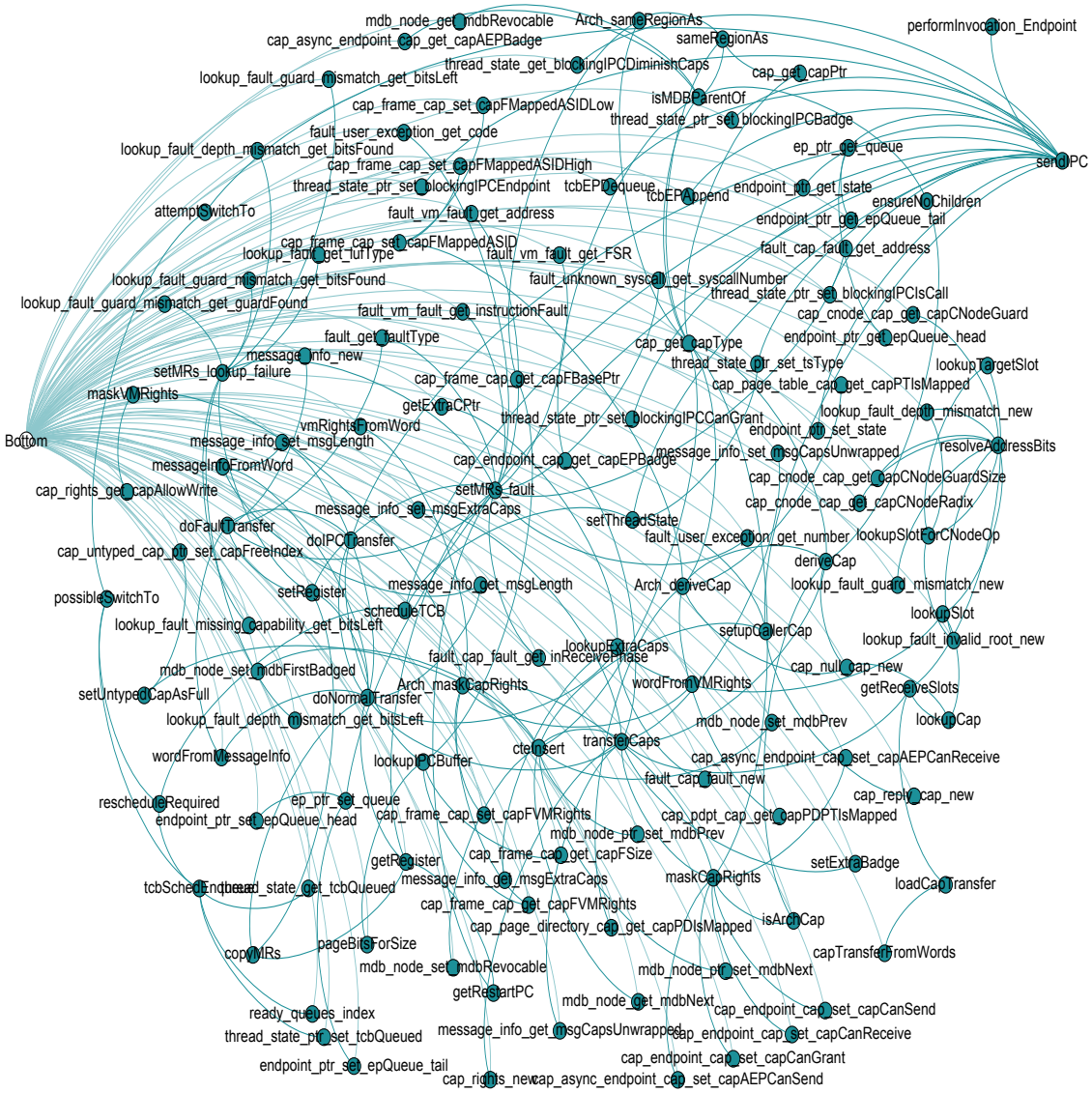


Figure 7.8: Complete Impl. Between sendIPC and performInvocation_Endpoint

7.7 Complete Implementation Verification of the SEL4-IPC Components

For the purpose of verification, we selected the SEL4-IPC subsystem as our focus area in SEL4 microkernel system. Complete implementation verification was performed on these SEL4-IPC components to check if our assumptions about which IPC components needed to be completely implemented were present in the implementation.

Following are the results for the actual implementation verification between selected IPC components.

1. **sendIPC() Completely Implements performInvocation_Endpoint()**: The *sendIPC()* function is a part of the synchronous IPC call chain. Different synchronous IPC calls

such as Call, ReplyWait, Reply are triggered via tha *sendIPC()* function. The *sendIPC()* component is responsible for making the IPC call through the endpoint created. After each IPC call has been properly verified for appropriate rights, the performed IPC call passes through this endpoint.

Elkaduwe et.al. state the following about the *sendIPC()* function:

> *"This function performs an IPC send operation, given a pointer to the sending thread, a capability to an endpoint, and possibly a fault that should be sent instead of a message from the thread."* [37].

Similarly, the *performInvocation_Endpoint()* function is also a part of the synchronous IPC call mechanism. The *performInvocation_Endpoint()* function is responsible for invoking the IPC call and acts as an API level function call. It is called by the higher level functions *decodeInvocation()* and *handleInvocation()* which check the rights of the IPC call and also check for the type of call performed. They are implemented for handling different system call scenarios according to the capability possessed by the invoking object. Since both the functions are part of the synchronous IPC mechanism in SEL4, we assume a direct relationship from *sendIPC()* to *performInvocation_Endpoint()* function.

SyModEx2 verifies the implementation from *sendIPC()* to *performInvocation_Endpoint()* and reports that the relationship between these two components is **Complete**. Figure 7.8 represents the generated chain for the two components. The blue links represent the given links present in the system model.

