

Simulysis: A Method to Change Impact Analysis in Simulink Projects Based on WAVE-CIA

Hoang-Viet Tran*, Ta Van Thang, Cao Xuan Son, Do Trong Thu, Pham Ngoc Hung
University of Engineering and Technology, Vietnam National University,
Hanoi No. 144 Xuan Thuy Street, Dich Vong Ward, Cau Giay District, Hanoi, Vietnam

Abstract—MATLAB and Simulink, which have more than 5 million users and are installed at more than 100,000 businesses, universities, and government organizations¹, are widely used in numerous large-scale projects across various industries. These projects continually evolve in response to changes in business logic. However, managing the impact of these changes on Simulink projects presents several challenges to guarantee the quality of these projects. To address this, we propose a WAVE-CIA-based method named Simulysis for change impact analysis (CIA) in Simulink projects. The core idea behind Simulysis is to directly analyze Simulink project files and construct the project's corresponding call graph. By comparing the call graphs from the old and new project versions, Simulysis computes the change set. Subsequently, Simulysis applies the WAVE-CIA method to this change set and the call graph to identify the impact set. Additionally, Simulysis proposes a signal tracing method helping the system engineers to follow, check, and debug signals through the system. We have implemented Simulysis as a tool with the same name and conducted experiments using several open-source Simulink projects. The experiments demonstrate that Simulysis effectively performs the CIA process and retrieves the impact set, producing optimistic results and proving the practical applicability of Simulysis for real-world projects. Further discussions about Simulysis are provided in the paper.

Keywords—Change impact analysis; WAVE-CIA; MATLAB/Simulink projects

I. INTRODUCTION

In the development of a Simulink project, where the focus is on modeling, simulating, and analyzing dynamic systems, the evolution of requirements and functions is a common occurrence. Factors such as adding new features or accommodating customer preferences often drive changes. Similar to the software development life cycle, even minor adjustments in Simulink models can lead to unforeseen consequences and defects in the system. Consequently, change requests pose challenges for maintaining and managing Simulink projects. For developers, devising a strategy to identify the modules that require modification to accommodate changes is complex. Testers face the burden of testing numerous test cases after system modifications. Project managers grapple with cost allocation and time management related to change management. For this reason, understanding the relationships between altered Simulink components and their impact on other elements is crucial for Simulink projects and software projects in general. To address this, Change Impact Analysis (CIA) [1], [2], has been developed and applied. Numerous

published papers discuss this solution [3], [1], [4], [5], [6], [2].

Researchers have indeed widely employed CIA across various systems, spanning different research fields and programming languages. In the domains of Business Process Management (BPM) and Service-Oriented Architecture (SOA), numerous published papers addressing CIA problems reveal that impact analysis solutions primarily fall into three categories: dependency analysis, traceability analysis, and history mining [4], [7]. First, dependency analysis is most frequently used. It encompasses various graph-based analyses, including control flow graph analysis [8], [9], message dependency graph [10], [11], trust dependency graph [12], directed hypergraph [13] and call graph analysis. Additionally, formal methods aid in change identification, formal modeling, and impact analysis [14], [15]. Quantitative analysis, which quantifies the depth of change using metrics, is essential for decision-making and problem-solving in service-based systems [16], [17], [18], [19], [20]. Lastly, execution trace analysis relies on information retrieved during program execution, either on-the-fly (online) or after program execution (offline) [21].

Following dependency analysis, traceability analysis plays a crucial role. It helps establish relationships between components and their impact on other elements, such as source code [22]. These relationships span multiple abstraction layers and are captured using traceability links. Lehnert's method enables automated or semi-automated impact analysis approaches for software, although it is not suitable for manual analysis [9]. Existing traceability can be categorized into vertical and horizontal traceability [23], [24]. Given the distributed nature of SOA, traceability analysis in SOA systems remains a challenging task.

History mining refers to the use of Mining Software Repositories (MSR) techniques to discover historical change dependencies [25], [21], [26]. This approach relies on analyzing the version histories of software to identify change patterns within source code repositories. By examining these patterns, the method can pinpoint processes or services that have frequently changed in tandem. It leverages change logs or version histories to predict the impact of future modifications. MSR methods have shown significant potential for impact analysis in software systems [27]. They have been applied across various aspects of software projects, including process model repositories [25], web service ecosystems [28], and process choreographies [29].

Change Impact Analysis (CIA) methods have been extensively implemented in various software languages. For instance, the CA Harvest Software Change Manager is a

¹<https://www.mathworks.com/content/dam/mathworks/fact-sheet/2023-company-factsheet-8-5x11-8282v23.pdf>

comprehensive tool designed to track and manage software changes throughout the development lifecycle [30]. It supports C/C++ and several mainframe server languages like COBOL, PL/I, and Assembly, offering features that help organizations manage and understand the impact of their source code modifications. JRipples, an open-source tool tailored for Java developers, aids in comprehending programs during source code changes [31]. It manages steps for impact analysis and change propagation, highlights dependencies, and assists developers in understanding how changes in one part of the source code can affect another. JArchitect, although based on CIA, exclusively supports Java [32]. It's a source code analysis and visualization tool that helps developers enhance their source code. It offers a comprehensive set of features, including source code metrics, visualization of dependencies through directed graphs and dependency matrices, and the ability to compare source code snapshots. Lastly, several tools were implemented, including various CIA methods. SonarQube supports over 30 languages, frameworks, and Infrastructure as Code (IaC) platforms [33]. Checkmarx supports Java, C#, JavaScript, Python, Ruby, and PHP [34]. Ansible supports specific domain-based languages on YAML [35]. Jira provides an interface for collaboration and project management [36].

Regarding the CIA for Simulink projects, in 2017, Rapos et al. introduced a method to address the Change Impact Analysis (CIA) problem in Simulink projects [5]. This method, based on a set of MATLAB scripts, was implemented in a tool named SimPact. SimPact was validated as an impact predictor for the maintenance history of a large set of industrial models and their tests. However, SimPact can only operate within the Simulink Interactive Development Environment (IDE), meaning it cannot function independently of MATLAB. Other tools, such as Diffplug and BDT, support the impact analysis of Simulink projects using the signal tracing method [37], [6]. Despite these advancements, no published paper has yet addressed the CIA problem of Simulink projects that can operate as a standalone solution and utilize the WAVE-CIA method for performing the CIA.

This paper introduces Simulysis, a method based on WAVE-CIA [3], designed to identify and analyze the impact of changes between two versions of Simulink projects. Simulysis reads data from the project's *.mdl and *.slx files and builds the corresponding call graphs. It then compares thoroughly, identifying discrepancies and modifications between the two project graphs. Subsequently, Simulysis applies the WAVE-CIA to the changed Simulink components to identify the affected ones. Simulysis makes four main contributions. First, it introduces a technique for extracting information from Simulink files and classifying the changes in different project versions. Second, it demonstrates how to generate a call graph of the project, a feature not supported by the current version of MATLAB/Simulink. Third, Simulysis utilizes the WAVE-CIA method to analyze the impact of changes using the generated call graph. Finally, Simulysis proposes a method to trace a given signal in the provided Simulink project. This provides a crucial method for Simulink engineers to follow, check, and debug signals and signal-related bugs. We have implemented Simulysis as a tool and conducted preliminary experiments with promising results.

The structure of this paper is as follows. Section II intro-

duces the essential background concepts. Section III delves into the details of the Simulysis methodology. Section III-B4 presents the details of the proposed signal tracing method. Section IV provides insights into the practical implementation of the Simulysis Tool. Initial experimental findings and subsequent discussions are presented in Section V. Section VI discusses the published papers related to Simulysis. The paper concludes in Section VII.

II. BACKGROUNDS

In this section, we introduce background concepts that will be utilized in this paper.

A. Change Impact Analysis

Change Impact Analysis (CIA) [2] is a systematic approach commonly employed in software development, project management, and system engineering. Its primary purpose is to evaluate the consequences of modifications made to a system. CIA aids in making well-informed decisions, minimizing the introduction of errors, and optimizing resource utilization. While it cannot guarantee complete predictability or replace thorough testing in every instance, the benefits of CIA include improved system reliability, reduced change risks, and efficient resource allocation. In software projects, CIA benefits software developers, testers, project managers, system architects, and stakeholders involved in the change process. Overall, the CIA process contributes to better decision-making and enhances system quality.

B. WAVE-CIA

WAVE-CIA, introduced by Li et al. in 2013 [3], represents a novel Change Impact Analysis (CIA) approach based on the traditional call graph-based techniques. Drawing inspiration from the propagation of water waves, this method computes a set of impacted components based on a change set and a call graph. Much like the ripples formed when stones are thrown into the water, WAVE-CIA views the call graph as a water surface and the change set as the thrown stones. The central concept is that a component is more likely to be impacted if it has dependencies with altered components, and these modified components propagate the impact to others in the graph. By meticulously analyzing the call graph, WAVE-CIA identifies precise change ripples and adeptly handles multiple changes with interference. In comparison to traditional call graph-based CIA methods, WAVE-CIA achieves a more accurate impact set with fewer false positives, although it may occasionally miss a few false negatives.

In the context of Simulink, several terms and concepts differ from those found in conventional software development codebases. These terms have been mapped to suitable WAVE-CIA equivalents for implementation. The Simulink project comprises numerous Simulink files with extensions such as *.mdl or *.slx. These files contain Simulink systems, which encompass various block types including Ports, Logic, References, Subsystems, and more, all interconnected by lines.

Definition 1 (Node or Vertex). *Nodes represent systems in a Simulink project. Each node corresponds to a specific system, and the edges connecting the nodes represent the caller and*

callee relationships between these systems. This helps define how systems interact with each other in the project.

Definition 2 (Call Graph). A call graph is a directed graph $G = (V, E)$, where V is a set of vertices representing Simulink components in the system, and $E \subseteq V \times V$ is a set of edges representing the call relationships between callers and callees. WAVE-CIA draws an analogy from water waves that ripple outward when stones are thrown into calm water. In the context of the call graph, which is akin to a water surface, Simulink components reachable by multiple altered components (meeting positions) are more likely to be impacted by the changes.

Definition 3 (Change Set - CS). A change set refers to a group of Simulink files and systems within a Simulink project that have been modified or are marked for modification. This set is typically identified by comparing data between two versions of the same Simulink project.

