Received 25 January 2024, accepted 9 February 2024, date of publication 12 February 2024, date of current version 20 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3365352

RESEARCH ARTICLE

BERDD: A Behaviour Engineering-Based Approach for Requirements Defects Detection

SAJID ANWER[1,2], LIAN WEN[1], MAHMOOD UL HASSAN[3], ZHE WANG[1], AMIN A. AL-AWADY[3], AND YAHYA ALI ABDELRAHMAN ALI[4]

[1]School of Information and Communication Technology, Griffith University, Brisbane, QLD 4111, Australia
[2]Department of Software Engineering, FAST-National University of Computer and Emerging Sciences, Islamabad 35400, Pakistan
[3]Department of Computer Skills, Deanship of Preparatory Year, Najran University, Najran 66241, Saudi Arabia
[4]Department of Computer Science and Information Systems, Najran University, Najran 66241, Saudi Arabia

Corresponding author: Sajid Anwer (sajid.anwer@nu.edu.pk)

This work was supported in part by the School of Information and Communication Technology, Griffith University, Australia; and in part by the Deanship of Scientific Research and Innovation, Ministry of Education, Najran University, Saudi Arabia, under the Institutional Funding Committee (IFC), under Project NU/IFC/2/SERC/-/47.

ABSTRACT Detecting software requirements defects is crucial in reducing the risk of software project failures. Existing methods for automatic detection, especially during requirements changes, are limited in coverage and often lack robust tool support. Addressing this gap, we define the four most common types of requirements defects (incompleteness, inconsistency, redundancy, and ambiguity) and present algorithms for their detection. We propose a novel behaviour engineering-based approach, translating software requirements into behaviour trees and then into the Web Ontology Language (OWL). We developed 'requirements defects identifier', a tool that queries the OWL knowledge base to identify potential defects during requirements analysis and change. Validated on three final-year student projects, our approach demonstrated success in detecting all four defect types, offering broader coverage compared to existing tools. A real-world case study has been used to validate the applicability of the proposed approach. Our experiments demonstrate that the tool can successfully detect all four different types of requirement defects during both requirements analysis and requirements change.

INDEX TERMS Behaviour tree, requirements defects, requirements change management, RCM, SPARQL, OWL.

I. INTRODUCTION

Software project success critically depends on the clarity and accuracy of its initial requirements. Defects in software requirements, such as incompleteness, inconsistency, and ambiguity, can significantly increase project risks and lead to failures [1], [2]. These defects arise not only during initial requirement elicitation but also through subsequent requirements changes, posing a continuous challenge throughout the Software Development Life Cycle (SDLC) [3], [4].

The Requirements Change Management (RCM) process may also introduce many requirements defects such as incompleteness, inconsistency, etc., which are also common and faced during the requirements elicitation and analysis phase. According to our previous work [5], requirements inconsistency is one of the major challenges faced during the RCM process in both in-house and global software development.

In the past, several studies have been carried out to address defects detection during the requirements elicitation and analysis phase and the RCM process. Previously, some of the studies use formal logic to analyse and detect requirements defects. For example, Nguyen et al. [6] proposed a description logic based formal approach to detect inconsistency defects during the requirements elicitation phase. Similarly, Goknil et al. [7] proposed a first-order logic-based approach

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

to perform change impact analysis and detect inconsistency defects in requirement analysis and RCM process. Some other techniques [9] also used a similar approach to detect requirements defects.

Formal logic is very useful for detecting requirements defects automatically [10], and all of these approaches produce good results. However, these approaches translate natural language requirements directly to some formal languages such as first-order logic, propositional logic [11] or description logic and then apply reasoning techniques to detect requirements defects. The key issue with these approaches is that the customers who provide the requirements may find it difficult to understand formal languages, while the engineers may not have the domain knowledge to interpret the customers' requirements correctly [12].

Furthermore, some of the existing formal approaches do not provide appropriate tool support. The reason is that they defined requirements defects through natural language descriptions, which might have ambiguous interpretations [40]. Due to this, it seems difficult to formalise defects, and as a result, it becomes challenging to develop an automated tool. Tool support is very important for formal logic-based approaches because an expert level understanding is required to implement formal languages. Without proper tool availability, it becomes very challenging and error-prone [13], [14].

In addition, existing approaches mostly focus on inconsistency defects and miss some other common defects such as incompleteness, redundancy, and ambiguity. Consistency defects cover only 27% of the overall requirements defects [15].

Given this need to address the above-discussed limitations of existing research, in this paper, we propose a Behaviour Engineering-based Requirement Defects Detection (BERDD) approach, which includes the following key contributions:

- We provide a formal definition of the four most requirements defects, such as incompleteness, ambiguity, redundancy, and inconsistency in Behaviour Tree (BT) context. To do this, we extend our previous work that introduces inertia axiom and then define a normal form for Requirement Behaviour Tree (RBT). In this paper, we use this normal formed RBT as a foundation to formalise requirements defects in the context. Requirements defects formalisation helps to detect requirements defects formally and automatically.
- To represent requirements in a formal specification, we propose an algorithm that converts system requirements represented in semi-formal language, i.e.BT, into formal language, Web Ontology Language (OWL). One of the advantages of this approach is that BTs, a semi-formal modelling notation introduced in Behaviour Engineering (BE), has been an ideal bridge to connect software requirements in natural languages to their formal representations [16], [17], [18]. Therefore, it helps the customers to verify the accuracy of the formal representation.
- We develop a tool, Requirements Defects Identifier (RDI), utilizing SPARQL queries on OWL knowledge bases to identify defects, thus reducing the need for expert-level understanding of formal languages.

This approach is validated through application to three final-year student projects, demonstrating its ability to successfully detect all four types of defects. Our approach offers broader coverage of requirements defects, a critical aspect considering these defects constitute a significant portion of total defects in the requirements engineering phase [15], [19].

Lastly, this work provides significant coverage of requirements defects, and according to Hayes [19], these four requirements defects types covered almost 83% of the total requirements defects. Similarly, according to another study, it covers 73% of the total defects in the requirements engineering phase [15]. Based on our literature research; which is given in Section III, no other researches have such high coverage.

The paper is structured as follows: Section II discusses fundamentals of techniques used in this research; Section III reviews existing works; Section IV describes the BERDD approach; Section V introduces the Inertia Axiom and formal defect definitions; Section VI details the developed tool; Section VII presents empirical evaluation; Section VIII discusses limitations and findings; and Section IX concludes with future work directions.

II. BACKGROUND

A. BEHAVIOUR TREE

Behaviour Trees (BTs) play a pivotal role in software engineering, particularly in the precise modeling of software requirements. Derived from natural language, they help mitigate the context-dependence and ambiguity often inherent in such languages. In Behaviour Engineering (BE), two graphical notations are employed: BTs for dynamic behaviors and Composition Trees (CTs) for static system aspects [20], [21]. BE, emerging as genetic software engineering, has evolved to streamline and clarify the requirements elicitation process [22].

BTs have undergone significant development over the years. Key advancements include metamodel extensions for clarity, particularly for the object-oriented community [23], formal semantic definitions [24], and the incorporation of non-monotonic reasoning and probabilistic theories to enhance reliability and performance [25], [26].