7.8 Unique Implementation Verification of SEL4-IPC Components

Similarly, Unique implementation verification was performed on these IPC components to check if our assumptions on which IPC components needed to be uniquely implemented were accurate.

Following are the results for the Unique implementation verification between selected IPC components.

1. **SEL4SlowPathIPCLLDesign Uniquely Implements SEL4IPCSlowPathDesign**: The **SEL4IPCSlowPathDesign** represents the high level design of slowpath mechanism followed during a system call. It dictates that the slowpath functionality can handle all system calls performed during execution. The *SEL4SlowPathIPCLLDesign* is the lower

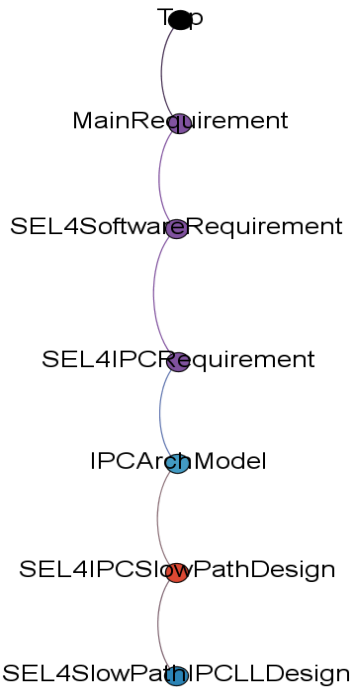


Figure 7.9: Unique Impl. Between SEL4SlowPathIPCLLDesign and SEL4IPCSlowPathDesign

level design model which describes how the slowpath mechanism is going to be implemented as code. It is similar to the pseudo-code description of the slowpath functionality. Here, all the links taken into consideration are the "Implements" links. Both the components considered are the higher level specification nodes and do not include any code components. Hence, we derive our hypothesis that these two higher level specification components are linked together singularly to each other. SyModEx2 reports that the relationship between these two components is *Unique* which proves that our hypothesis is correct.

Figure 7.9 represents the generated singular chain for the discussed two components *SEL4SlowPathIPCLLDesign* and *SEL4IPCSlowPathDesign*.

2. **SEL4FastPathIPCLLDesign Uniquely Implements SEL4IPCFastPathDesign**: The *SEL4IPCFastPathDesign* component relates to the fastpath mechanism's higher level design. It discusses how the fastpath mechanism will supposedly function and what are the specific system calls it will work for during the IPC call execution. The *SEL4FastPathIPCLLDesign* represents the lower level design for the IPC component model. The *SEL4FastPathIPCLLDesign* is further categorized into *SEL4FastPathCallLLD*

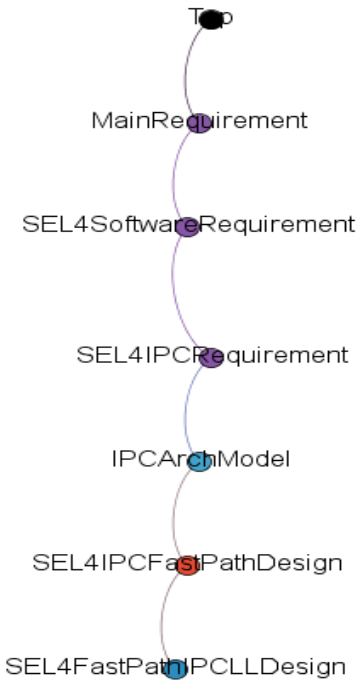


Figure 7.10: Unique Impl. Between SEL4FastPathIPCLLDesign and SEL4IPC-FastPathDesign

and *SEL4FastPathReplyWaitLLD* which represent the `fastpath_call` and `fastpath_ReplyWait` system calls respectively. The *SEL4FastPathIPCLLDesign* is a higher level aggregation of `fastpath_call` and `fastpath_ReplyWait` components. Further information about the design of these components is described in Section 7.5. From this discussion, it becomes clear that the specification between *SEL4IPCFastPathDesign* and *SEL4FastPathIPCLLDesign* is unique. After verification using SyModEx2, we find that the relationship between these two components is *Unique*. Figure 7.10 represents the generated chain for the two components. All the links present are the "Implements" links.

3. **slowpath() Uniquely Implements SEL4SlowPathIPCLLDesign**: The *slowpath()* function is responsible for making all general system calls. Further information regarding both the components is described in Section 7.5. Primarily, our assumption was that *SEL4SlowPathIPCLLDesign* and *slowpath()* were uniquely related. Hence, we assumed that the two components would be related in a Unique fashion. However, using our tool the relationship between these two components was reported as **Non-Unique**. We inspected as to why this condition occurred. After analysis, we found that the *slowpath()*