Definition 4 (Core Set). The core set is comprised of closely related Simulink components that are likely to be affected by multiple modifications in the change set. This set is generated by identifying Simulink components close to the change set within a call graph. These components, which are accessible by multiple modified components, are more susceptible to changes. The core set is determined by verifying if a component is influenced by more than one modified component, and it encompasses all the modified components in the change set. The likelihood of components in the core set being affected by multiple changes is high. Subsequently, the impact set is calculated using propagation analysis based on the core set.

Definition 5 (Impact Set - IS). The impact set is made up of Simulink files and systems that could be affected by the Simulink components in the change set. It includes elements that may need to be reworked or revalidated. WAVE-CIA determines the impact set through the process of mining the call graph.

C. Simulink

Simulink [38], a graphical extension to MATLAB [39] developed by MathWorks, is designed for modeling and simulating dynamic systems. It offers key advantages such as the capability to model nonlinear systems, which is not easily achievable with transfer functions, and the ability to include initial conditions for a more accurate depiction of system behavior. Simulink provides a graphical interface for users to build systems using block diagrams, incorporating various elements like transfer functions, adding nodes, virtual input/output devices, etc. Its integration with MATLAB ensures smooth data transfer between the two platforms. Due to its wide usage in system modeling, controller design, and simulation, Simulink is a potent tool in engineering and scientific applications.

Table I describes some common blocks used in Simulink, and Table II shows the data from the source code of Simulink blocks.

Additionally, we clarify the key terms used in Simulink modeling to ensure consistency throughout the paper. The terms *system*, *subsystem*, *component*, *block*, and *file* are fundamental to Simulink, but they have specific meanings and

relationships that must be clearly understood.

Definition 6 (System). A system in Simulink is a complete model representing a specific dynamic behavior or a control logic. It encompasses all the elements necessary for simulating a particular functionality. A system can be a standalone model or can include multiple subsystems and components working together.

Example: A vehicle dynamics model that includes the engine, transmission, and braking systems.

Definition 7 (Subsystem). A subsystem is a modular, hierarchical component of a system. It encapsulates a set of blocks that perform a specific function within the larger system. Subsystems help organize and manage complex models by breaking them into smaller, more manageable parts.

Example: Within a vehicle dynamics system, the engine control unit can be a subsystem that handles all the engine-related computations and control logic.

Definition 8 (Component). A component in Simulink refers to any reusable, self-contained unit within a model. Components can be atomic or composite and can represent hardware or software elements. Components often refer to reusable libraries or pre-built blocks that can be integrated into various models.

Example: A PID controller block can be a component used in various systems to manage control processes.

Definition 9 (Block). A block is the fundamental building element in Simulink models. Blocks represent mathematical operations, signal routing, or functional units. They can be combined to form systems and subsystems. Blocks can be simple (like a gain block) or complex (like a Simulink Function block).

Example: A summation block that adds input signals together is a simple block, while a Stateflow chart is a more complex block.

Definition 10 (File). A file in the context of Simulink refers to the physical storage of models and components. Simulink models are typically saved with the .slx or .mdl file extensions. These files contain all the necessary information to define and simulate a model, including block diagrams, parameters, and configuration settings.

Example: `vehicle_dynamics.slx` is a file that contains the complete Simulink model for vehicle dynamics.

D. Relationships in Simulink Project

To implement the CIA analysis of the Simulink projects, we have to identify the relationships between components and elements of Simulink projects.

1) *Relationships between files:* Within the context of file relationships, we establish connections based on the structure of the files in the project. This is essential for the implementation of WAVE-CIA, which depends on comprehending the dependencies among files. The various types of file relationships are illustrated in Table III.

TABLE I. COMMONLY USED SIMULINK BLOCKS







ID	Icon	Name	Description
1		Logical Operator	The Logical Operator block does the given logical operation on its inputs. If it is nonzero, the input value is true (1). Otherwise, it is false (0).
2		Relational Operator	The Relational Operator block does the given relational operation on the input. The value chosen for the Relational operator parameter determines if the block accepts one or two input signals.
3		Inport	Inport blocks in Simulink link signals from outside a system into the system.
4		Output	Output blocks in Simulink link signals from inside a system to a destination outside of the system. Output blocks can connect signals originating from a subsystem to other parts of the model. Output blocks can also supply external outputs at the top level of a model hierarchy.
5		Subsystem	A Subsystem block is a block that contains a subset of a model. We can have multiple subsystems in a model, and we can have subsystems inside other subsystems.
6		Constant	The Constant block of Simulink generates a real or complex constant value signal. We can use this block to provide a constant signal input.

Fig. 1 illustrates the relationships between various files. The file under examination in this figure is “Logic111”. It is referenced within “Function12”, which makes “Function12” the caller of “Logic111”. Within the context of “Logic111”, there are references to two other files: “Block1111” and “Logic112”. This makes these two files the callees of “Logic111”.

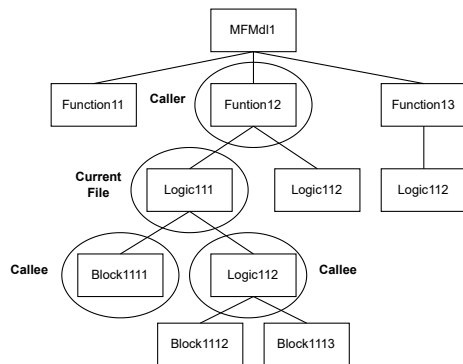


Fig. 1. Example of file relationships.

2) *Relationships between systems:* Within the context of Simulink systems relationships, we establish connections according to the structural organization of systems within a file. This definition is crucial for implementing WAVE-CIA.

Fig. 2 illustrates the relationships among systems. The Subsystem block “Function21” is currently under assessment for its relationships. It has two callers connected to it: the “Bit Set” block and the Inport block “In2_1”. The output of “Function21” is directly linked to the Logical Operator block AND, which is therefore its callee.

E. Types of Change

In the process of analyzing two versions of a Simulink project, pinpointing the differences between them is crucial,

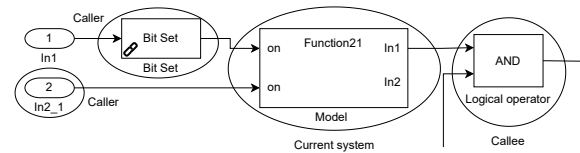


Fig. 2. Example of system relationships.

with a special emphasis on the modified components. The set of these modified elements is termed the change set (CS). Modifications are classified into three categories: addition, deletion, and change.

Simulysis employs the “Name” property of a Simulink system as the key to discern whether it has been added, deleted, or changed. This is a critical aspect of Simulysis, given that two systems within the same Simulink file cannot possess identical names. If a system name is present in both versions of the Simulink project, it is considered altered if there exists any discrepancy in its properties. Table V presents the various types of modifications in a Simulink project.

III. SIMULYSIS METHOD

A. Simulysis Overview

Simulysis is a method based on WAVE-CIA for performing CIA on Simulink projects. This method encompasses three primary phases in the analysis of a specified Simulink project. An overview of Simulysis is depicted in Fig. 3.

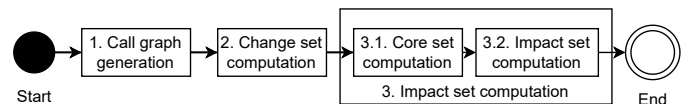








Fig. 3. An overview of simulysis.

TABLE II. SOURCE CODE OF SIMULINK BLOCKS

ID	Icon	Name	Source code
1		Logical Operator	<p>Block source code:</p> <pre><Block BlockType="Logic" Name="Logical&#xA;Operator" SID="12"> <PortCounts in="2" out="1"/> <P Name="Position">[580, 152, 610, 183]</P> <P Name="ZOrder">12</P> <P Name="AllPortsSameDT">off</P> <P Name="OutDataTypeStr">boolean</P> </Block></pre> <p>Properties:</p> <ul style="list-style-type: none"> - Position - ZOrder - AllPortsSameDT - OutDataTypeSt
2		Relational Operator	<p>Block source code:</p> <pre><Block BlockType="RelationalOperator" Name="Relational&#xA;Operator" SID="16"> <PortCounts in="2" out="1"/> <P Name="Position">[270, 257, 300, 288]</P> <P Name="ZOrder">16</P> <P Name="Operator">&lt;=</P> <P Name="InputSameDT">off</P> <P Name="OutDataTypeStr">boolean</P> <P Name="RndMeth">Simplest</P> </Block></pre> <p>Properties:</p> <ul style="list-style-type: none"> - Position - ZOrder - Operator - InputSameDT - OutDataTypeStr - RndMeth
3		Inport	<p>Block source code:</p> <pre><Block BlockType="Inport" Name="In1" SID="10"> <P Name="Position">[370, 153, 400, 167]</P> <P Name="ZOrder">10</P> </Block></pre> <p>Properties:</p> <ul style="list-style-type: none"> - Position - ZOrder
4		Outport	<p>Block source code:</p> <pre><Block BlockType="Outport" Name="Out1" SID="14"> <P Name="Position">[730, 158, 760, 172]</P> <P Name="ZOrder">14</P> </Block></pre> <p>Properties:</p> <ul style="list-style-type: none"> - Position - ZOrder
5		Subsystem	<p>Block source code:</p> <pre><Block BlockType="SubSystem" Name="Subsystem" SID="19"> <PortCounts in="1" out="1"/> <P Name="Position">[565, 229, 665, 271]</P> <P Name="ZOrder">19</P> <P Name="ContentPreviewEnabled">on</P> <System Ref="system_19"/> </Block></pre> <p>Properties:</p> <ul style="list-style-type: none"> - Position - ZOrder - ContentPreviewEnabled - System Ref.
6		Constant	<p>Block source code:</p> <pre><Block BlockType="Constant" Name="Constant" SID="3"> <P Name="Position">[380, 60, 410, 90]</P> <P Name="ZOrder">3</P> </Block></pre> <p>Properties:</p> <ul style="list-style-type: none"> - Position - ZOrder

- Phase 1 (Call graph generation): Given a project, Simulysis obtains the project data by analyzing the project files. Next, it builds a call graph based on the structure and relationships between the Simulink systems (blocks) and files by using the callee map. A

callee map is a set of Simulink components called by other components through the Outports. This phase is vital as it establishes the interconnections within the Simulink project and generates a call graph that will be utilized in subsequent phases.

TABLE III. FILE RELATIONSHIPS

ID	Relationship	Roles	Description
1	Parent	Caller	A file is deemed the parent and caller of another file when it contains references to that other file.
2	Children	Callee	A file is considered a child and callee of another file when it is referred to in the content of that other file.
3	No Relation		Two files are considered unrelated if they don't share a parent-child relationship, regardless of whether they have common parents or children.