Definition: A BT is a formal, tree-like graphical representation that models the behavior of entities in terms of state changes, decision-making, event responses, information exchange, and control flow."

The BT modelling process is exemplified using a simplified case study of a microwave oven [28]. Each system requirement is translated into an individual Requirements

Behaviour Tree (RBT), and then these RBTs are integrated to form an Integrated Behaviour Tree (IBT). For example, consider these two requirements

R1: Originally, the oven is in Idle state and the Door is closed and when the button is pushed, the Power-tube will be energised the oven will start cooking.

R2: If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.

Although we list only two requirements of the oven system, the objective here is to elucidate the BT modelling process with a set of requirements. The BTs are constructed in two steps, translation and integration. The integration step uses two axioms called precondition axiom and integration axiom [29].

In the first step, each system requirement is translated into a separate behaviour tree called a Requirements Behaviour Tree (RBT). For example, Fig. 1 (a) shows the RBT for R1. Originally, the oven is in an Idle state and the Door is closed and when the button is pushed, the Power-tube will be energised, and the oven will start cooking. Similarly, R2 is translated into an RBT shown in Fig. 1 (b). The directed arrows show the connection between individual nodes. In this translation, we have followed the convention of writing component names in the capital.

Another point is that regarding including external entities in BE, external entities can be treated as 'components' in the RBTs, representing their actions as events that trigger system responses [32]. However, in this paper, the requirements do not explicitly mention external agents. Therefore, we chose to model without including them. However, this decision is purely a matter of modelling style and does not affect the capability of using behaviour trees to model requirements. The BE approach's versatility allows for different modelling styles while still accurately capturing the system's behaviour.

In the second step, the individual RBTs are integrated to get a complete BT of the system, which is called an integrated Behaviour Tree (IBT). In this step, nodes with the same behaviour in the RBTs are identified and integrated to form an IBT such as node with component name 'OVEN' and behaviour 'Cooking' used to combine RBT of R1 with RBT of R2, and an IBT based on these two requirements is shown in Fig. 1 (c). The At sign '@' is used to show the integration node in an IBT. Here, we have explained the BT modelling process informally with a simple example; a more detailed discussion about this process can be found elsewhere [29].

The syntax and semantics of BTs are essential for understanding their construction and interpretation. A BT node comprises a component name and associated behavior, qualified by a behavior type. Attributes like traceability links, traceability status, and optional node labels further define the node (as shown in Fig.2). For instance, a node's behavior type can be represented as a state realisation ([...]), event (??...??), guard (???...???), selection (?...?), input (<...>), or output (>...<). Node operators and reversion nodes dictate control flow and connections between nodes.

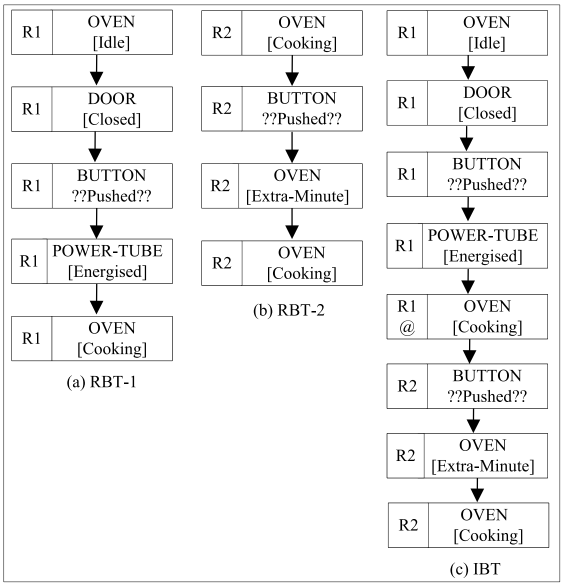
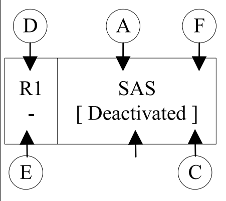


FIGURE 1. Behaviour tree examples.



Label	Description
A	Component Name
B	Behaviour
C	Behaviour Type
D	Traceability Link
E	Traceability Status
F	Operator

FIGURE 2. Attributes of BT node.

In the broader BE-based software design process, as shown in Fig. 3, initial RBTs are created from functional requirements and then integrated into an IBT, which is further developed into a Design Behaviour Tree (DBT). The DBT encompasses both the IBT and additional design decisions, ultimately guiding the creation of comprehensive design documents.

Our research utilizes the foundational steps of translating requirements into RBTs and their integration into IBTs as a basis for developing a requirements defects detection approach. This approach leverages the precision of BTs in modeling and the effectiveness of their syntax and structure in identifying potential defects.

B. WEB ONTOLOGY LANGUAGE (OWL)

Ontologies play a crucial role in representing knowledge, particularly in specialized domains, by defining vocabularies and the interrelationships of terms. In business and software modeling, ontologies facilitate precise knowledge representation and reasoning.

The Web Ontology Language (OWL), a W3C standard, serves this purpose in the Semantic Web, offering well-defined meanings and enabling ontology reasoning. Introduced in 2004, OWL 1 comprised sublanguages OWL Lite, OWL DL, and OWL Full, each varying in expressiveness and computational properties. OWL Lite focuses

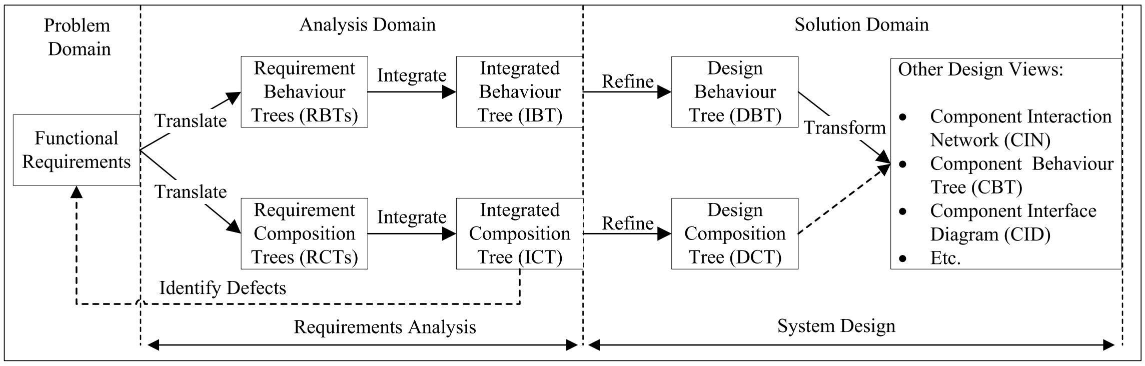

Problem Domain
Analysis Domain
Solution Domain
Functional Requirements
Translate
Translate
Requirement Behaviour Trees (RBTs)
Requirement Composition Trees (RCTs)
Integrate
Integrate
Integrated Behaviour Tree (IBT)
Integrated Composition Tree (ICT)
Refine
Refine
Design Behaviour Tree (DBT)
Design Composition Tree (DCT)
Transform
Other Design Views:
Component Interaction Network (CIN)
Component Behaviour Tree (CBT)
Component Interface Diagram (CID)
Etc.
Identify Defects
Requirements Analysis
System Design

FIGURE 3. Behaviour engineering process.

on basic classification and constraints, while OWL DL aligns with description logic, balancing expressiveness with computational completeness and decidability. OWL Full offers the greatest expressiveness but lacks computational guarantees.