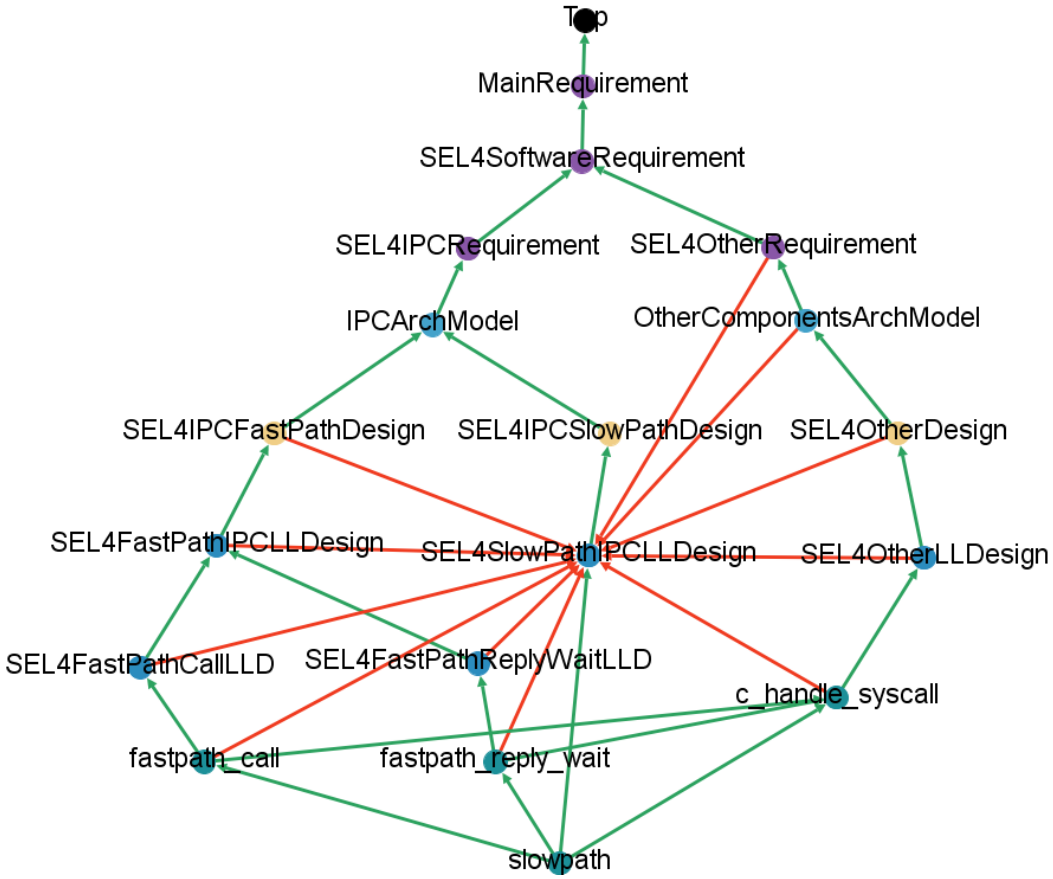


Figure 7.11: Unique Impl. Between slowpath and SEL4SlowPathIPCLLDesign

mechanism was also called by the fastpath mechanism in case of failures or lack of rights. Hence, even though we assumed wrongly that these components were uniquely related; our tool found that this was not a unique relation.

The hidden dependency between the fastpath mechanism and the slowpath mechanism was correctly reported using SyModEx2. A hidden dependency means that there is an alternative method to invoke the *slowpath()* function. This could be a security vulnerability in terms of how threads with lower capabilities may pass from the fastpath mechanism onto the slowpath function and utilize it's functionality. Such escalation of privilege problem could be highly critical for the SEL4 microkernel which has been primarily designed for security.

Figure 7.11 represents the generated chain for the two components. Here, the red links represent the error links that have been found using our tool. The green links represent the given links present in the system model.

7.9 Completeness of Analysis and Threats to Validity

The graph-oriented system model, created in this thesis, is generated using the SEL4 reference manual and different SEL4 related research papers. The SEL4-IPC subsystem modeled in this thesis is a direct mapping of the information presented in these articles. Also, the excerpt of the documentation further justifies the system model generated for the SEL4-IPC subsystem. The node-pairs selected for unique and complete analysis are selected on the basis of their design in the SEL4-IPC subsystem. We believe these node-pairs are important to validate the IPC subsystem and may result in vulnerabilities if not implemented properly.

Another factor we need to stress is the current design of the SEL4 microkernel. If the design of the SEL4 has gone through considerable changes and this was not properly documented then this outline will not be similar to our design of the system. The result will be that the model we have illustrated will be inconsistent with the current system model which could be a threat to our analysis. Hence, the similarity between the implemented model and the system documentation in the SEL4-IPC subsystem is highly important.

Chapter 8

Related Work

This chapter discusses current ongoing research work, categorized into six sections. The first section, describes research efforts for the coexistence of Agile, Formal and Traditional methods. The second section describes the relevance of information traceability for Agile methods. The third section gives an overview of information retrieval-based techniques to identify traceability connections between different system components. The fourth section presents a description of other mechanisms and tools used for the discovery of traceability links. The fifth section describes Massachusetts Institute of Technology's SepcTRM (Specification Toolkit and Requirements Methodology) and Intent Specifications. The sixth section describes two different formal traceability approaches. Finally each section, after the related work has been described, discusses how the approach presented in this thesis compares with the related work.

8.1 Coexistence of Agile, Formal and Traditional Techniques

A common trend seen in today's software industry is the approach to use both agile and traditional software development methodologies. Even though agile is gaining increased popularity in today's software market, the practitioners believe that agile might not be a suitable software development framework in all cases and it does require a suitable organizational culture. A combined effort for software development based on the project's features is presented by Vinekar, Slinkman and Nerur in [88]. In this paper, they describe a ambidextrous approach for the coexistence of agile and traditional approaches.

Vinekar et.al. propose a tightly integrated ambidextrous organization composed of the agile subunit and the traditional subunit [88]. The traditional subunit contains strict hierarchical order, discipline, software process model, bureaucracy and specialized personnel working individually. The agile subunit provides a decentralized flexible architecture, collaborative work environment, evolutionary software development approach and puts customer in a pivotal role. The paper describes a buffered region where each of the two development methodologies are practiced separately free from each others influence, preserving their own culture. Rather than a different method, our approach can be a part of both subunits and check for consistency of the software model using Complete and Unique testing.

Another effort worth mentioning is the work done by Eleftherakis and Cowling [36]. They discuss the importance of correct computer system designed using formal design techniques and addressing changing requirements in today's fast paced industry using agile methods. In their paper, the authors outline a combined approach using both formalism and an agile development methodology called XFun. XFun is created using the formal X-machine system [35, 36, 53] and Unified Process (UP) [59] as a lightweight agile technique. X-machine is a formal method which has the capability of modeling both the data and control of the system and can serve as a possible specification language [36].