TABLE IV. SYSTEM RELATIONSHIPS

ID	WAVE-CIA	Description
1	Caller	The system that triggers a call to another system is referred to as the "caller".
2	Callee	The system that receives a call from another system is referred to as the "callee".
3	No Relation	Two systems are considered unrelated if they are not directly connected.

TABLE V. TYPES OF CHANGE

ID	Change Type	Description
1	Addition	Simulink components that appeared in the new version but not the old version of the project are listed as Additions.
2	Deletion	Simulink components that are removed in the new version and existed in the old version of the project are listed as Deletions.
3	Change	Simulink components that still exist in both versions of the Simulink project but have different properties are listed as Changes.

- Phase 2 (Change set calculation): When provided with two call graphs corresponding to two versions of a Simulink project, Simulysis calculates the change set. This computed change set is then utilized in Phase 3 to determine the impact set.
- Phase 3 (Impact set computation): In this phase, the WAVE-CIA algorithm is implemented, using the change set from Phase 2 as an input. This process identifies the core set, relying on the call graph from Phase 1, and computes the impact set by implementing the WAVE-CIA algorithm. It's important to note that while Simulysis utilizes the WAVE-CIA technique for calculating the impact of changes, it's possible to employ any other CIA method that facilitates the computation of the impact set based on the call graph.

The subsequent sections delve into a comprehensive examination of each phase in the Simulysis method for Change Impact Analysis (CIA) of Simulink projects.

B. Call Graph Generation

There are many methods for the analysis of Simulink projects, such as manual navigation through the signals of the project [37], based on MATLAB-provided script [5], AI-based change distribution analysis [40], etc. By modeling a given Simulink project as a call graph, we have an efficient method to analyze, fix bugs, improve the model, trace the required signals, etc. In this paper, we employ the call graph for change impact analysis of Simulink projects for the two reasons below:

- Signal tracing: Simulink projects are designs of systems with many signals running across the model. By using the call graph to model the given project, signals can be seen as paths through the graph. For this reason, the signal tracing problem can be addressed by using the graph traversing problem and using such well-known algorithms as depth-first search, breadth-first search, etc. This greatly contributes to the system analysis and debugging of Simulink projects.
- Change impact analysis: By modeling a Simulink project as a call graph, the change impact analysis problem can be solved by applying such graph-based CIA methods as the WAVE-CIA method [3].

To create the call graph for a given Simulink project, Simulysis analyzes project files and generates the required call graph using the callee map. The creation of a call graph for a specified Simulink project involves the following three primary steps, as shown in Fig. 4.

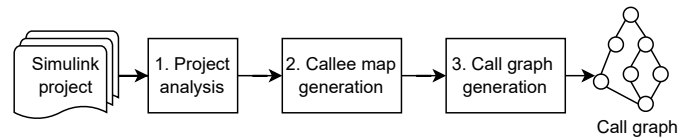


Fig. 4. The main steps of the call graph generation process.

- Step 1 (Project Analysis): Simulysis scans all the Simulink files in the specified project and extracts data about all existing systems, lines, and branches. Details of this step are presented in Section III-B1.
- Step 2 (Callee Map Generation): For every system within the project, Simulysis identifies other systems that are linked to it by employing a signal tracing algorithm. A system in Simulink serves as a computational unit that accepts input data via in-ports, processes this data, and generates output through the out-ports. Simulysis aims to locate the callees of the system, which are systems connected to the system's out-ports. The specifics of this step are illustrated in Algorithm 1.
- Step 3 (Call Graph Generation): Simulysis cycles through each system in the project, generating a new corresponding node in the call graph. Subsequently, Simulysis establishes connections to other nodes in the graph, utilizing the callers and callees of the system identified in Step 2. The specifics of this step are demonstrated in Algorithm 2.

1) *Project analysis*: For a specific Simulink project, Simulysis analyzes by scanning the list of Simulink files in the project and extracting a list of systems (blocks) and the lines that connect these systems.

The structure of a typical Simulink project and a Simulink file is depicted in Fig. 5 and Fig. 6. As a Simulink project comprises numerous files and folders, it is crucial to pinpoint the files where pertinent data is stored. Depending on the MATLAB version of the project, the primary information of a Simulink project is housed in various file types such as ".mdl"

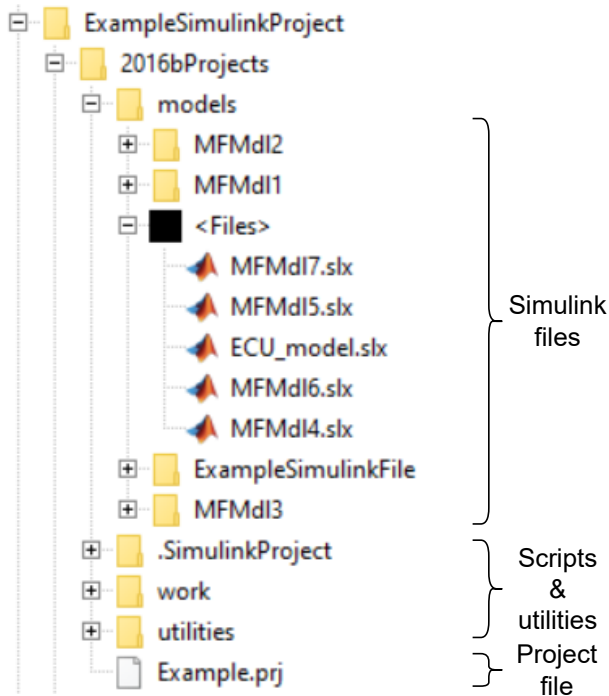


Fig. 5. Simulink project structure example.

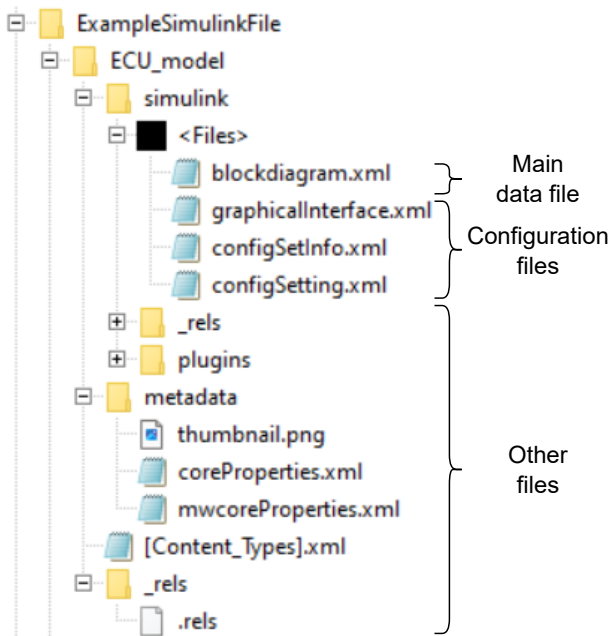


Fig. 6. Simulink file structure example.

and “.slx”. Simulysis filters the project by verifying the validity of the model state and the file extension to locate files with the “.mdl” or “.slx” extension and other project information.

Before MATLAB version R2016b, Simulink files had the “.mdl” extension, and the data could be accessed directly. For newer versions, Simulink introduces a new “.slx” file extension, which is a zip file comprising multiple files and folders. To access the file, Simulysis alters the “.slx” file extension to “.zip” and then unzips the file. As only the

main data file (“blockdiagram.xml”) is needed to recreate a visualization and execute other functions, only this file is analyzed to retrieve system and line information.

Fig. 7 presents an instance of a “blockdiagram.xml” file within an “.slx” file. The file adheres to a standard XML format where the initial tags represent the file configuration and setting parameters. Simulysis utilizes only a few properties within these tags to construct the necessary call graph. The primary data is housed within the `<System>` tag, which encompasses the information of blocks (`<Block>` tag) and lines (`<Line>` tag). Leveraging the information from the blocks and lines, Simulysis assembles the required call graph for use in subsequent phases.

```
<?xml version="1.0" encoding="utf-8"?>
<ModelInformation Version="1.0">
  <Model>
    <P Name="LastSavedArchitecture">win64</P>
    <ConfigManagerSettings>
      .....
    </ConfigManagerSettings>
  </Model>
  <System>
    <Block BlockType="SubSystem" Name="MFMdl1" SID="51">
      <System>
        <P Name="Location">[-8, -8, 1374, 695]</P>
        <P Name="Open">off</P>
        <Block BlockType="From" Name="From" SID="162">
          <P Name="Position">[195, 121, 235, 149]</P>
          <P Name="ZOrder">104</P>
        </Block>
        .....
      </System>
    </Block>
    <Line>
      <P Name="ZOrder">68</P>
      <P Name="Src">327#out:1</P>
      <P Name="Dst">164#in:1</P>
    </Line>
    .....
  </System>
</ModelInformation>
```

Configuration and setting parameters

Block data

Line data

Main data of Simulink file

Fig. 7. Simulink file content example.

2) *Callee map generation*: The purpose of the callee map is to identify the subsystems (i.e., callees) associated with each system in the Simulink project. This is an important step in creating the call graph, which helps analyze call relationships between systems and determine the impact of changes in the project. Details of the callee map generation process are shown in Algorithm 1.

The algorithm starts by initializing the map (*systemCalleeMap*) as an empty set (\emptyset) (line 2). It then iterates over each system, checking its type to determine the appropriate process (lines 3 - 30).

If a system is of type “From”, it does not connect to other systems via lines. Instead, it directly jumps to a system of the “To” type with the same identifier, facilitating data transfer from the “From” system to the “To” system. Consequently, the algorithm identifies the corresponding “To” systems of the current “From” system and appends them as a caller-callee pair to the “*systemCalleeMap*” map (lines 4 - 6).

If a subsystem is then connected to another system, the algorithm will find the block corresponding to the port type and port number connected (lines 12 - 13). In Simulink, a system can connect to output port of type “Output”, “LConn” and “RConn” (in which “LConn” and “RConn” represents left and right side physical connection port - a special port type in Simulink). The corresponding block that links a signal from inside the subsystem to outside the subsystem is of type “Outport”, “PMIOPort with “Side” attribute equals “Left”, “PMIOPort with “Side” attribute equals “Right”.