The evolution to OWL 2 in 2009 marked a significant increase in expressiveness. Unlike OWL 1, OWL 2 does not segregate into sublanguages, and tools such as Protégé, along with reasoners like Pellet [30] and HermiT [31], have supported its application. In our research, we utilize OWL 2 for representing software requirements, leveraging its formal structure to define classes, properties, individuals, and their interrelations.

The primary rationale for choosing OWL in this research over other formal languages is multi-fold:

- Propositional logic lacks expressiveness and struggles with large datasets [33], while first-order logic, despite its expressiveness, faces computational challenges.
- OWL's suitability for defining domain knowledge and semantics in requirements analysis is well-established [34].
- The availability of off-the-shelf OWL reasoners aids in practical application [30], [31], [35].
- OWL is recognized as a prominent ontology language in requirements engineering processes, as evidenced by systematic literature reviews [36].

This research employs OWL 2 to formalize software requirements, providing a structured approach to identify and analyze requirements defects. The choice of OWL is grounded in its expressiveness, computational properties, and the supportive ecosystem of tools and reasoners, making it a fitting choice for this domain.

C. REQUIREMENTS DEFECTS CLASSIFICATION

This subsection discusses the classification of requirements defects, highlighting their diverse definitions in literature. We focus on the four most common types of requirement defects: incompleteness, ambiguity, redundancy, and inconsistency. These defects, along with their subcategories, are listed in Table 1.

In our research, we specifically address these defects with a nuanced understanding of their subcategories. For incompleteness, we focus on both the individual requirement aspects (1a and 1b) and the system-wide requirements perspective (1c). Ambiguity is addressed in two forms: through the translation phase of the BT process for definitions 2a and 2b, and directly in our research for definition 2c. Inconsistency is tackled in relation to conflicting stakeholder requirements (3b) and overall document coherence (3c), while ambiguity-related inconsistency (3a) is covered under the ambiguity defect. Redundancy is comprehensively addressed in both its forms (4a and 4b).

The coverage of our approach is significant, as these defect types collectively account for a major portion of total requirements defects. We aim to automate the detection of these defects with minimal human intervention, focusing on areas that can be effectively addressed through our approach. For instance, while certain aspects like the overall system requirement completeness may require domain knowledge, individual requirement completeness can be automatically checked. Similarly, we believe our approach effectively addresses significant aspects of consistency and redundancy.

This detailed classification and our approach's focus areas lay a solid foundation for our automated detection method, contributing to enhanced accuracy and efficiency in the requirements analysis process.

III. RELATED WORK

The research landscape in requirements engineering (RE) defect detection is diverse, with methodologies spanning formal languages, Natural Language Processing (NLP), and SPARQL-based analyses. This body of work forms the foundation upon which our research builds and can be broadly categorized into three distinct areas:

1) **Formal Languages for Automated Analysis:** This category includes research that leverages formal languages to automate the detection of RE defects. Approaches vary from direct translations of requirements to formal languages to indirect methods using semi-formal languages as intermediaries.
2) **Natural Language Processing (NLP) Techniques:** Studies in this category apply NLP techniques to parse, interpret, and analyze requirements written in natural language, aiming to identify and rectify defects.

TABLE 1. Requirements Defects Classification.

No.	Defect Type	Definitions
1	Incompleteness	1a. Information relevant to the requirement is missing [37]. 1b. A requirement must have all relevant components [38]. 1c. Required behaviour and output for all possible states under all possible constraints [19], [39].
2	Ambiguity	2a. The requirement contains information or vocabulary that can have more than one interpretation [37]. 2b. The information in the requirement is subjective [19], [37]. 2c. Two different terms are used to refer to the same thing [40].
3	Inconsistency	3a. Two or more requirements in a specification contradict each other due to inconsistent use of words and terms [19], [41]. 3b. Two or more stakeholders have different, conflicting requirements [42]. 3c. The requirement or the information contained in the requirement is inconsistent with the overall document [37].
4	Redundancy	4a. The requirement is a duplicate of another requirement or part of the information it contains is already present in the document becoming redundant. [37]. 4b. Two different executions are redundant if they produce indistinguishable functional results from an external viewpoint [43].

3) **SPARQL-based Requirement Analysis:** The third category encompasses research utilizing SPARQL queries for diverse analyses on software requirements, offering a unique approach to understanding and rectifying requirements issues.

Each of these categories represents a different strategy in tackling the challenges of RE defect detection. The upcoming subsections will delve into these approaches in detail, discussing their methodologies, strengths, and limitations. This review not only contextualizes our research within the existing academic discourse but also highlights the innovative aspects of our approach in addressing RE defects.

A. REQUIREMENTS TRANSLATION DIRECTLY TO FORMAL LANGUAGES

Translating software requirements directly into formal languages, such as first-order logic, propositional logic, and description logic, is a prevalent approach in requirements analysis. These logics facilitate detailed analysis and reasoning, yet present challenges in practical application.

Chen et al. [44] introduced SafeNL for specifying safety requirements of railway interlocking systems in natural language, which are then converted into formal constraints. While effective in detecting inconsistencies using model checking tools, this approach relies heavily on the representational power of SafeNL.

Zowghi et al. [11] proposed a technique using propositional logic to detect inconsistencies, implemented in the CARL tool. However, the limited expressiveness of propositional logic restricts its ability to model complex systems [33].

Nguyen et al. [6] developed REInDetector, a tool translating requirements into DL-based ontologies for defect analysis. Yet, this approach struggles with requirements that do not readily translate into DL, like those with temporal operators [45].

Chanda et al. [46] employed regular grammars to ensure syntactic consistency across different UML models. While effective in maintaining syntactic correctness, this method does not address semantic inconsistencies.

These studies illustrate the potential of formal languages in detecting requirements defects. However, their successful application often demands an expert-level understanding of formal logics, posing a challenge for stakeholders not versed in these languages [13]. This gap can lead to difficulties in verifying the correctness of formal representations, especially when the engineers interpreting these requirements lack domain-specific knowledge.

Our research seeks to address these challenges by developing an approach that balances the rigor of formal languages with the accessibility and interpretability needed by stakeholders, thereby enhancing the practicality and applicability of requirements defect detection.

B. REQUIREMENTS TRANSLATION USING SEMI-FORMAL LANGUAGES AS A BRIDGE

Semi-formal languages serve as an intermediary in translating natural language requirements into formal specifications. This two-step process typically involves first converting requirements into a semi-formal representation, followed by a translation into formal language for defect analysis.

Khan and Porres [47] utilized OWL to analyze the consistency of UML models (state and class diagrams). They translated UML diagrams into OWL, performing consistency checks with reasoning engines like Pellet and HermiT. This approach, while effective for UML models, is limited by the expressiveness of the OWL translations.

Kaneiwa and Satoh [48] adopted a similar strategy for class diagrams, translating them into first-order logic. Despite its effectiveness, this method's applicability is limited to UML diagrams with basic constructs, highlighting scalability issues in more complex scenarios.