In XFun, the user requirements are described in a X-machine model. Then verifications are done on the model and adjustments are done accordingly until we obtain a correct working system [36]. XFun uses several techniques such as communicating X-machine method for enabling modular modeling, the verification and testing strategies and XMDL (X-machine description language) that ensures the development of a reliable, safe, robust and correct system. XFun allows the proponents of agile and formal approach to use a combined approach for creating a safe and secure system which scales and is flexible in the same time. Complete and Unique formal verification can also be added as formal analysis methods in the XFun framework so that it can better help with the verification procedure. For this developers have to create a corresponding X-machine model and encode the specifications and algorithms for the Complete and Unique verification mechanism.

8.2 Identifying Traceability Links in Agile Development Methodology

Agile development techniques have been gaining a lot of popularity because of their notable features such as customer-oriented design, focus on code and testing, self-organizing team structure, etc. With the advent of Agile methodology, agile supporters tend to focus more on the implementation rather than documentation. Hence, some agile proponents argue that traceability is not a needed trait.

A study for implementing traceability in agile software development was first carried out by Jakobsson in 2009 [55]. In this field study, the author surveyed people from various software development companies and investigated why agilists are against the idea of using traceability in their daily work structure. The reasonings presented by these individuals were loss in creativity, belief that Scrum is complete, documents are not read by users and hence a waste, heavy overhead work, hard copy documents which are hard to analyze, bureaucracy and lack

of motivation [55, 66, 79]. However, an idea that agile focuses on is having full working knowledge of the system being developed for which a system model is required; this system model is created in turn using traceability links between different project artifacts. The author provides some practices which can be later used to retrieve traceability in the project such as including traceability information in source code check-in tools, search-able document formats, etc. Jakobsson's work also serves as motivation for developing the SyModEx2 system because it also derives traceability links between system specification and implementation code and creates a system model.

Another research endeavor lead by Antonino and Germann [3] advocates of a non-invasive approach to trace relationship between different project related components. Here, an automated approach to manage and retrieve traceability information is presented using a tool called Traceability Manager or TraceMan. The authors argue that rather than introducing a fully new framework for traceability in the organization, a more practicable approach would be to let the project stakeholder use their present documentation techniques and then automatically retrieve them for analysis. The tool, TraceMan, is capable of identifying the traceability relationship between requirements, architectural specification, source code, test cases and user stories. Some similarities between the TraceMan tool and SyModEx2 is that they both identify different hierarchy level of a system, create a system model from all the traceability links, perform testing for convergence and divergence between the created and implemented model. A set of differences between our tool and TraceMan is that their's cannot calculate the levels of hierarchy at which the project specifications reside while SyModEx2 can, our tool includes a formal testing mechanism for checking the dependency relationship while they use Fraunhofer's SAVE tool [33] to check this and finally our approach makes it possible to make the specifications as a part of the source code which is quite different from their information retrieval from documents approach.

Another related work worth mentioning is that presented by Simko [79]. This approach focuses on adding relevant traceability information during source code check-in sessions. This check-in related information is shown in the JIRA project management tool [40] as a part of the traceability information to a particular user story or test case. A few companies using agile's Scrum methodology have actually used this tool to automatically generate and maintain traceability information. The traceability links are generated between the source code and the "JIRA ticket" specification and this data is used for analysis of improvement areas. Similar to

approaches mentioned above, this approach requires minimal changes in the developer's behavior. Simko's approach is similar to our approach because we both are trying to automatically derive traceability links between source code and specification. We can also write or reference the "JIRA Ticket" by adding a new attribute such as "JIRA TicketID" in EXMPLRAD which would help explain why a particular function was developed. Some differences between SyModEx2 and Simko's tool are that they do not generate a formal system model and they do have a mechanism to add the Complete and Unique verification for crucial system components.

8.3 Identification of Traceability Using Information Retrieval Frameworks

One approach used for discovering traceability links is Information Retrieval (IR) methods [4]. In this method, traceability between different components is identified from the project artifacts using some kind of textual matching algorithms after the assumption has been made that the traceability information is simply not present or needs to be rediscovered [4, 6, 7].

Different versions of probabilistic and statistical information retrieval techniques have been suggested by researchers for finding traceability links present between requirements [7], requirements and design [5], design and source code [16, 17] and source code and project code documentation [4, 6].

Antoniol, et.al [4, 6, 7] apply probabilistic and vector space information retrieval techniques to trace the relationship between C++ source code and developer's manual pages for a software application. They also use a similar approach to perform analysis between the functional requirements of a software application and the Java source code implementing these requirements. The analysis is based on the the intuition that developers use similar mnemonics in the source code (as identifiers) as per the given requirement specification document.

Antoniol, et.al [4] used two IR framework to implement the document indexer and classifier which are responsible for identifier indexing and classifying and ranking these identifiers between documents according to their similarities. A Bayesian classifier was used for the probabilistic IR model. In the case of the vector space IR model, the identifier indexing along with computation of similarities were implemented using Perl scripts.

Research work has also been done by Huffman Hayes, et.al, for derivation of traceability links using information retrieval mechanisms [50, 51]. In their work, they have compared three statistics-based information retrieval methods used to recover traceability links. These methods, namely: retrieval with key-phrases, vanilla vector method and thesaurus retrieval

are used for retrieving the links between components. Their results showed that the thesaurus based techniques proved to superior to their counterparts [50]. In their later work, they have developed a combined approach to these three methods and created a tool named *RETRO* which stands for *REquirement TRacing On-target* [51].