Algorithm 1 Generate the Callee Map

Input: *systemList*: List of systems of the project

lineList: List of lines of the project

Output: The required callee map

```
1 begin
2   systemCalleeMap ← ∅
3   foreach system in systemList do
4     if system type is "From" then
5       calleeSystem ← findGoto(systemList, system)
6       systemCalleeMap.append(system, calleeSystem)
7     else
8       lines ← findLines(lineList, system)
9       foreach line in lines do
10        callerSystem ← system
11        calleeSystems ← ∅
12        if system type is "SubSystem" then
13          callerSystem ←
14            getOutputPortBlock(system, line)
15          if line has branches then
16            foreach branch in line.getBranches() do
17              calleeSystem ← findSystem(systemList,
18                branch)
19              if calleeSystem type is "SubSystem" then
20                calleeSystem ←
21                  getInputPortBlock(calleeSystem, line)
22              calleeSystems ← calleeSystems ∪
23                calleeSystem
24            end
25          else
26            calleeSystem ← findSystem(systemList, line)
27            if calleeSystem type is "SubSystem" then
28              calleeSystem ←
29                getInputPortBlock(calleeSystem, line)
30            calleeSystems ← calleeSystems ∪
31              calleeSystem
32          end
33        end
34        systemCalleeMap.append(callerSystem,
35          calleeSystems)
36      end
37    end
38  end
39  return systemCalleeMap
40 end
```

After finding out the corresponding output port blocks, the algorithm assigned it as the caller (line 13)

If the system type is not "From", the algorithm identifies lines connected to it. In Simulink, a line block contains some branch properties, which are lines connecting to other blocks. Therefore, for each line, the algorithm checks if the line contains any branches. If so, the algorithm iterates through the branches to find the system that is connected to them and adds them to the callee list (*calleeSystem*) (lines 14 - 20). If not, the algorithm finds the system connecting to the line and adds it to the callee list (*calleeSystem*) (lines 21 - 26).

If a system is connected to a system of type "SubSystem" through the subsystem's input ports (lines 17 - 18 and 23 - 24), the algorithm will find the port block inside the subsystem correspond to port type and port number connected. In Simulink, a system can connect to an input port of type "Input", "LConn" and "RConn". The correspond block that links signal from outside the subsystem to the inside are of type "Inport", "PMIOPort" with "Side" attribute equals "Left", "PMIOPort" with "Side" attribute equals "Right". After finding all corresponding input port blocks, the algorithm then and adds them to the "calleeSystems" array (line 19 and 25).

After finding out the caller and the callee systems,

the algorithm adds the caller system (*callerSystem*) and its corresponding callee list (*calleeSystems*) to the *systemCalleeMap* map (line 27). Once all systems in *systemList* have been processed, the algorithm returns *systemCalleeMap*.

Time Complexity Analysis

Theorem III.1. Let n be the total number of systems in *systemList*, m be the maximum number of lines in the *lineList* and k be the maximum number of lines in the *line.getBranches()*. The time complexity of Algorithm 1 is $O(n * m * k)$.

Proof: From Algorithm 1, we see that the *foreach* loop at lines 3 - 30 iterates n times, resulting in time complexity of this loop is $O(n)$. Lines 4 - 6 check the type of each system. If the *systemType* is "From", its find the corresponding "To" system. This check has a time complexity $O(1)$ for each system. Line 8 finds *lines* connected to the system, which has time complexity $O(1)$ for each system. Line 9 - 28 iterate through each line connected to the system: Line 14 - 20 handle lines with branches, iterating through each branch and finding the connected systems. This has a time complexity of $O(k)$ in the worst case, where k is the maximum number of branches for any *line* in *lines*. Line 21 - 26 handle lines without branches, finding the connected system. This has a time complexity of $O(1)$ for each line. Since we need to consider all lines of the system, the time complexity for processing lines is $O(m * k)$. Combining these steps, the total time complexity for processing each system and its connected lines is $O(m * k)$ in the worst case. For this reason, iterating over all n systems results in an overall time complexity of $O(n * m * k)$ of Algorithm 1. ■

3) *Call graph generation:* Once the callee map is prepared, Simulysis proceeds to generate the call graph, which will be utilized later in the CIA process.

The generation of this call graph for a given Simulink project is detailed in Algorithm 2.

The algorithm takes as input the list of systems from Step 1 (*systemList*) and the callee map from Step 2 (*systemCalleeMap*). After processing, the algorithm outputs the desired call graph. The process begins by initializing the call graph as an empty set (\emptyset). It's important to note that Simulink already generates a unique identifier, known as SID. For this reason, Simulysis leverages this SID value as the identifier for each node in the call graph. The algorithm iterates over each system (*system*) in the system list (*systemList*), retrieving the SID of the system (line 4). It then either retrieves an existing node or creates a new one in the call graph (*callGraph*) that corresponds to the system (*system*) (line 5). Next, the algorithm fetches the callee list (*calleeList*) of the system (*system*) from the callee map (*systemCalleeMap*) (line 6). It then cycles through the callee list (*calleeList*), identifies the node corresponding to the callee system node in the call graph, and establishes an edge between the callee and caller (lines 7 - 11). This entire process is repeated for each system (lines 3 - 12). Finally, the algorithm returns the call graph (line 13).

Time Complexity Analysis

Algorithm 2 Generate the Corresponding Call Graph for a Simulink Project

Input: *systemList*: List of systems in the project
systemCalleeMap: A map of callees by system

Output: *callGraph*: The corresponding call graph of the project

```
1 begin
2   callGraph ← ∅
3   foreach system in systemList do
4     systemSID ← getSystemSID(system)
5     node ← getOrCreateNode(callGraph, systemSID)
6     calleeList ← getCallees(systemCalleeMap,
7                           system)
8     foreach callee in calleeList do
9       calleeSystemSID ← getSystemSID(callee)
10      calleeNode ← getOrCreateNode(callGraph,
11                                calleeSystemSID)
12      createEdge(callGraph, node, calleeNode)
13    end
14  end
15 return callGraph
16 end
```

Theorem III.2. Let n be the total number of systems in *systemList*, and m be the maximum number of callee systems for any system in *systemList*. The time complexity of Algorithm 2 is $O(m * n)$.

Proof: From Algorithm 2, we see that the *foreach* loop at lines 3 - 12 iterates n times, resulting in time complexity of this loop is $O(n)$. Given a specific system (*system*) in the loop, other actions in this loop at lines 4 - 6 are getting the system id (*systemSID*), its corresponding node (*node*), and its list of callee systems (*calleeList*). These are simple actions that retrieve data from a data store such as a database or file, resulting in a time complexity of $O(1)$. Let m be the maximum number of callee systems in all *calleeList*, the second loop (line 7 - 11) will have time complexity of $O(m)$. As a result, the overall time complexity of this algorithm is $O(m * n)$. ■

4) *Signal tracing method:* In large Simulink projects, when facing a bug, especially a signal-related bug, system engineers must be able to fix the bug by checking the signal to find the bug. We call this the signal tracing problem. This is a complex task as the signal can go from the input through multiple levels of the system model to the output of the model. Using the corresponding call graph, the signal tracing problem can be modeled as the path-finding problem inside a directed graph.

It is important to identify the relationships between objects in Simulink models to implement the Depth-First Search algorithm (DFS). To reduce the complexity of the algorithm, Simulysis proposes identifying relationships between objects, including systems, lines, and branches. There are six types of relationships: System-Line, Line-Branch, Branch-Branch, Branch-System, From-Goto, and Parent-Child.

System-Line Relationship The System-Line relationship is the most common in the Simulink model, defined through the object's ID attribute. Fig. 8 illustrates an example of information about systems and one line in an ".slx" file. It can be seen that objects are connected through ID values;

each block has a unique ID value that does not duplicate any other block within the same file. The line will have two child tags <P> with the attributes "Name=Dst" and "Name=Src". The value of the "Name=Dst" tag will match the ID value of a system in the file, indicating that it is the destination system. The value of the "Name=Src" tag will indicate the source system.

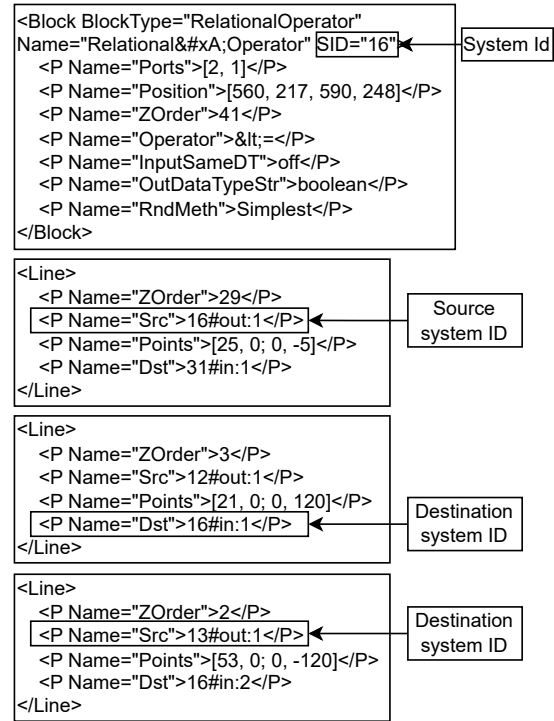


Fig. 8. System-line relationship.

System-Branch Relationship

Fig. 9 illustrates an example of the System-Branch relationship. Similar to the System-Line relationship, the System-Branch relationship is determined by the ID value. The value of the child tag "Name=Dst" will also indicate the destination system connected by the branch. However, unlike a line, a branch will not have a child tag <P> with the attribute "Name=Src".

Line-Branch Relationship

A branch is defined as a child object of a line in Simulink in the model. A connection line can have multiple branch lines. Fig. 10 describes the information about a line in the ".slx" file. The <Line> tag contains two child tags <Branch> that represent the line with two corresponding branch lines.

Branch-Branch Relationship

Similar to the Line-Branch relationship, the Branch-Branch relationship is formed when a branch contains child tags that are other branches, as shown in Fig. 11.

Goto-From Relationship

Goto and From are two special types of blocks, where a Goto block can transmit a signal to a corresponding From block and vice versa. The notable point is that these two

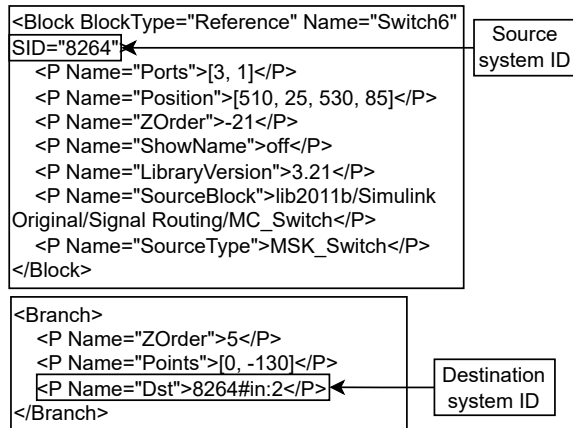


Fig. 9. System-Branch Relationship.