Kamalrudin et al. [49] developed MaramaAIC, a tool for identifying inconsistencies in natural language requirements. By extracting essential use case patterns, this approach identifies inconsistencies but is constrained by its reliance on a pre-defined interaction library.

Liu et al. [50] proposed a technique to detect defects in use case diagrams by translating them into activity diagrams through dependency parsing. This approach is limited by the scope of its parsing rules, which cover only a subset of use case constructs.

Kroha et al. [51] implemented an OWL-based method to ensure the consistency of UML models. They transformed

UML models into textual requirements, then built an ontology for consistency checks using OWL reasoners. While this method bridges the gap between UML models and formal languages, its effectiveness is dependent on the accurate transformation of models into textual form.

These studies demonstrate the potential of using semi-formal languages like UML as intermediaries in requirements analysis. However, a challenge arises from the inherent complexity and variety of UML diagrams [52], [53], [54]. Unlike UML's extensive range of diagrams, Behaviour Trees (BTs) employ a minimal set of coherent notations, simplifying the detection of inconsistencies [27], [55], [56]. This streamlined approach in BTs, compared to the multifaceted nature of UML, underlines the efficiency of using a more focused set of modeling notations, which is a key aspect of our research in detecting requirements defects.

C. REQUIREMENTS DEFECTS ANALYSIS USING NATURAL LANGUAGE PROCESSING

Natural Language Processing (NLP) has increasingly been employed in detecting requirements defects, with various studies adopting different techniques. Mavin et al. [57] utilized constrained natural language, Arora et al. [58] employed predefined templates, and Tjong and Berry [59] applied a rule-based approach for detecting RE defects.

Hasso et al. [60] proposed a rule-based method targeting the German language, while Femmer et al. [61] achieved 59% precision with their rule-based detection approach. Ferrari et al. [62] implemented an NLP pattern-based approach in the railway domain, achieving a notable 83% precision.

Despite these successes, NLP-based approaches face inherent limitations. Zhao et al. [63] conducted a study revealing that current NLP research in RE is predominantly applied on a small scale, with only 7.18% of studies involving industrial case studies for evaluation. A major limitation is the focus on syntactic rather than semantic analysis, which is crucial for comprehending the context-dependent nature of natural language requirements [63]. Furthermore, Dalpiaz et al. [64] highlighted gaps in current NLP research, including the lack of appropriate performance metrics and a deep understanding of the context-dependency in natural language requirements.

These limitations underscore the challenges in applying NLP techniques for requirements analysis. While NLP provides a valuable tool for syntactic analysis, its application in semantic interpretation and context understanding remains an area for further development. This gap motivates our research to explore alternative or complementary approaches that can address these challenges in requirements defects detection.

D. APPLICATION OF SPARQL FOR REQUIREMENTS ANALYSIS

SPARQL, a semantic query language, has been increasingly utilized in software requirements analysis, though its application in defect detection has been limited. Several studies have explored the use of SPARQL for analyzing UML diagrams and natural language requirements, each with distinct focuses and methodologies.

Wei et al. [65] proposed a method to analyze various UML diagrams, including class, sequence, and state machine diagrams. By converting these diagrams into OWL, they used SPARQL queries to perform diverse analyses. However, their work did not extend to detecting requirements defects. Similarly, Sadowska and Huzar [66] employed SPARQL for analyzing class diagrams, but again, without a focus on defect detection.

Siegemund et al. [67] took a more direct approach to requirements defect detection. They transformed natural language requirements into goal-oriented models and developed SPARQL rules for identifying inconsistencies and incompleteness. Verma and Kass [68] introduced a controlled syntax for writing software requirements, with SPARQL queries used to check for inconsistencies and incompleteness based on this syntax.

Despite these applications, SPARQL's use in detecting requirements defects remains underexplored. Only a few studies, such as those by Siegemund et al. and Verma and Kass, have directly addressed defect detection, focusing primarily on inconsistency and incompleteness. A notable limitation of current SPARQL-based approaches is the requirement for analysts to possess SPARQL knowledge, which can be a barrier to broader adoption. Furthermore, the existing studies lack comprehensive tool support, limiting their practical utility.

Given these gaps, there is a clear opportunity for developing a more accessible and holistic approach that leverages SPARQL for comprehensive requirements defect detection. This research aims to address these challenges by proposing a methodology that not only covers a wider range of defect types but also enhances tool support, making the analysis process more accessible to analysts without extensive SPARQL expertise.

IV. BERDD OVERVIEW

This section presents BERDD (Behaviour Engineering-based Requirements Defects Detection), a novel approach devised for identifying and addressing requirements defects in the phases of requirements analysis and Requirements Change Management (RCM). We first outline the overall structure of BERDD, elucidating its foundational components, and then proceed to detail its workflow, demonstrating the step-by-step process in identifying and managing requirements defects.

A. BERDD STRUCTURE

The BERDD approach is structured around four key domains, as illustrated in Fig. 4: the problem domain, techniques domain, tools domain, and solution domain.

- **Problem Domain:** This domain encompasses the original and new software requirements. BERDD is

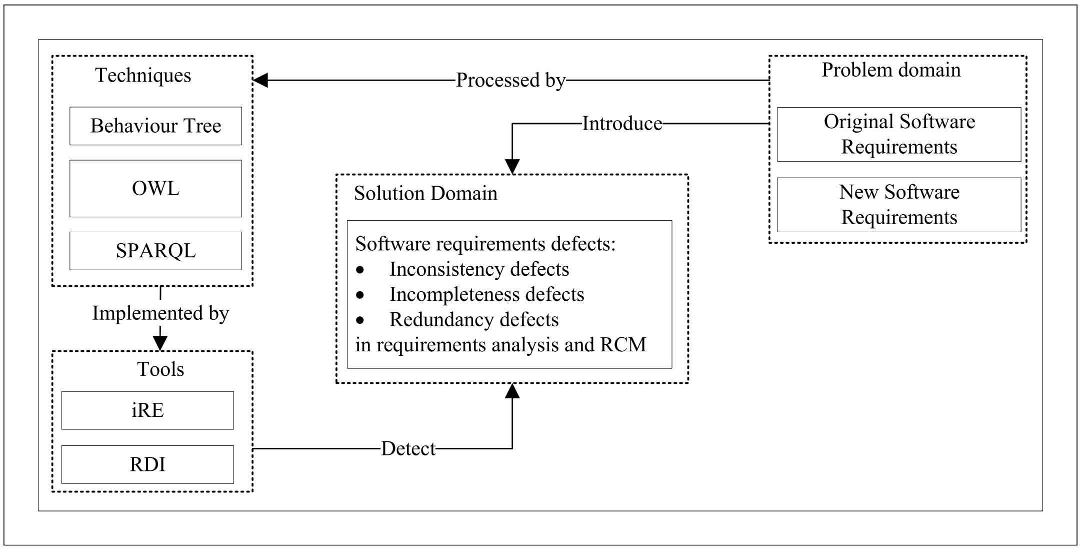


FIGURE 4. Key elements of BERDD.

primarily designed for defect detection during the requirements analysis phase but is also applicable to the Requirements Change Management (RCM) process. The new or modified requirements in RCM are treated as new software requirements, setting the stage for defect detection.