Cleland-Huang, et.al, found dissimilar results from those proposed by Huffman Hayes, et.al. Using thesaurus based methods had no seemingly large improvements over the traceability links data collected. Due to the flaws present in the statistical information retrieval techniques, Cleland-Huang, et.al, introduced three different improvements frameworks [18]. These frameworks are: graph pruning, clustering, and hierarchical. On the course of their research they discovered that clustered grouping of contextual information provided better traceability link generation.

Our approach based on Conte de Leon, et.al, justifies that finding dependencies between components should be carried out in all system development stages [23–25] for all project development strategies. Our approach assumes that there is a systematic methodology to information generation for product artifacts and they are linkable during each project lifecycle defined by various project stakeholders. This will lead to strong connections between specification components and implementation components.

8.4 Tools for Discovering and Modeling Traceability Links

Various new research tools have emerged for the exhaustive discovery of all traceability links present in the system [43]. The SERL (Software Engineering Research Laboratory) have developed fact storage and processing technique to unearthing connections between system components. Van Lepthien, et.al, have developed a tool known as *InfinTe* which is capable of uncovering, creating, storing, and processing the generated connections [2]. "Integrators" which have the capability to generate transitive connection between components is used to create "chained" new links. Our tool performs similar operation as per the partial implementation requirements which gives all the transitive links present in the model. In theory it is possible to implement the partial order and SimpleCompleteness frameworks using the TOOR tools and FOOPS modules and integrate it into the *InfinTe* tool.

Another useful tool for conducting formal verification is through the use of the tool called Xlintkit [70]. This tool developed by Nentwich, et.al, has the ability to search for problems between system components. The backend of this tool is composed of XML and predicate

calculus. This tool can be fully used in conjunction with our own tool to make further formal analysis for the generated system model.

Another approach developed by Egyed is based on finding traceable connections with the use of source code signatures [34]. The tool, named as TraceAnalyzer generates traceability links using source code profiling techniques. After this step further links are added manually using hypothetically connected links. These generated links more closely relate to the "Given" links present in the system model. One main disadvantage of using this tool is that the abstraction levels present in the system are not properly differentiated and the links generated are more coalesced.

Cleland-Huang, et.al, have devised a tool which is based on notifying users to take action as changes are made to the system [16]. The motivation behind this is that when a change is implemented on the system then it's properties change. These changes as a result need to be reflected in the calculation and derivation of traceability links as well. All the new links are a result of manual addition of them and no automated action is performed by the tool. Hence, the tool user is solely responsible for the creation of new connections.

8.5 SpecTRM and Intent Specifications Approach

Leveson, Reese and Heimdahl have prepared a framework for stating the system specifications across various abstraction hierarchies present in the system [62].

SpecTRM-RL (Specification Tools and Requirements Methodology) was created to allow engineers to express specifications more easily and with little effort [62]. SpecTRM-RL also made it possible to generate models using the specifications input by the project developers. These system models were formally testable and verifiable enhancing the safety and security aspect of the system. The tool also contained a visualization suite which enabled different views of the system according to the task being executed.

Intent Specifications is a component structuring specification methodology based on work breakdown ability of humans and solving hard problems using small manageable unit components [62]. The intents can be specified at five different levels : System Purpose, System Principles, Black-box Behavior, Design Representation and Code (Physical Representation). The System Purpose intent is used to specify the functional and non-functional requirements inherent in any system. This intent corresponds to our Requirement level abstraction in our tool. The System Principles intent comprises of alternative architecture models of the system,

design models, etc. This intent closely resembles the Architectural Component hierarchy in our tool. The Black-box behavior refers to black-box functionality model of the system which are executable and formally analyzable. The Design Representation refers to the system design guidelines present in the system. This intent is represented as a combination of the high level design and low level design abstractions. The Code represents the actual code present to represent the working components in the system. This intent is represented by the functions present in C and any further properties are specified using the codeFunction abstraction.

The work presented by Leveson et.al. [62] closely resembles to the work we have done in this thesis. However, some notable differences that are present in our system is comprised of creation of transitive links, automated creation of abstraction levels based on relationship and the formal verification between system modules to ensure Unique and Complete implementation.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The ever-increasing complexity of computer programs introduces new challenges to software engineers for correctly designing and implementing software systems. Even though there are tools for checking and testing the code components in a software program; functionality to test the relationship between the system specifications and code level components is still missing.

Furthermore, different software development methodologies have different views towards system documentation. While the traditional and formal approaches embrace the idea, Agile shuns it. This is because it is hard to keep track of documents, as requirements for the functionality change. With all the products and by-products generated during the project life-cycle, it becomes an increasingly arduous task for the project stakeholders to properly amass and assess the complete workings of a system [63, 67].

Hence, we need effective software development techniques and toolset which is compatible with traditional as well as modern software development methodologies and has the potential to create a system model that can be used to formally verify relationship between functional code and specification components. Our approach to tackle this problem is two-fold. Firstly, we create EXMPLRAD through which we can describe the system specifications and code components and establish relationships between them. Secondly, we retrieve these model relationships, create a self-contained system model and formally verify the Complete and Unique relationship between component pairs. Any missing dependency is reported promptly and the system errors are avoided.

The contributions this thesis provides are:

1. Created an *Extended Modeling and Programming Language with Requirements, Architecture and Design Integration* (EXMPLRAD) for expressing system specifications.

2. Developed a recursive descent top down parser and integrated it to the expert tool *SyModEx2* for parsing the C language as well as the embedded extended language.

3. Generated a holistic system model which includes function definitions and calls related to the C functional code as well as the written specifications using a Semantic Analyzer.

4. Developed an algorithm for the determination and ordering of the abstraction levels that are present in a system model.

5. Formally verified dependencies between described model components using the uniqueness and completeness formal verification techniques.