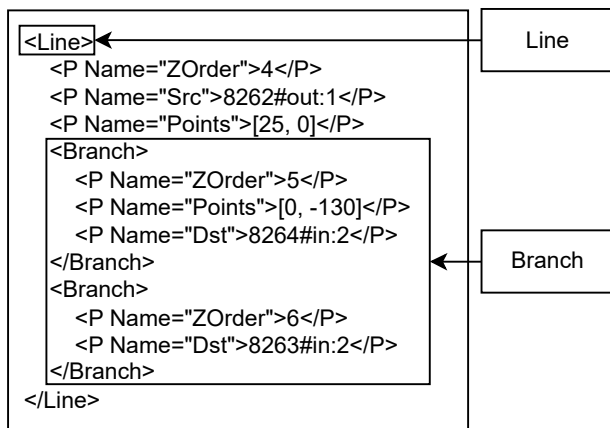


Fig. 10. Line-branch relationship.

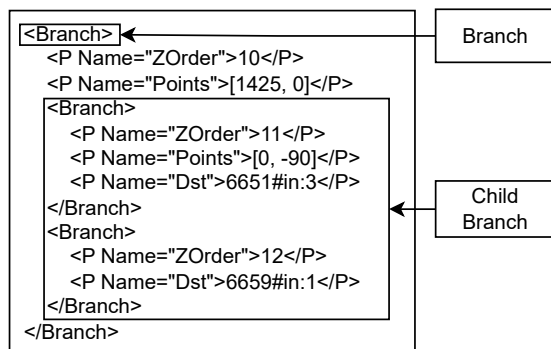


Fig. 11. Branch-branch relationship.

blocks do not connect with each other through a line. A Goto block can connect with multiple From blocks located at different positions in the model. The Goto-From relationship is determined through the GotoTag parameter. Two blocks with the same GotoTag value are considered to be connected. As shown in Fig. 12, the GotoTag value of both blocks is “xclcn”, so they are connected.

Parent-Child Relationship

A system is called a parent if it contains a subsystem; the

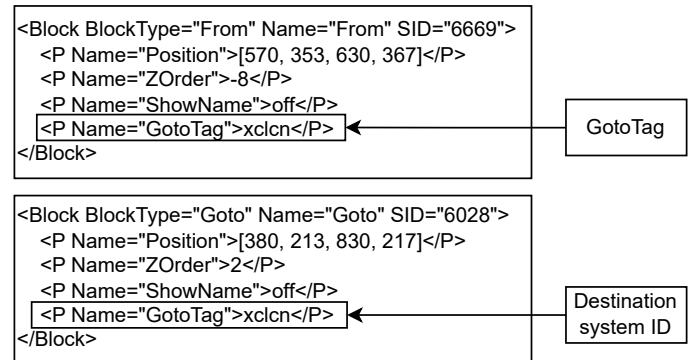


Fig. 12. Goto-from relationship.

objects within the subsystem will be children of that block. In the structure of a MATLAB Simulink model, Simulysis identifies three types of systems that contain subsystems: “Subsystem”, “ModelReference”, and “Reference”. The “ModelReference” and “Reference” blocks have similar structures; the files containing the subsystems of these two types of blocks will be in different files within the project. The “Subsystem” block has a simpler structure, with its subsystem contents residing in the same file as the current model. Fig. 13 describes an example of a line being the input to a subsystem.

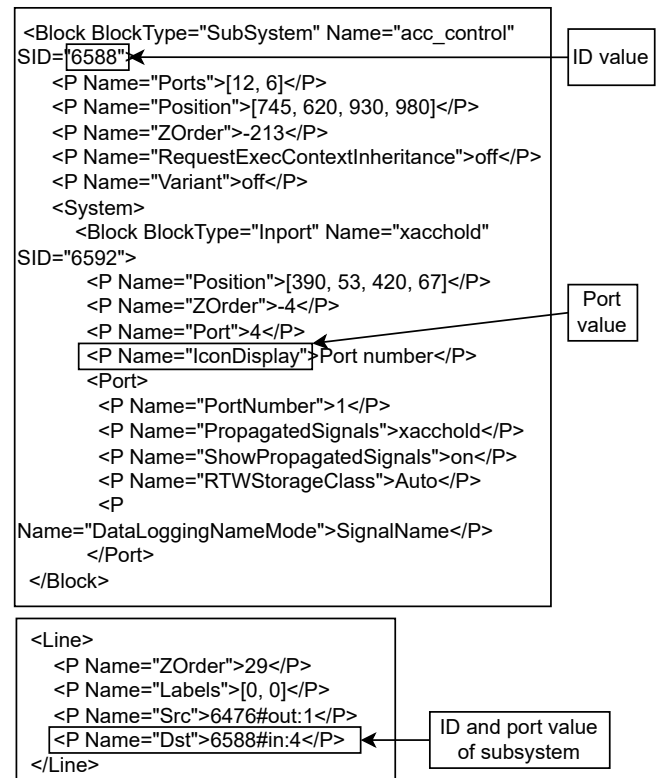


Fig. 13. Parent-child relationship

In this figure, the subsystem will be connected to the parent system through the Port value. A line/branch line will be an input to a subsystem if it contains a child tag <P> with the attribute “Name=Dst” having an ID value matching the parent system’s ID value and the destination Port value matching an

input Port value of the parent system. Similarly, a line will be an output of the subsystem if it contains a child tag <P> with the attribute “Name=Src” having an ID value matching the parent system’s ID value and the source Port value matching an output Port value of the parent system.

Signal Tracing Algorithm

Simulysis uses a DFS algorithm to traverse the call graph with the support of a stack of nodes. We maintain the three global parameters to support the DFS traversing algorithm. First, *pathList* is the list of all paths from the *startNode* to *endNode*. Second, *path* is a temporary variable for storing the path being processed. Third, *stack* is a stack data structure to store nodes of the call graph being processed. The algorithm accepts *startNode* and *endNode*, which are the start and end nodes of the signal being checked. When the algorithm finishes running, *pathList* stores the list of all paths from *startNode* to *endNode*. Details of the algorithm are shown in Algorithm 3.

Algorithm 3 Signal Tracing

Input: *startNode* \in *callGraph*: the start node of the *path* being checked.

endNode \in *callGraph*: the end node of *path* being checked.

Output: The list of paths is stored in *pathList*

```
1 begin
2   pathList, path, stack  $\leftarrow$   $\emptyset$ 
3   stack.push(startNode)
4   while stack is not empty do
5     curNode  $\leftarrow$  stack.pop()
6     path  $\leftarrow$  path  $\cup$  {curNode}
7     if curNode is endNode then
8       newPath  $\leftarrow$  clone_path(path);
9       pathList  $\leftarrow$  pathList  $\cup$  {newPath};
10      if stack is not empty then
11        curNode  $\leftarrow$  stack.pop()
12        path  $\leftarrow$  remove_visited_nodes(path, curNode)
13        call Algorithm 3(curNode, endNode);
14      end
15    else
16      childList  $\leftarrow$  get_child_nodes(curNode)
17      foreach child in childList do
18        stack.push(child)
19      end
20      if stack is not empty then
21        curNode  $\leftarrow$  stack.pop()
22        call Algorithm 3(curNode, endNode);
23      end
24    end
25  end
26 end
```

The algorithm starts by initializing the path list (*pathList*), temporary path (*path*), and the stack data structure (*stack*) as empty sets (\emptyset) (line 2). Then, the algorithm push *startNode* to the *stack* (line 3). While the *stack* is not empty, the algorithm does the following smaller tasks. First, the algorithm retrieves the top node from the stack (*stack*) and adds it to the temporary path (*path*) (lines 5 - 6). In case the current node (*curNode*) is the end node (*endNode*), we have found one path from the start node (*startNode*) to the end node (*endNode*). The algorithm adds the cloned path (*newPath*) of *path* to the path list (*pathList*) (lines 8 - 9). We clone the path to ensure that later changes does not affect the cloned

path. Next, the algorithm finds the next path from the start node (*startNode*) to the end node (*endNode*). If the stack (*stack*) is not empty, the algorithm retrieves the current node (*curNode*) from the top node of the stack (line 11). After getting the previous path, currently, *path* still contains nodes from the previous path. We remove all unnecessary nodes from *path* (line 12). These nodes are all nodes from the end of *path* to the sibling nodes of *curNode* that previously belonged to *path*. Then, the algorithm recursively calls itself to find paths from the current node (*curNode*) to the end node (*endNode*) (line 13).

In case the current node (*curNode*) is not the end node (*endNode*), the algorithm retrieves all next nodes (we call them child nodes) (*childList*) in the directed call graph (line 16) and pushes them to the stack (*stack*) (lines 17 - 19). Then, if the stack (*stack*) is not empty, the algorithm retrieves the top node (*curNode*) and recursively calls itself to find path from the current node (*curNode*) to the end node (*endNode*) (lines 20 - 23).

The time complexity of Algorithm 3 is $O(V, E)$, where V and E is the sets of nodes and edges in the call graph [41].

C. Change Set Computation

Simulysis generates call graphs corresponding to two versions of a Simulink project, as outlined in Algorithm 2. For impact analysis using the WAVE-CIA method, it’s necessary for Simulysis to identify the change set between these two call graph versions. The process relies on the use of the “Name” and “SID” properties of a Simulink block to determine its presence across both graph versions. If a block is found in both versions, Simulysis examines if any of its properties have changed. If a change is detected, the block is marked as changed. The specifics of this process are detailed in Algorithm 4.