- **Techniques Domain:** Three techniques constitute this domain. Behaviour Trees are utilized for modeling software requirements, the Web Ontology Language (OWL) for formal representation, and the SPARQL query language for querying the OWL-based knowledge base to detect defects.
- **Tools Domain:** Two tools facilitate the implementation of these techniques. The iRE tool, previously published [16], is employed for designing and modifying Behaviour Trees and mapping them to XML. The newly developed Requirement Defects Identifier (RDI) translates BTs from XML to OWL and executes SPARQL queries to identify defects.
- **Solution Domain:** This domain contains the identified requirements defects. While this paper focuses on algorithms for detecting the four most common defect categories (incompleteness, inconsistency, redundancy, and ambiguity), the approach is adaptable to other defect types with appropriate algorithms.

In summary, BERDD integrates these domains to provide a comprehensive approach for detecting and addressing defects in software requirements, both during initial analysis and subsequent changes.

B. BERDD WORKFLOW

The BERDD methodology comprises a three-step workflow, depicted in Fig. 5, each step focusing on a specific aspect of requirements defects detection.

1) **Step 1: Designing and Modifying RBTs:** This initial step involves the creation or modification of Requirement Behaviour Trees (RBTs) using tools like iRE [16]. In the context of Requirements Change Management (RCM), existing RBTs are updated to reflect changes. A critical prerequisite for applying BERDD during RCM is the availability of all RBTs and the Integrated Behaviour Tree (IBT) based on previous requirements. This step also includes the integration of new RBTs with the existing IBT. Some defects, particularly incompleteness, may become apparent during the translation of requirements to RBTs or during their integration into an IBT. The final IBT is then converted to an XML file using iRE.
2) **Step 2: Formalization of Defects:** In this step, requirements defects are formalized within the context of Behaviour Trees. This process is independent of the specific IBT and does not require any prerequisites, allowing it to be conducted concurrently with Step 1.
3) **Step 3: Utilizing RDI for Defect Detection:** The third step involves the prototyping tool RDI, introduced in Section VI. RDI translates the IBT from XML format into an OWL representation and employs query languages like SPARQL to access the OWL knowledge base, enabling the detection of formalized requirements defects. While SPARQL queries are used in this research, other query languages such as DL query can also be applied.

The first step aligns with standard BE methodologies, as detailed in the background section and previous publications [16], [29]. The specifics of Step 2 will be elaborated in the following section, while Step 3, which involves the application of the RDI tool, will be discussed in Section VI.

V. REQUIREMENTS DEFECTS

This section delves into Step 2 of the BERDD process, focusing on formally defining the four most prevalent types of requirements defects within the context of Behaviour

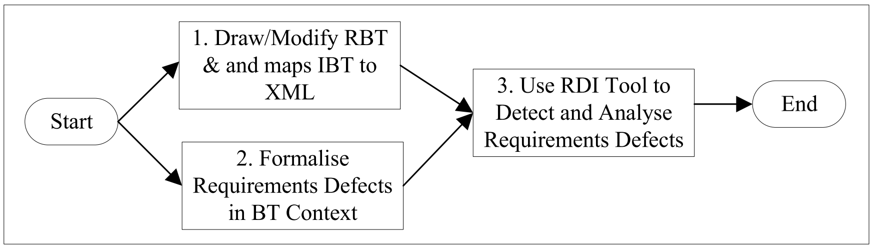

Start
1. Draw/Modify RBT & and maps IBT to XML
2. Formalise Requirements Defects in BT Context
3. Use RDI Tool to Detect and Analyse Requirements Defects
End

FIGURE 5. **BERDD workflow.**

Trees (BTs), and outlining the algorithms designed for their detection. The foundation of this step is the Inertia Axiom, which is instrumental in establishing a normal form for Requirement Behaviour Trees (RBTs). This normal form serves as a baseline for identifying and defining the various types of requirements defects.

In the subsequent subsections, we will explore each of these common defects - incompleteness, inconsistency, redundancy, and ambiguity - and formalize them within the BT framework. Utilizing semi-formal or formal notations is advantageous for defect detection, as they typically offer greater clarity and precision compared to natural language descriptions of requirements. These theoretical concepts and defect types will be illustrated with simple examples for ease of understanding.

The practical application of these concepts, including a more complex case study and the utilization of the developed tool for defect detection, will be presented in Section VII. That section will also discuss how SPARQL queries are employed within the BERDD framework to effectively identify and analyze these defects.

A. NORMAL FORM OF REQUIREMENT BEHAVIOUR TREE

This subsection introduces the concept of a normal formed Requirement Behaviour Tree (RBT), based on the newly introduced Inertia Axiom. The Inertia Axiom complements the existing axioms of Precondition and Integration, established in the original BT approach [29], by adding a crucial dimension to the analysis of software requirements.

Definition (Inertia Axiom): "A system will remain in a state unless influenced by an external or internal event, potentially transitioning into a new state, which may be the same as or different from the original state."

This axiom aligns with fundamental software engineering principles, emphasizing the importance of analyzing each requirement in terms of its precondition, triggering event, and resulting postcondition. It provides a structured framework for understanding and modeling software behavior.

Normal Formed RBT: "An RBT is normal formed if it comprises three sequential components: precondition, event, and postcondition, each delineated by their respective behavior types."

The precondition and postcondition must include at least one state realization node, whereas the event typically involves nodes denoted as events, selections, or guards.

To illustrate this concept, we refer to a simplified requirement from a microwave oven case study [28]:

R5: "Whenever the oven is cooking, opening the door stops the cooking."

In Fig. 6 (a), the RBT for R5 demonstrates this structure: the precondition shows the "OVEN" in a "Cooking" state, followed by the event of the "DOOR" being "Opened," leading to the postcondition of "Cooking-Stopped." Similarly, the RBT for another requirement (R1) is depicted in Fig. 6 (b), showcasing multiple nodes in its precondition and postcondition.

A normal formed Integrated Behaviour Tree (IBT) is an extension of this concept, constructed by integrating a finite set of normal formed RBTs. This structure significantly enhances the clarity and quality of requirements modeling, aiding in the formalization and subsequent detection of defects. While this research primarily focuses on the four most common requirements defects, the methodology can be extended to other types of defects that can be defined based on the normal formed IBT structure. This alignment with the BT taxonomy [27] ensures consistency in modeling notation and principles.

B. INCOMPLETENESS DEFECT AND ITS DETECTION

1) DEFINITION

Incompleteness in requirements can be viewed from two perspectives: the overall system requirements and individual requirements. This research focuses on the latter, utilizing the normal formed RBT structure as defined by the Inertia Axiom. According to this axiom, a complete individual requirement in an RBT should include a precondition, an event that triggers a state transition, and a postcondition.

Definition: A requirement is deemed incomplete if any of these components-precondition, event, or postcondition-is missing.

The above-listed definition of incompleteness defect also covers the definition listed in Table 1. The definition 1c listed in Table 1 considered input and output, but only listed as output parameters. The absence of a precondition or postcondition is more commonly observed than a missing event. To evaluate a requirement for completeness, the analyst should determine based on domain knowledge if each of these components is clearly defined. If any component is unclear or absent, the requirement is classified as incomplete.