6. Created graphs in standard formats such as GEXF and DOT language, which can be visualized using visualization tools such as Gephi and CytoScape, for better understanding the system model.

These techniques can help ensure that all dependencies between components present in the system models are mechanically derived. This method also provides a platform where system stakeholders can describe software specifications as code which can be appropriately changed as requirement changes are made. The SEL4-IPC example shows that this tool can be used to automatically derive the system model of a working system. Additionally, the results of this thesis also show how relationship between components can be verified formally with the help of Unique and Complete principles and the SyModEx2 toolset.

9.2 Future Enhancements

We envision four main research tracks where further work would be beneficial. Firstly, full support for all the C language constructs need to be implemented. Secondly, a common language specification mechanism is to be implemented so that other popular programming languages can be parsed and analyzed. Thirdly, it is necessary to examine the various other relationships between the model components so that further information can be gathered and in turn making the tool more effective. Lastly, empirical research needs to be conducted to visualize, filter and report the presented formal traceability analysis into industrial strength open source tools. We describe each of these four individual areas below.

1. **Enhanced Support for the C Programming Language and Preprocessing** : Even though our tool supports all of the C constructs and can parse the preprocessing directives `#include` and `#define` directives present in the program source code; we still need the ability to parse the following additional constructs [49]:

   (a) C Extensions

   (b) C Stringification

(c) Macro Expansion and Substitution

(d) Preprocessor Conditional Evaluation

As discussed in section 4.3, the C language is composed of two distinctly separate languages: the C language and the C preprocessing language [45]. Hence, creating a two pass parser to actually read and evaluate the C preprocessing language constructs and then read the resulting C language constructs is necessary. This step will help us analyze both the C language and C preprocessing language constructs appropriately. Also, we have the advantage that it follows the general processing mechanism of the **gcc** parser, which will help us combine and evaluate the results between our parser and the **gcc** parser. Also, we envision that the parse tree we generate for the C Grammar will be same as that from the **gcc** parser and we have some mechanism to verify the similarity between these trees.

2. **Support for Parsing other Programming Languages** : Many other programming languages exist which are widely being used to create new complex software systems. Hence, a key area which would increase the usability of this tool is by making it compatible with other programming and/or specification languages.

   Furthermore, we envision that we would be able to parse a project which is a combination of multiple languages, which is the case in many modern projects. Being able to collect information from a blend of these languages and extract as well as link information between them will provide the project stakeholders with a clear picture of the whole system. Being able to parse multiple languages will also help the developers community to integrate our tool into many other software tool suites and make the formal checking process flexible and seamless.

   A big advantage would be the freedom it provides to end-users to check and test their applications. The overall compatible application base would widely increase and even miniature or small applications could be tested with this tool. Similar to the SEL4 case-study we could then perhaps analyze crucial Java applications such as android kernel system's OS services which could then be checked for inconsistencies or errors.

3. **Evaluation of Different Link Types Between Interconnected Components** : In this thesis, we outlined the "Given", "Transitive" and "GeneratedBySimpleCompleteness"

links which were related to each other using the "ImplementationDependency". A large number of other such tags and categories can be readily created using the already available knowledge base created by SyModEx2. One thing that we should deeply consider is the different types of relationships that exist within a system model. These link types may provide more valuable information regarding the structure and orientation of the model in terms of data flow, event triggers [60].

These other types of links could help identify the source of errors in the system and also the central components which are crucial in operating the system. For example, different levels of module, sub-module, and file containment can also be visualized using the "contains" link-type.

4. **Further Testing** : Currently, we only test a subset of files from the gcc-testsuite. This is due to time constraints but our overall objective is to complete the testing for all the C files present in the gcc-testsuite. We would like to test SyModEx2 against all the files present in the gcc-testsuite. This will help prove that our parser is complete and free from errors.

5. **Visualization and Filtering of Formal Traceability Results** : Presently, different graphs relating to the "Implements" relationship between components are generated. These graphs are written in DOT language [44] and GEXF [47] format. Currently, GraphViz DOT tool [38] and Gephi [46] are used for graph visualization and manipulation. Limitations for both the GraphViz DOT tool and Gephi have been discussed at length in Section 4.8 and it is seen that these tools are not be well adapted to visualize the system model. Possible reasons are due to their inability to scale for large graphs and lack of graph manipulation operations. Perhaps by incorporating a new powerful graphing tool we can properly filter and visualize the system model and make it easier for the project stakeholders to have a correct and understandable view of the system.

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, Principles, Techniques.* Addison wesley, 1986.

[2] Anderson, K. M., Sherba, S. A., and Lepthien, W. V. Towards large-scale information integration. In *Proceedings on International Conference on Software Engineering* (2002), pp. 524–534.

[3] Antonino, P. O., Keuler, T., Germann, N., and Cronauer, B. A non-invasive approach to trace architecture design, requirements specification and agile artifacts. In *Australian Software Engineering Conference* (2014), pp. 220–229.

[4] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings International Conference on Software Maintenance* (2000), pp. 40–49.

[5] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. Maintaining traceability links during object-oriented software evolution. *Software: Practice and Experience Vol. 31*, 4 (2001), 331–355.

[6] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering Vol. 28*, 10 (2002), 970–983.

[7] Antoniol, G., Canfora, G., Casazza, G., Merlo, E., et al. Tracing object-oriented code into functional requirements. In *Proceedings on International Workshop on Program Comprehension* (2000), pp. 79–86.

[8] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing Vol. 1* (2004), 11–33.

[9] Bastian, M., Heymann, S., Jacomy, M., et al. Gephi: an open source software for exploring and manipulating networks. *International Conference on Web and Social Media Vol. 8* (2009), 361–362.

[10] Beck, K. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[11] Bendersky, E. Some problems of recursive descent parsers. http://eli.thegreenplace.net/2009/03/14/some-problems-of-recursive-descent-parsers, 2009.