Algorithm 4 Compute the Change Set

Input: *callGraphBefore*, *callGraphAfter*

Output: *changeSet*

```
1 begin
2   changeSet, changedList, addedList, deletedList  $\leftarrow$   $\emptyset$ 
3   foreach node in callGraphAfter do
4     if  $\neg$  NodeExistsIn(node, callGraphBefore) then
5       AddToList(node, addedList)
6     end
7     if NodeExistsIn(node, callGraphBefore) and
       PropertyChanged(node, callGraphBefore,
8         callGraphAfter) then
9       AddToList(node, changedList)
10    end
11  end
12  foreach node in callGraphBefore do
13    if  $\neg$  NodeExistsIn(node, callGraphAfter) then
14      AddToList(node, deletedList)
15    end
16  end
17  changeSet  $\leftarrow$  toSet(addedList, deletedList,
18    changedList)
19  return changeSet
20 end
```

Algorithm 4 takes in two call graphs (*callGraphBefore*, *callGraphAfter*) corresponding to two versions of a given Simulink project. The output is the required change set (*changeSet*). The algorithm begins by initializing the change set (*changeSet*) and three temporary lists: *changedList*, *addedList*, and *deletedList* as empty set (\emptyset). For each *node* in *callGraphAfter*, the algorithm checks if the node also exists in *callGraphBefore*. If it does not, the node is considered as added and is added to the added node list (*addedList*) (lines 4 - 6). If the node exists in *callGraphBefore* and any of its properties have changed in the two graphs (detected by the changes in the text of properties), the node is added to the change list (*changedList*) (lines 7 - 9). The algorithm then checks for any node that exists in *callGraphBefore* but not in *callGraphAfter*. Such a node is considered as deleted and is added to the deleted node list (*deletedList*) (lines 11 - 15). Finally, the algorithm adds the three lists: *addedList*, *deletedList*, and *changedList* to the *changeSet* (line 16) and returns the result.

Time Complexity Analysis

Theorem III.3. Let n be the maximum number of nodes in the call graphs for both the previous and updated project versions. The time complexity of Algorithm 4 is $O(n)$.

Proof: Let m be the number of nodes in the call graph corresponding to the new project version (*callGraphAfter*). From Algorithm 4, it is evident that the *foreach* loop at lines 3 - 10 has a time complexity of $O(m)$. In this loop, thanks to the utilization of the hash table data structure, the action that verifies if a node exists in a call graph has a time complexity of $O(1)$ (line 4 and line 7). The functions *PropertyChanged* and *AddToList* are straightforward functions with a time complexity of $O(1)$, resulting in the time complexity of this loop being $O(m)$. Let k be the number of nodes in the call graph corresponding to the old project version (*callGraphBefore*). Following the same reasoning as above, the loop at lines 11 - 15 would have a time complexity of $O(k)$. The *toSet* function at line 16 is a simple function with a complexity of $O(1)$. From the above analysis, let $n = \max(m, k)$, the overall time complexity of the algorithm is $O(\max(m, k)) \equiv O(n)$. ■

D. Impact Set Computation

Upon obtaining the change set from Phase 2, Simulysis proceeds to compute the impact set. This set includes Simulink files and systems that are influenced by the Simulink components present in the change set. Given a call graph and a change set, which is a subset of the call graph, Simulysis follows a two-step process to calculate the impact set. The first step involves the computation of the core set, which includes the computation of the neighbor set as detailed in Section III-D1. The second step is the computation of the impact set itself.

1) *Neighbor Set Computation:* In WAVE-CIA, a core set comprises Simulink systems that are potentially influenced by several systems within the change set. The computation of the core set utilizes the concept of a neighbor set associated with a given component. A neighbor set of a specific component, with a depth of n , includes other components in the call graph that can be reached via a maximum of n edges.

The parameter “depth” plays a crucial role in the zoning of the Coreset. The depth parameter significantly impacts the accuracy and efficiency of the Coreset. Depth determines the level at which data points are considered and clustered within the Coreset. A higher depth value implies a finer granularity, resulting in more detailed and potentially more accurate coresets, whereas a lower depth value leads to a more generalized and coarser representation of the data. To identify the optimal depth for a specific problem, empirical experimentation is essential. The appropriate depth varies depending on the nature of the dataset and the specific requirements of the analysis. By conducting experiments with varying depth values, we can evaluate the performance and accuracy of the resulting Coreset, thereby determining the most suitable depth parameter for their particular application.

Simulysis employs a recursive algorithm to identify the neighbors of a specific system node within the call graph. The procedure for determining the neighbor set is detailed in Algorithm 5.

Algorithm 5 Compute the Neighbor Set

Input: *systemNode*: a given system in a call graph
depth: The maximum edges between two Simulink to consider neighbors
callGraph: The call graph being analyzed
Output: *neighborSet*: The required neighbor set

```
1 begin
2   systemNode.isVisited  $\leftarrow$  true
3   neighborSet  $\leftarrow$   $\emptyset$ 
4   callers  $\leftarrow$  findCallers(systemNode, callGraph)
5   callees  $\leftarrow$  findCallees(systemNode, callGraph)
6   neighborSet  $\leftarrow$  neighborSet  $\cup$  callers  $\cup$  callees
7   if depth  $\leq$  1 then
8     return neighborSet
9   end
10  foreach neighborNode  $\in$  neighborSet and
    neighborNode.isVisited = false do
11    subNeighborSet  $\leftarrow$  call Algorithm 5(neighborNode,
    depth - 1, callGraph)
12    neighborSet  $\leftarrow$  neighborSet  $\cup$  subNeighborSet
13  end
14 end
```

The algorithm initiates by marking the *systemNode* as visited (line 2) and setting the *neighborSet* as an empty set (\emptyset) (line 3). It then identifies the direct caller (*callers*) and direct callee (*callees*) lists of the current node, and appends these lists to the neighbor set (lines 4 - 5). This step locates all neighbor nodes that are connected to the current *systemNode* via a single edge. The recursion terminates when *depth* \leq 1 (lines 7 - 9). Subsequently, for each unvisited *neighborNode* in the *neighborSet*, the algorithm recursively calls itself to find the neighbor set (*subNeighborSet*) of the *neighborNode* and adds it to the *neighborSet* (lines 10 - 13). In this step, the algorithm locates all direct neighbors of the *neighborNode*, hence the parameter *depth* must be *depth* - 1. Upon termination of the algorithm, we obtain the neighbor set of the given *systemNode*, with a *depth* number equivalent to the number of edges from the *systemNode*.

2) *Time Complexity Analysis:*

Theorem III.4. Let n be the total number of nodes in the call graph, d be the maximum degree of any node in the call graph, and k be the specified depth. The time complexity of Algorithm 5 is $O(n * d^k)$.

Proof: From Algorithm 5, we see that lines 4 - 5 finds direct callers and callees of *systemNode* and adds them to *neighborSet* which involves checking connections and has a time complexity of $O(d)$ since d is the maximum degree of any node. Lines 7 - 9 checks if *depth* is less than or equal to 1 and, if so, returns *neighborSet*, which has a time complexity of $O(1)$. Line 10 - 13 handle the recursive exploration of neighbors. For each unvisited neighbor in *neighborSet*, the algorithm recursively calls itself with *depth* - 1. This recursion explores up to k levels, and at each level, it may process up to d neighbors. The time complexity for exploring neighbors at each level is $O(d)$, and since the recursion depth is $O(d^k)$. Given that there are n nodes in the call graph and in the worst case, each node needs to process its neighbors, the overall time complexity of the algorithm is $O(n * d^k)$. As a result, the overall time complexity of this algorithm is $O(n * d^k)$. ■

3) *Core Set Computation:* The computation of the core set is a crucial step in the WAVE-CIA method. This computation depends on the change set, and its outcome is subsequently utilized to calculate the impact set. The specifics of the core set computation process can be found in Algorithm 6.

Algorithm 6 Compute Core Set

Input: *callGraph*: The call graph of the Simulink project
changeSet: A set of components that have been changed
depth: An integer specifying the neighbor set exploration depth

Output: *coreSet*: the required core set

```
1 begin
2   coreSet ← ∅
3   foreach node ∈ callGraph do
4     if node ∉ changeSet then
5       neighborSet ← call Algorithm 5(node, depth,
6         callGraph)
7       if —neighborSet ∩ changeSet— > 1 then
8         coreSet ← coreSet ∪ {node}
9       end
10    end
11  coreSet ← coreSet ∪ changeSet
12  return coreSet
13 end
```

The algorithm takes in three inputs: the call graph (*callGraph*), the change set (*changeSet*), and an integer (*depth*) that determines the extent of the neighbor set computation. The output is the desired core set (*coreSet*). The algorithm starts by initializing the core set as an empty set (\emptyset) (line 2). It then iterates over each node (*node*) in the call graph (*callGraph*) that is not in the change set (*changeSet*), and invokes Algorithm 5 to compute the neighbor set (*neighborSet*) of the node (line 5). If the *neighborSet* and *changeSet* share more than one node, the node is added to the *coreSet* (lines 6 - 8). After all nodes in the *callGraph* have been checked, the algorithm incorporates the *changeSet* into the *coreSet*, as the

core set is a superset of the change set (line 11). The algorithm concludes by returning the *coreSet* (line 12).

Time Complexity Analysis

Theorem III.5. Let n be the total number of nodes in the call graph, d be the maximum degree of any node in the call graph, and k be the specified depth. The time complexity of Algorithm 5 is $O(n * d^k)$.

Proof: From Algorithm 6, we see that line 2 initializes the core set with a time complexity of $O(1)$. Lines 3 - 10 iterate through each node in the call graph that is not in the change set. For each node, line 5 calls Algorithm 5 to compute the neighbor set, which has a time complexity of $O(d^k)$ since Algorithm 5 explores up to k levels of neighbors. Line 6 checks the intersection of the neighbor set and the change set, which involves set operations with a time complexity of $O(d)$. Line 7 adds the node to the core set if the intersection condition is met, with a time complexity of $O(1)$. Line 11 merges the change set into the core set, which has a time complexity of $O(n)$ in the worst case. Given that there are n nodes in the call graph and each node's neighbor set computation involves $O(d^k)$ operations, the overall time complexity of the algorithm is $O(n * d^k)$. ■

4) *Impact set calculation:* Although the core set is expected to encompass many components affected by the change sets, other impacted components may not be included in the core set. Consequently, it becomes necessary to broaden the change ripple to account for additional impacts that might have been overlooked, or in simpler terms, to enlarge the core. The fundamental concept of this procedure is that any node not already in the impact set will be added if both its caller and callee sets share nodes with the existing impact set. The specifics of this procedure can be found in Algorithm 7.