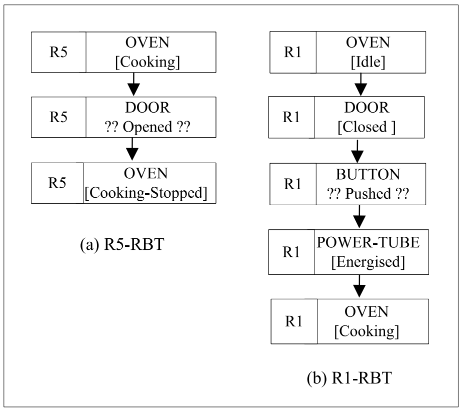


FIGURE 6. **Normal formed RBT examples.**

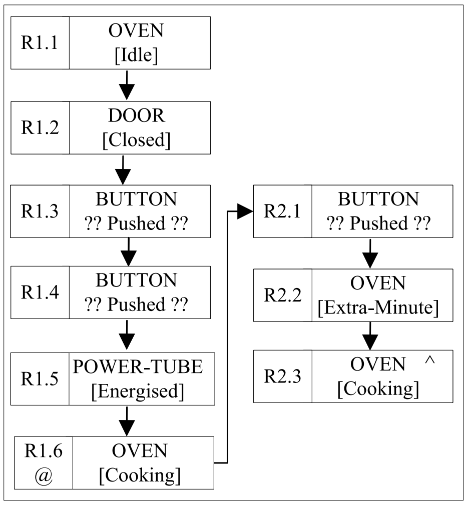


FIGURE 7. **Oven IBT-Incompleteness.**

Incompleteness related to the missing precondition in the BT context can be identified if the behavior type of the first node in a new RBT is not a state realization, or if the root node of an RBT does not correspond with any node in an IBT. If the root node's behavior type is a state realization but does not match any IBT node, it may indicate a root requirement or an incomplete requirement. Missing event or postcondition is identified if the RBT lacks an event node after the precondition or a state realization node after an event node, respectively.

2) EXAMPLE

To demonstrate how incompleteness is identified in BERDD, we consider a case study of a microwave oven [17]. In this example, we apply BERDD in the context of requirements change. Initially, we have two functional requirements for the microwave oven, as shown in Table 2.

The corresponding IBT for these requirements is shown in Fig. 7.

A new requirement (**R3:** Opening the door stops the cooking) is introduced during a requirements change, with its RBT depicted in Fig. 8.

Evaluating R3's RBT against the normal formed RBT criteria reveals that the root node lacks a state realization behavior type, indicating a missing precondition. The responsibility then falls to the requirements analyst to either integrate the missing precondition based on domain knowledge or consult the stakeholder for clarification.

This practical application illustrates how the BERDD framework facilitates the detection of incompleteness defects in individual requirements, enhancing the overall quality and coherence of software requirement specifications.

C. AMBIGUITY DEFECT AND ITS DETECTION

1) DEFINITION

Ambiguity in software requirements, particularly when described in natural languages, can lead to misinterpretations and system failures. Ambiguity can manifest in several forms, such as semantic ambiguity, where a sentence could have multiple interpretations, or terminological ambiguity, where different terms are used to refer to the same concept, or a single term refers to multiple entities.

In this research, we focus on the last two types of ambiguities, as semantic ambiguity is typically resolved during the translation of requirements into RBTs. In the context of BT, where behavior types are predefined but component and behavior names are problem-specific, these ambiguities can arise in both component and behavior names. The definition of ambiguity in this context is:

Definition: Requirements are ambiguous if two different terms refer to the same component or behavior name, or if the same term refers to different component names or behavior names.

It is essential to analyze component and behavior names together to accurately identify ambiguities. Separate consideration of these elements might not provide a clear understanding of the ambiguity involved.

2) EXAMPLE

To illustrate ambiguity detection, we consider the Security Alarm System (SAS) as an example. In this scenario, we apply BERDD during the requirements analysis phase. The SAS has four functional requirements as shown in Table 3.

A potential ambiguity arises with the component names "3-DIGIT-CODE" and "CODE". The ambiguity is identified when these terms are analyzed in conjunction with the behavior name. For instance, if both terms are associated with the behavior name "Entered" and the behavior type "Event" in the IBT, it suggests that they refer to the same component. The corresponding IBT is shown in Fig. 9.

Upon identifying the ambiguity, the requirements analyst must conduct a thorough analysis to decide how to address the potential confusion. This process demonstrates the importance of contextual analysis in ambiguity detection within the BERDD framework.

TABLE 2. Microwave Oven Initial Requirements.

ID	Requirement
R1	If the oven is idle with the door closed and you push the button, the power tube will energize and the oven will start cooking.
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.

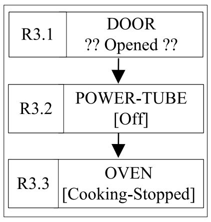


FIGURE 8. RBT of R3.

D. REDUNDANCY DEFECT AND ITS DETECTION

1) DEFINITION

In software engineering, redundancy is often utilized to enhance fault tolerance, reliability, and self-healing capabilities in self-adaptive systems [69], [70], [71], [72]. However, the concept of redundancy in requirements is distinct from this general software redundancy. While software redundancy is a deliberate design choice, redundancy in requirements refers to unintended duplication, which can lead to confusion and inefficiency when mistakenly considered as two different requirements. The redundancy issue is more problematic during system maintenance as compared to the system development phase.

Definition: In the context of requirements, redundancy is defined as the duplication of a requirement, where two requirements are redundant if they have identical preconditions, events, and postconditions.

Requirements redundancy typically emerges during the requirements elicitation phase and should be identified and resolved in the analysis phase. In some cases, a single requirement may express multiple events, leading to overlap with other requirements.

2) EXAMPLE

To demonstrate redundancy defects, we use the microwave oven system as an example. Three functional requirements are shown in Table 4, and their corresponding Integrated Behaviour Tree (IBT) is depicted in Fig. 10.

Upon examining the IBT, a potential redundancy defect is identified. The nodes R1.2, R1.3, R1.4, and R1.5 have similarities with nodes R3.5, R3.6, R3.7, and R3.8. This redundancy indicates that parts of these requirements may be expressing the same actions or states, potentially leading to unnecessary duplication.

In this context, redundancy does not necessarily imply error but could indicate areas where requirements can be consolidated for clarity and efficiency. This paper not only addresses requirements during system analysis but

TABLE 3. SAS Requirements.

ID	Requirement
R1	Whenever the SAS is deactivated, it will be activated by pressing the set button.
R2	If a trip signal occurs while the SAS is set, a high-pitched tone (alarm) is emitted.
R3	A three-digit code must be entered to turn off the alarm.
R4	Whenever the SAS is activated, correct entry of the code deactivates the SAS.

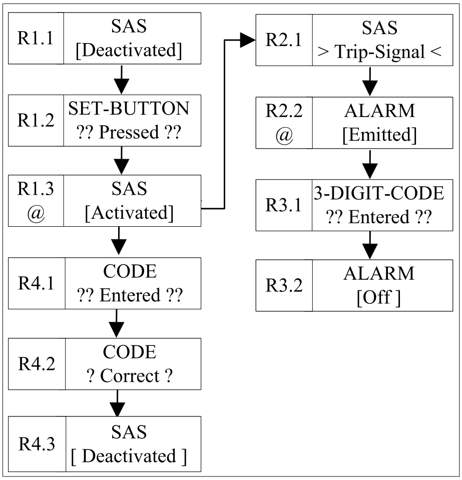


FIGURE 9. SAS IBT – ambiguity.

also during requirements change management. Consider a scenario, when the user asks for a requirements change and the requested change is related to the same two components. If we do not identify redundancy defect, then maybe we only incorporate change in one instance and leave other instances unchanged, which results in inconsistency in system features.