[12] Boyton, A. A verified shared capability model. *Electronic Notes in Theoretical Computer Science Vol. 254* (2009), 25–44.

[13] Bratko, I. *Prolog programming for artificial intelligence*. Pearson Education, 2001.

[14] C. Michael, H. Why engineers should consider formal methods. Tech. rep., 1997.

[15] Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. *Non-functional requirements in software engineering*, vol. Vol. 5. Springer Science & Business Media, 2012.

[16] Cleland-Huang, J., Chang, C. K., Sethi, G., Javvaji, K., Hu, H., and Xia, J. Automating speculative queries through event-based requirements traceability. In *Proceedings on IEEE Joint International Conference on Requirements Engineering* (2002), pp. 289–296.

[17] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., and Christina, S. Goal-centric traceability for managing non-functional requirements. In *Proceedings on International Conference on Software engineering* (2005), pp. 362–371.

[18] Cleland-Huang, J., Settimi, R., Duan, C., and Zou, X. Utilizing supporting evidence to improve dynamic requirements traceability. In *Proceedings on International Conference on Requirements Engineering* (2005), pp. 135–144.

[19] Clocksin, W., and Mellish, C. S. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[20] Cockburn, A. *Crystal clear: a human-powered methodology for small teams*. Pearson Education, 2004.

[21] Cohn, M. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[22] Conte de Leon, D. *Formalizing traceability among software work products.* University of Idaho, 2002.

[23] Conte de Leon, D. *Completeness of implementation traceability for the development of high assurance and critical computing systems.* PhD thesis, University of Idaho, 2006.

[24] Conte de Leon, D., and Alves-Foss, J. Hidden implementation dependencies in high assurance and critical computing systems. *IEEE Transactions on Software Engineering Vol. 32*, 10 (2006), 790–811.

[25] Conte de Leon, D., Alves-Foss, J., and Oman, P. W. Implementation-oriented secure architectures. In *International Conference on System Sciences* (2007), pp. 278a–278a.

[26] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms.* MIT press, New York, NY, USA, 2009.

[27] Coyotos. The Coyotos Secure Operating System. http://www.coyotos.org/, 2008.

[28] Crockford, D. The application/json media type for javascript object notation (json). *Internet Engineering Task Force* (2006).

[29] Degener, J. ANSI C grammar, Lex specification. http://www.quut.com/c/ANSI-C-grammar-l-1999.html, 2012.

[30] Degener, J. ANSI C Yacc grammar. http://www.quut.com/c/ANSI-C-grammar-y-1999.html, 2012.

[31] Degener, J. Documentation: Instructions for the builder and engine. http://goldparser.org/doc/index.htm, 2012.

[32] Derrin, P., Elkaduwe, D., and Elphinstone, K. SEL4 Reference Manual. *NICTA-National Information and Communications Technology Australia* (2006).

[33] Duszynski, S., Knodel, J., and Lindvall, M. Save: Software architecture visualization and evaluation. In *European Conference on Software Maintenance and Reengineering* (2009), pp. 323–324.

[34] Egyed, A. A scenario-driven approach to traceability. In *Proceedings on International Conference on Software engineering* (2001), pp. 123–132.

[35] Eilenberg, S. *Automata, machines and languages*, vol. Vol. A. Academic Press, 1974.

[36] Eleftherakis, G., and Cowling, A. J. An agile formal development methodology. In *Proceedings in South-East European Workshop on Formal Methods* (2003), pp. 36–47.

[37] Elkaduwe, D., Klein, G., and Elphinstone, K. Verified protection model of the sel4 microkernel. In *Verified Software: Theories, Tools, Experiments*. Springer, 2008, pp. 99–114.

[38] Ellson, J., North, S., et al. Graphviz-graph visualization software. http://www.graphviz.org, 2008.

[39] Elphinstone, K., and Heiser, G. From L3 to SEL4 what have we learnt in 20 years of L4 microkernels? In *Symposium on Operating Systems Principles* (2013), pp. 133–150.

[40] Fisher, J., Koning, D., and Ludwigsen, A. Utilizing Atlassian JIRA for Large-Scale Software Development Management. In *International Conference on Accelerator & Large Experimental Physics Control Systems* (2013), pp. 505–508.

[41] Fowler, M., and Highsmith, J. The agile manifesto. *Software Development Vol. 9*, 8 (2001), 28–35.

[42] Galin, D. *Software quality assurance: from theory to implementation*. Pearson education, 2004.

[43] Galvao, I., and Goknil, A. Survey of traceability approaches in model-driven engineering. In *IEEE International Enterprise Distributed Object Computing Conference* (2007), pp. 313–313.

[44] Gansner, E., Koutsofios, E., and North, S. Drawing graphs with dot. In *Technical Report AT&T Bell Lab* (2006).

[45] Gazzillo, P., and Grimm, R. Superc: Parsing all of C by taming the preprocessor. In *ACM Special Interest Group on Programming Languages Notices* (2012), vol. 47, pp. 323–334.

[46] Gephi Consortium. Gephi. *Computer program](version 0.8. 2 Beta) http:// gephi.github. io/. Accessed Vol. 14* (2014).

[47] GEXF Working Group. *GEXF 1.2 Draft Primer*, March 2012.

[48] Glinz, M. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference* (2007), pp. 21–26.

[49] GNU. The C Preprocessor. https://gcc.gnu.org/onlinedocs/cpp/, 2015.

[50] Hayes, J. H., Dekhtyar, A., and Osborne, J. Improving requirements tracing via information retrieval. In *Proceedings on International Requirements Engineering Conference* (2003), pp. 138–147.

[51] Hayes, J. H., Dekhtyar, A., and Sundaram, S. K. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering Vol. 32*, 1 (2006), 4–19.