Algorithm 7 Compute the Impact Set

Input: *callGraph*: The call graph of the Simulink project
coreSet: A core set

Output: *impactSet*: the required impact set

```
1 begin
2   impactSet ← coreSet
3   isStable ← false
4   while not isStable do
5     isStable ← true
6     foreach node ∈ callGraph do
7       if node ∉ impactSet then
8         callerSet ← findCallers(node, callGraph)
9         calleeSet ← findCallees(node, callGraph)
10        if —callerSet ∩ impactSet— > 0 and
11          —calleeSet ∩ impactSet— > 0 then
12          impactSet ← impactSet ∪ {node}
13          isStable ← false
14        end
15      end
16    end
17  end
18  return impactSet
end
```

Algorithm 7 takes in two inputs: the call graph (*callGraph*) and the core set (*coreSet*). It outputs the desired impact set

(*impactSet*). The algorithm begins by initializing the impact set (*impactSet*) with the core set (*coreSet*) (line 2). It then continuously checks each node in the call graph that is not in the impact set to determine if it has at least one caller and one callee within the impact set. If it does, the node is considered to be impacted by the change set (lines 6 - 15). This process is repeated until the impact set reaches a state of stability, where no additional nodes are added during the checking process. The stability of the impact set is monitored using a boolean flag, *isStable* (line 3), which is initially set to *true* (line 5) at the start of the call graph checking loop (lines 6 - 15). This flag is reset to *false* when a new node is added to the impact set (line 12). If the flag remains *false* after the loop, the loop is rerun. If not, the algorithm terminates the loop and returns the impact set (*impactSet*) (line 17).

Time Complexity Analysis

Theorem III.6. Let n be the total number of nodes in the call graph, and d be the maximum degree of any node in the call graph. The time complexity of Algorithm 6 is $\mathcal{O}(n \times d)$.

Proof: From Algorithm 7, we see that line 2 initializes the impact set with a time complexity of $\mathcal{O}(1)$. The outer loop (lines 4 - 16) repeats until no changes occur, and within each iteration, the inner loop (lines 6 - 15) iterates through each node in the call graph. Line 10 checks the intersections of the caller set and callee set with the impact set, each of which involves set operations with a time complexity of $\mathcal{O}(d)$. Lines 11 - 12 add the node to the impact set if the conditions are met and set the changed flag, both of which have a time complexity of $\mathcal{O}(1)$. Assuming the algorithm stabilizes after a constant number of iterations, the time complexity for each iteration through the call graph is $\mathcal{O}(n \times d)$. Therefore, the overall time complexity of the algorithm is $\mathcal{O}(n \times d)$. ■

IV. IMPLEMENTATION

We've developed a tool named Simulysis², utilizing the MVC framework and C#.NET within the Microsoft Visual Studio IDE. The architecture of Simulysis, depicted in Fig. 14, comprises three primary components.

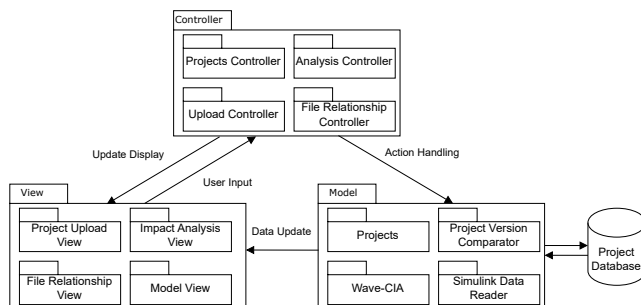


Fig. 14. The Simulysis architecture.

- **Model:** The Model component, encompassing four sub-components, is tasked with managing project data. The Projects sub-component oversees the information of Simulink projects. The Simulink Data

Reader extracts and handles crucial data from Simulink projects. The Project Version Comparator sub-component performs comparisons between different versions of Simulink projects. Lastly, the WAVE-CIA sub-component carries out the change impact analysis.

- **Controller:** The Controller component, composed of four sub-components, acts as a bridge, overseeing user interactions and the flow of the application. The four sub-components include the Upload Controller, Project Controller, Analysis Controller, and File Relationships. These are tasked with handling requests on project uploads, project information management, project file and structure analysis, and file relationship management, respectively.
- **View:** The View component displays necessary information via interface packages such as Project Upload View, File Relationship View, Impact Analysis View, and Model View. These views facilitate the user interface for Simulink project uploads, file relationship management, impact analysis execution, and the visualization of changes and impacts within a specific project, respectively.

Fig. 15 and 16 display Simulysis screens, showcasing a standard Simulink file. The screen is divided into three sections, each marked with numbers for straightforward reference. The three sections are described as follows.

- **Section 1:** This section displays the differences between two specified file versions within the Simulink project. It features a search bar and tabs labeled “Added”, “Changed”, “Deleted”, and “Impacted”. These tabs categorize the systems, which are listed with an option to adjust their visibility.
- **Section 2:** This section outlines the functionalities associated with the WAVE-CIA method. It comprises two components: a dropdown list for selecting the *depth* of the Core set computation, and a button to initiate the WAVE-CIA analysis.
- **Section 3:** This section displays the Model View of the chosen Simulink file. It illustrates the Simulink systems (blocks) using the data of blocks and lines extracted from Simulink files, as detailed in Section III-B1. In this view, systems that have been added are highlighted in green; changed systems are marked in orange; and impacted systems are indicated in cyan. Other systems are either not filled or their color is determined by their properties.

V. EXPERIMENTS

To assess the practical utility and efficacy of Simulysis, we have conducted tests using various open-source projects. These projects, sourced from GitHub³, have a Git history of committed changes to Simulink source files. The projects include the following:

²<https://github.com/KaoSon2004/Simulysis-CIA>

³<https://github.com/>

Fig. 15. An example of Simulink file in Simulysis.

Fig. 16. An example of impacted systems in Simulysis.

TABLE VI. EXPERIMENTAL RESULTS

Project	CB	IB	TB	%	Mem (MB)	Time (s)
Intelligent-EMS	88	126	397	31.73	1.31	0.123
Failure Detection	16	26	1059	2.46	4.33	0.35
FMT-Model	2085	2668	9700	27.51	16.64	1.911
Reinforcement Learning	118	156	396	39.39	0.65	0.232
Kulge	263	322	1240	25.97	2.72	0.288
Propulsion	189	269	1115	24.13	5.78	0.767

Based on the experimental results presented in Table VI, we can make the following observations:

- For the “Intelligent-EMS” project, out of 397 blocks, 23 are impacted, which constitutes 5.79% of the total.
- For the “Failure Detection” project, out of 1059 blocks, 26 are impacted, which represents 2.46% of the total.
- For the “FMT-Model project”, out of 9700 blocks, 289 are impacted, which represents 2.98% of the total.
- For the “Reinforcement Learning” project, out of 396 blocks, 50 are impacted, which represents 12.63% of the total.
- For the “Kugle”, out of 1240 blocks, 76 are impacted, which represents 6.13% of the total.
- For the “Propulsion”, out of 1115 blocks, 75 are impacted, which represents 6.73% of the total.
- In comparing the change calculation results, we found them to be accurate when cross-checked with MATLAB’s Simulink model comparison tool. For the impact analysis, we tested our implementation and confirmed it aligns correctly with the WAVE-CIA algorithm. However, the correctness of the WAVE-CIA algorithm itself is beyond the scope of this discussion.
- For the largest project (FMT-Model), which comprises a total of 9700 systems, the memory and time consumption is 24.25 MB and 14.79 seconds, respectively. Given the current capabilities of memory and processing, we consider this to be acceptable. This demonstrates the practical applicability of Simulysis for real-world projects.
- While it appears that the memory and time usage may be proportional to the number of changed and impacted systems, there is no definitive evidence to support this observation.

VI. RELATED WORKS

Numerous studies have been conducted on the Change Impact Analysis (CIA) issue, specifically about Simulink projects and more generally, software projects. In this section, we focus solely on the most relevant studies and tools [37], [5], [6], [40], [43], [44], [3], [45].

In the realm of model-based testing, many researchers have made significant contributions to UML impact analysis. In

2003, Briand et al. introduced a UML model-based approach to impact analysis that can be implemented prior to any changes [43]. Initially, they ensure the consistency of the UML diagrams. Subsequently, they identify the modifications between two distinct versions of a UML model. Following this, they determine the model elements that are directly or indirectly affected by these changes using impact analysis rules (formulated in Object Constraint Language). Our interest aligns with Briand’s in the area of software change impact analysis. However, our approach involves analyzing the source code of Simulink projects to construct the call graph and applying the WAVE-CIA method to compute the impact set.

In 2010, Fournier et al. introduced an automated model-based impact analysis [44]. Fournier’s approach involves a set of algorithms that identify dependent and impacted elements following the detection of changes in a UML/OCL Statechart diagram. This allows an engineer to pinpoint the critical aspects of the software and determine which functionalities require regression testing. While our interests align with Fournier’s in the area of software system regression testing, our focus is specifically on the change impact analysis of Simulink projects.

In 2010, Li et al. introduced the WAVE-CIA method, a novel approach for calculating the impact set derived from the call graph of various types of software projects [3]. Our work aligns with this interest in change impact analysis. However, our application of the WAVE-CIA method is specifically tailored to Simulink projects. We construct the call graph using Simulink files and subsequently apply the WAVE-CIA method to these generated call graphs.

Diffplug [37] conducts impact analysis of Simulink projects using a straightforward method known as signal tracing. This method allows users to select a block and trace its signal, enabling them to determine the potential impact of a block change. However, Diffplug necessitates that users manually navigate through each level of the model to identify the impacted target inputs and/or outputs. Conversely, Simulysis automates this process by analyzing the project, calculating changes, and computing the impact set using the WAVE-CIA method.

In 2017, Rapos and Cordy introduced SimPact, a method and tool for analyzing the impact of changes in Simulink models [5]. SimPact’s impact analysis of a Simulink model involves two stages: change isolation and the determination of the impact of changes on test values. Initially, SimPact identifies the differences between two versions of a model. Subsequently, it assesses the potential impact of these changes

on inputs and outputs by traversing the entire model hierarchy, tracing, and highlighting all potential impacts from a specific block at all levels. While our interest in the impact analysis of Simulink projects aligns with that of SimPact, there are key differences between SimPact and Simulysis. In the change isolation phase, SimPact utilizes the model differencing script of Rapos and Cordy [46], which employs the internal Simulink comparison operator. Conversely, Simulysis directly analyzes the Simulink project to construct the call graph and compares the two call graph versions to identify the change set. In the impact analysis phase, SimPact depends on the caller-callee relationship to identify the impact set, which is fundamentally different from Simulysis, which uses the WAVE-CIA method to determine the impact set.

In 2020, Bennett et al. introduced the Boundary Diagram Tool (BDT), a method designed for change impact analysis of large Simulink projects within embedded systems [6]. BDT conducts impact analysis by tracing changes throughout the Simulink model. Our work aligns with this interest in Simulink project impact analysis. However, our approach differs as we directly analyze Simulink projects to construct the call graph and utilize the WAVE-CIA method to compute the impact set.