This example demonstrates how the BERDD framework, through the analysis of RBTs and IBTs, facilitates the identification of redundancy in software requirements.

E. INCONSISTENCY DEFECT AND ITS DETECTION

1) DEFINITION

Detecting inconsistencies in requirements is a critical aspect of requirements engineering. Inconsistencies often arise when different stakeholders have conflicting requirements. In the context of a normal formed Behaviour Tree (BT), inconsistency can be formally defined as follows:

Definition: Requirements are inconsistent if they share the same precondition and event but lead to different postconditions, or if they share the same event and postcondition but originate from different preconditions.

This definition aligns with the type (3c) inconsistency as discussed in the background section, focusing on conflicts between stakeholder requirements [42].

TABLE 4. Oven Requirements - Redundancy.

ID	Requirement
R1	If the oven is idle with the door closed and you push the button, the oven will start cooking (that is, the power tube is energised).
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	While the oven is cooking, if the user opens the door, the cooking will be paused, and it resumes if the door is closed and the user pushes the button.

TABLE 5. OVEN Requirements - Inconsistency.

ID	Requirement
R1	If the oven is idle with the door closed and you push the button, the oven will start cooking (that is, the power tube is energised).
R2	While the oven is cooking, if the user opens the door, the oven pauses the cooking and resumes if the door is closed and the user pushes the button.
R3	Whenever the oven is cooking and opening the door will stop the cooking.

2) EXAMPLE

To illustrate the detection of inconsistency defects, we use the microwave oven system as an example. The system's three functional requirements are listed in Table 5, with the corresponding Integrated Behaviour Tree (IBT) displayed in Fig. 11.

Analysis of the IBT indicates a potential inconsistency. The nodes R1.5, R2.1, R2.2 are similar to nodes R1.5, R3.1, R3.2, but the subsequent nodes R2.3 and R3.3 differ. This discrepancy indicates that the same precondition and event lead to different postconditions, signaling an inconsistency.

Upon identifying this inconsistency, the requirements analyst is tasked with resolving the conflict, ensuring that the requirements are coherent and aligned with the system's intended functionality.

VI. TOOL (RDI) ENVIRONMENT

This section presents the newly developed tool, RDI (Requirements Defects Identifier), which plays a pivotal role in executing step 3 of the BERDD workflow (refer to Fig. 5). The focus is first on delineating the RDI workflow, followed by a detailed exploration of its functionality in translating IBT-XML to OWL.

A. RDI WORKFLOW

The RDI tool, illustrated in Fig. 12, is designed with a five-step workflow to facilitate the detection and management of requirements defects.

1) **Step 1: Requirements Gathering or Change Capture:** This initial phase involves either collecting original software requirements or capturing requirement change requests from stakeholders. In cases of requirement changes, additional analysis is performed to categorize the type of request, such as new requirements or modifications to existing ones.
2) **Step 2 (a, b): RBT Design or Modification:** Here, a new RBT is created, or an existing one is modified using BT drawing tools like iRE [16]. The integration of these RBTs into an IBT and their conversion to XML format is also accomplished in this step. RDI works in conjunction with iRE to provide comprehensive support for requirements analysts during both the analysis and change phases.
3) **Step 3: XML to OWL Translation:** The XML file, representing the IBT, is converted into an OWL representation based on predefined translation rules. This step bridges the gap between semi-formal representations and formal ontologies.
4) **Step 4: SPARQL Query Construction:** Users select a defect type and input related parameters using the RDI interface, from which SPARQL queries are automatically generated. These queries are executed using a SPARQL query engine (dotNetRDF), and the results are presented to the user. This process simplifies the query construction, eliminating the need for users to have extensive knowledge of SPARQL.
5) **Step 5: Analysis of Query Results:** Finally, the requirements analyst evaluates the query results and decides on the appropriate actions to resolve the detected defects, utilizing domain knowledge.

Steps 2 through 5 demonstrate the seamless integration of RDI with the iRE tool and the process of defect detection and resolution. Details of steps 4 and 5 will be further elucidated with a case study in the following section. RDI was developed using Visual Studio 2019 as a Windows form application in C#. The dotNetRDF library [73], integrated as a .dll file, facilitates the execution of SPARQL queries. This subsection provides an overview of RDI's key elements and workflow, with more comprehensive details and tool screenshots to be presented in the subsequent section.

B. BEHAVIOUR TREE REPRESENTATION IN OWL

This subsection delves into step 3 of the RDI workflow, which involves translating an Integrated Behaviour Tree (IBT) from XML to OWL. This translation is vital for semantic analysis and querying within the RDI tool.

The key to this translation lies in understanding the BT taxonomy, derived from a previously published metamodel [23]. From this taxonomy, four critical *meta-classes* are identified:

1) Component Name
2) Behaviour Name
3) Behaviour Type
4) BT Nodes

Each node in an IBT consists of attributes corresponding to these meta-classes, and in the OWL representation, these attributes are denoted as *individuals*. Relationships between

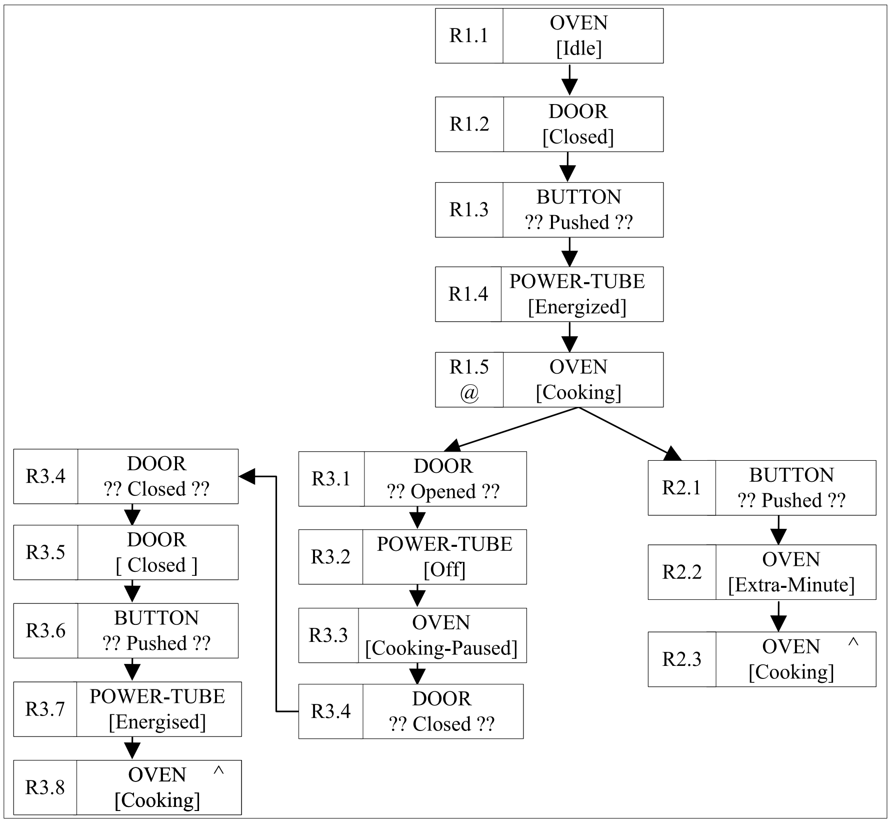


FIGURE 10. Oven IBT - redundancy.