[52] Heymann, S. Sigmajs tutorial. https://github.com/jacomyal/sigma.js/wiki, 2014.

[53] Ipate, F., and Holcombe, M. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics Vol. 63* (1997), 159–178.

[54] ISO/IEC 9899:1999 (E). ISO C99 Definition. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, 1999.

[55] Jakobsson, M. *Implementing traceability in agile software development.* Department of Computer Science, Lund University, 2009.

[56] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. Sel4: formal verification of an operating-system kernel. *Communications of the ACM Vol. 53*, 6 (2010), 107–115.

[57] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems Vol. 32*, 1 (2014), 2.

[58] Knight, J. C. Safety critical systems: challenges and directions. In *Proceedings on International Conference on Software Engineering* (2002), pp. 547–550.

[59] Kruchten, P., and Bonfiglio, F. *Rational unified process: An Introduction*. Addison-Wesley Publishing Company, 2000.

[60] Lamb, L. C., Jirapanthong, W., and Zisman, A. Formalizing traceability relations for product lines. In *Proceedings on Traceability in Emerging Forms of Software Engineering* (2011), pp. 42–45.

[61] Laprie, J. C., and Randell, B. Fundamental Concepts of Computer Systems Dependability. In *Proceedings of the Workshop on Robot Dep., Seoul, Korea* (2001).

[62] Leveson, N. G., Reese, J. D., and Heimdahl, M. P. SpecTRM: A CAD system for digital automation. In *Proceedings on Digital Avionics Systems Conference* (2003), vol. Vol. 1, pp. B52–1.

[63] Levi, K., and Arsanjani, A. A goal-driven approach to enterprise component identification and specification. *Communications of the ACM Vol. 45*, 10 (2002), 45–52.

[64] Linz, P. *An introduction to formal languages and automata*. Jones & Bartlett Publishers, 2011.

[65] Louden, K. C. Compiler construction. *Cengage Learning* (1997).

[66] Meyer, B. The Ugly, the Hype and the Good: An assessment of the agile approach. Springer International Publishing, 2014, pp. 149–154.

[67] Nagano, S., Ichikawa, Y., and Kobayashi, T. Recovering traceability links between code and documentation for enterprise project artifacts. In *Proceedings on Computer Software and Applications Conference* (2012), pp. 11–18.

[68] Nance, R. E., and Arthur, J. D. *Managing software quality: a measurement framework for assessment and prediction*. Springer Science & Business Media, 2002.

[69] NASA Langley Formal Methods. What is Formal Methods? http://shemesh.larc.nasa.gov/fm/fm-what.html, 2001.

[70] Nentwich, C., Capra, L., Emmerich, W., and Finkelsteiin, A. xlinkit: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology Vol. 2*, 2 (2002), 151–185.

[71] Padioleau, Y. Parsing C/C++ code without pre-processing. In *Compiler Construction* (2009), pp. 109–125.

[72] Pohl, K. *Process-centered requirements engineering.* John Wiley & Sons, Inc., 1996.

[73] Pressman, R. S. *Software engineering: a practitioner's approach.* Palgrave Macmillan, 2005.

[74] Rajlich, V. Changing the paradigm of software engineering. *Communications of ACM Vol. 49*, 8 (Aug. 2006), 67–70.

[75] Roch, B. Monolithic kernel vs. Microkernel. *TU Wien Journal* (2004).

[76] Royce, W. W. Managing the development of large software systems. In *Proceedings on IEEE WESCON* (1970), vol. Vol. 26, pp. 328–388.

[77] Sagonas, K., Swift, T., and Warren, D. S. XSB as an efficient deductive database engine. In *ACM Special Interest Group on Management of Data Record* (1994), vol. Vol. 23, pp. 442–453.

[78] Schwaber, K., and Sutherland, J. *Software in 30 days: how agile managers beat the odds, delight their customers, and leave competitors in the dust.* John Wiley & Sons, 2012.

[79] Simko, R. Automating traceability in agile software development. *University of Lund* (2015).

[80] Slonneger, K., and Kurtz, B. L. *Formal syntax and semantics of programming languages*, vol. Vol. 340. Addison-Wesley Reading, 1995.

[81] Sommerville, I. *Software Engineering 10th Edition.* Addison-Wesley, 2011.

[82] Sommerville, I., and Kotonya, G. *Requirements Engineering: Processes and Techniques.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[83] Steinberg, U., and Kauer, B. Nova: a microhypervisor-based secure virtualization architecture. In *European conference on Computer systems* (2010), pp. 209–222.

[84] Sterling, L., and Shapiro, E. Y. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.

[85] Swift, T., Warren, D., Sagonas, K., Freire, J., et al. The XSB System Version 3.2. Volume 2: Libraries, Interfaces and Packages, 2009.

[86] Swift, T., and Warren, D. S. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming Vol. 12*, 1-2 (2012), 157–187.

[87] Team, C. P. Capability maturity model integration for systems engineering/software engineering/integrated product and process development/supplier sourcing, version 1.1 continuous representation. *Software Engineering Institute, CMU/SEI-2002-TR-004* (2002).

[88] Vinekar, V., Slinkman, C. W., and Nerur, S. Can agile and traditional systems development approaches coexist? An ambidextrous view. *Information systems management Vol. 23* (2006), 31–42.

[89] Warren, T. S. D. S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R. F., Saha, D., Dawson, S., et al. The XSB System Version 3.5. x Volume 1: Programmers Manual.

[90] Wong, W. E., Debroy, V., Surampudi, A., Kim, H., and Siok, M. F. Recent catastrophic accidents: Investigating how software was responsible. In *International Conference on Secure Software Integration and Reliability Improvement* (2010), pp. 14–22.