In 2021, Jaskolka et al. [40] conducted an examination of real industrial software repositories and their version control systems to gain insights into potential changes in Simulink. Their intention was to guide the usage of Simulink in industrial practices and understand the distribution of change types in Simulink projects. However, Simulysis differs fundamentally in its objectives, as it primarily aims to compute the impact set.

In 2023, Tran et al. developed CIA4CS, a method specifically designed for Change Impact Analysis (CIA) of C# projects, based on the WAVE-CIA approach [45]. CIA4CS fills a significant void in existing methodologies, considering the widespread use of the C# language in the software industry. The method utilizes static code analysis to build dependency graphs, which then facilitates the application of the WAVE-CIA. The authors have integrated CIA4CS into a tool and conducted tests on a variety of C# projects. Their experiments demonstrated the tool's efficiency in analyzing project components, dependencies, and the elements impacted across different versions. While our interests align with Tran's in terms of the CIA and the WAVE-CIA method, our focus is specifically on the CIA of Simulink projects.

VII. CONCLUSION

We have introduced Simulysis, a method designed for the change impact analysis of Simulink projects. When provided with two versions of a Simulink project, Simulysis performs the CIA by constructing the call graph from the files, identifying the changes, and calculating the impact set using the WAVE-CIA method. We have incorporated this method into a tool, also named Simulysis, and conducted several experiments on open-source projects, yielding promising results. Further discussion about the method is provided in the paper.

Looking ahead, we envision several enhancements to Simulysis to augment its capabilities. Firstly, we are actively working on improving the speed of call graph construction and

refining the algorithm used for impact set determination. Secondly, we plan to upgrade the tool's user interface to facilitate better visualization of large-scale Simulink projects. Lastly, we aim to develop additional features such as report exporting and system integration support to better accommodate user environments. Through these advancements, our goal is to offer users an efficient, user-friendly tool that bolsters the role of Simulysis in ensuring effective software quality assurance for Simulink projects.

ACKNOWLEDGMENTS

This research was funded by the research project QG.25.09 of Vietnam National University, Hanoi.

REFERENCES

- [1] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1475>
- [2] M. Kretsou, E.-M. Arvanitou, A. Ampatzoglou, I. Deligiannis, and V. C. Gerogiannis, "Change impact analysis: A systematic mapping study," *Journal of Systems and Software*, vol. 174, p. 110892, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S016412122030282X>
- [3] B. Li, Q. Zhang, X. Sun, and H. Leung, "Wave-cia: A novel cia approach based on call graph mining," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1000–1005. [Online]. Available: <https://doi.org/10.1145/2480362.2480554>
- [4] K. A. Alam, R. Ahmad, A. Akhunzada, M. H. N. M. Nasir, and S. U. Khan, "Impact analysis and change propagation in service-oriented enterprises: A systematic review," *Information Systems*, vol. 54, pp. 43–73, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0306437915001179>
- [5] E. J. Rapos and J. R. Cordy, "SimPact: Impact analysis for simulink models," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 489–493.
- [6] B. Mackenzie, V. Pantelic, G. Marks, S. Wynn-Williams, G. Selim, M. Lawford, A. Wassyng, M. Diab, and F. Weslati, "Change impact analysis in simulink designs of embedded systems," ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1274–1284. [Online]. Available: <https://doi.org/10.1145/3368089.3417060>
- [7] Bohner, "Impact analysis in the software change process: a year 2000 perspective," in *1996 Proceedings of International Conference on Software Maintenance*, 1996, pp. 42–51.
- [8] O. Bouchaala, M. Yangu, S. Tata, and M. Jmaiel, "Dat: Dependency analysis tool for service based business processes," in *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, 2014, pp. 621–628.
- [9] S. Lehnert, "A taxonomy for software change impact analysis," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, ser. IWPSE-EVOL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 41–50. [Online]. Available: <https://doi.org/10.1145/2024445.2024454>
- [10] S. Basu, F. Casati, and F. Daniel, "Toward web service dependency discovery for soa management," in *2008 IEEE International Conference on Services Computing*, vol. 2, 2008, pp. 422–429.
- [11] D. Romano and M. Pinzer, "Using vector clocks to monitor dependencies among services at runtime," in *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, ser. QASBA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–4. [Online]. Available: <https://doi.org/10.1145/2031746.2031748>

- [12] S. Qi, B. Li, C. Liu, X. Wu, and R. Song, "A trust impact analysis model for composite service evolution," in *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01*, ser. APSEC '12. USA: IEEE Computer Society, 2012, p. 73–78. [Online]. Available: <https://doi.org/10.1109/APSEC.2012.30>
- [13] D. Zhao, S. Liu, L. Wu, R. Wang, and X. Meng, "Hypergraph-based service dependency resolving and its applications," in *2012 IEEE Ninth International Conference on Services Computing*, 2012, pp. 106–113.
- [14] Y. Wang, J. Yang, W. Zhao, and J. Su, "Change impact analysis in service-based business processes," *Service Oriented Computing and Applications*, vol. 6, no. 2, p. 131–149, 2012.
- [15] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and managing web services: issues, solutions, and directions," *The VLDB Journal*, vol. 17, no. 3, p. 537–572, 05 2008. [Online]. Available: <https://doi.org/10.1007/s00778-006-0020-3>
- [16] P. Kumar and Ratneshwer, "A review on dependency analysis of soa based system," 2014. [Online]. Available: <https://api.semanticscholar.org/CorpusID:11112313>
- [17] S. Black, "Computing ripple effect for software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 4, pp. 263–279, 2001. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.233>
- [18] H. Xiao, J. Guo, and Y. Zou, "Supporting change impact analysis for service oriented business applications," in *International Workshop on Systems Development in SOA Environments (SDSOA'07: ICSE Workshops 2007)*, 2007, pp. 6–6.
- [19] S. Wang and M. A. Capretz, "A dependency impact analysis model for web services evolution," in *2009 IEEE International Conference on Web Services*, 2009, pp. 359–365.
- [20] —, "Dependency and entropy based impact analysis for service-oriented system evolution," in *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, vol. 1, 2011, pp. 412–417.
- [21] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1475>
- [22] L. Westfall, "Bidirectional requirements traceability," *Software Quality Professional Magazine*, vol. 10, 2007. [Online]. Available: <https://api.semanticscholar.org/CorpusID:107717465>
- [23] C. Boldyreff, E. Burd, R. Hather, M. Munro, and E. Younger, "Greater understanding through maintainer driven traceability," in *WPC '96. 4th Workshop on Program Comprehension*, 1996, pp. 100–106.
- [24] S. Pfleeger and S. Bohner, "A framework for software maintenance metrics," in *Proceedings. Conference on Software Maintenance 1990*, 1990, pp. 320–327.
- [25] H. K. Dam and A. Ghose, "Mining version histories for change impact analysis in business process model repositories," *Computers in Industry*, vol. 67, pp. 72–85, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0166361514001833>
- [26] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 563–572.
- [27] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol.*, vol. 19, no. 2, p. 77–131, mar 2007. [Online]. Available: <https://doi.org/10.1002/smr.344>
- [28] H. Khanh Dam, "Predicting change impact in web service ecosystems," *International Journal of Web Information Systems*, vol. 10, no. 3, pp. 275–290, mar 2014. [Online]. Available: <https://doi.org/10.1108/IJWIS-03-2014-0006>
- [29] W. Fdhila, S. Rinderle-Ma, and C. Indiono, "Memetic algorithms for mining change logs in process choreographies," in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 47–62.
- [30] Broadcom, "Ca harvest software change manager," accessed date: 2024-02-15. [Online]. Available: <https://www.broadcom.com/products/software/service-management/harvest-software-change-manager>
- [31] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "Jripples: A tool for program comprehension during incremental change," in *Proceedings of the 13th International Workshop on Program Comprehension*, ser. IWPC '05. USA: IEEE Computer Society, 2005, p. 149–152. [Online]. Available: <https://doi.org/10.1109/WPC.2005.22>
- [32] CoderGears, "Jarchitect :: Java static analysis and code quality tool," accessed date: 2024-02-15. [Online]. Available: <https://www.jarchitect.com>
- [33] SonarSource, "Clean code: Writing clear, readable, understandable; reliable quality code — sonar," accessed date: 2024-02-15. [Online]. Available: <https://www.sonarsource.com/products/sonarqube/>
- [34] Checkmarx, "Application security testing company — software security testing solutions — checkmarx," accessed date: 2024-02-15. [Online]. Available: <https://checkmarx.com/>
- [35] R. Hat, "Ansible is simple it automation," accessed date: 2024-02-15. [Online]. Available: <https://www.ansible.com>
- [36] Atlassian, "Jira," accessed date: 2024-02-15. [Online]. Available: <https://www.atlassian.com/software/jira>
- [37] Diffplug, "Free simulink viewer and differ — diffplug," accessed date: 2024-02-15. [Online]. Available: <https://www.diffplug.com/features/simulink>
- [38] MathWorks, *Simulink - Simulation and Model-Based Design - MATLAB*, accessed date: 2024-02-25. [Online]. Available: <https://www.mathworks.com/products/simulink.html>
- [39] —, *Matlab*, accessed date: 2024-02-25. [Online]. Available: <https://www.mathworks.com/products/matlab.html>
- [40] M. Jaskolka, V. Pantelic, A. Wassyng, M. Lawford, and R. Paige, "Repository mining for changes in simulink models," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2021, pp. 46–57.
- [41] R. Tarjan, "Depth-first search and linear graph algorithms," in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1971, pp. 114–121.
- [42] G. Nava and D. Pucci, "Failure detection and fault tolerant control of a jet-powered flying humanoid robot," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 12 737–12 743.
- [43] L. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of uml models," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003, pp. 256–265.
- [44] E. Fournieret and F. Bouquet, "Impact analysis for uml/ocl statechart diagrams based on dependence algorithms for evolving critical software," sep 2010, accessed date: 2024-02-15. [Online]. Available: <https://publiweb.femto-st.fr/tntnet/entries/120/documents/author/data>
- [45] H. V. Tran, N. T. M. Loan, D. D. Kien, N. H. Trang, L. V. Huy, and P. N. Hung, "Cia4cs: A method for change impact analysis of c# projects," *Journal on Information Technologies and Communications*, vol. 2024, no. 01, 11 2023. [Online]. Available: <https://doi.org/10.1145/2480362.2480554>
- [46] E. J. Rapos and J. R. Cordy, "Examining the co-evolution relationship between simulink models and their test cases," in *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, ser. MiSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 34–40. [Online]. Available: <https://doi.org/10.1145/2896982.2896983>