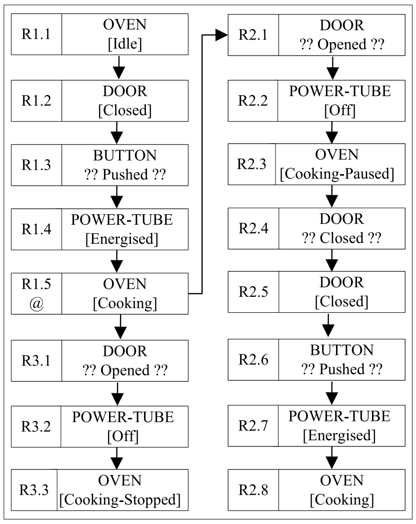


FIGURE 11. Oven IBT - Inconsistency.

individuals are established using *object properties*. The primary object properties used are hasComponentName, hasBehaviourName, hasBehaviourType, and nextNode.

The translation process, modeled on UML to OWL conversion methodologies [9], [66], is detailed in Algorithm 1. This algorithm takes an XML-formatted IBT as input and outputs its OWL representation. The algorithm executes the following steps:

The process involves declaring meta-classes and object properties in OWL, initializing an empty list for node attributes, and sequentially reading and processing each node from the XML file. Each node's attributes are declared as OWL individuals, and relationships are established using the defined object properties.

This translation forms a foundation for semantic querying and analysis in the RDI tool, enabling sophisticated defect detection and resolution in software requirements.

VII. EVALUATION

In this section, we evaluate the BERDD approach's effectiveness in detecting and analyzing Requirements Engineering (RE) defects using the RDI tool and SPARQL queries. The RDI tool enables users to input defect-related parameters, automatically generating and executing queries to detect defects. While this section focuses on queries for incompleteness defects, additional queries for other defect types are provided in Appendix B.

A. CASE STUDY: AMBULATORY INFUSION PUMP (AIP)

To demonstrate the BERDD approach and RDI tool, we applied them to three undergraduate final-year projects, with a detailed discussion on one of the projects, the Ambulatory Infusion Pump (AIP). The AIP, a medical device used in drug therapy, has nine main functional requirements, as shown in Table 6. For our evaluation, we considered seven

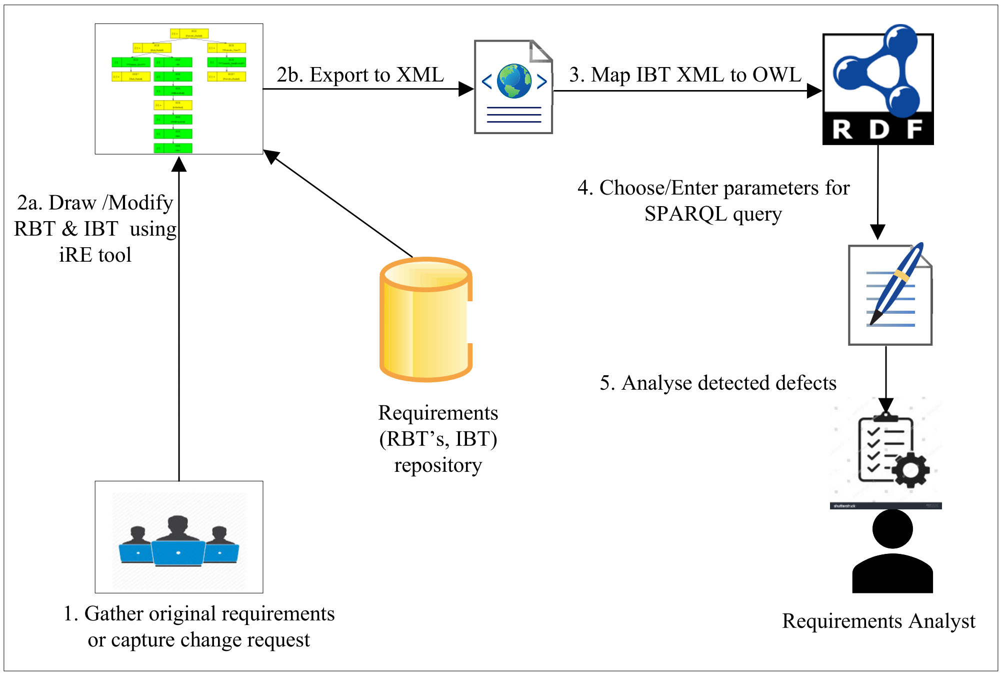

2b. Export to XML
3. Map IBT XML to OWL
RDF
4. Choose/Enter parameters for SPARQL query
2a. Draw /Modify RBT & IBT using iRE tool
Requirements (RBT's, IBT) repository
5. Analyse detected defects
1. Gather original requirements or capture change request
Requirements Analyst

FIGURE 12. RDI workflow.

initial requirements and treated two additional requirements (R8 and R9) as changes introduced during the Requirements Change Management (RCM) process.

Using the iRE tool, we created Behaviour Trees (BTs) for the initial seven requirements and integrated them into an Integrated Behaviour Tree (IBT), as detailed in Appendix C. The IBT for these requirements contains 50 nodes. The BTs for the new requirements (R8 and R9), with 18 nodes, are shown in Fig. 13 and Fig. 14.

B. DEFECT DETECTION WITH RDI TOOL

Using RDI, we first checked for incompleteness defects in the new requirements. This process involved selecting parameters and analyzing the results generated by the SPARQL queries. The RDI indicated that R9 was complete, while R8 was incomplete, lacking a precondition (Fig. 15 and Fig. 16). The SPARQL query used for this analysis is presented in Algorithm 2.

Looking at Algorithm 2, the first four lines are default prefixes of any SPARQL query, and line 5 is the prefix related to our ontology defined in OWL syntax. The real query starts from line 6. The SELECT clause contains three variables (?cname, ?bname, ?btype), and the variables written in the select clause show the title of columns in the result list. The triple patterns and other conditions are enclosed in the WHERE clause that starts from line 7. The triple patterns are between lines 8 and 11. The condition on variables is in the FILTER clause between lines 12 and 14. In the FILTER clause, the @parm1, @parm2 and @parm3 are the values the user will enter through RDI interface. This query will return an empty result for R8 because no node with these parameters exists in the AIP IBT.

We then look at ambiguity defects detection. We have implemented two different techniques in RDI, either based on component name or based on behaviour name. After integrating new requirements with the existing IBT, we will check ambiguity defects based on either technique. Using the component name-detection technique, we selected the BEPPER component to check whether the behaviour names associated with this component have clear and precise semantics or potential ambiguity. A thorough analysis revealed two potential ambiguities, one is with R9.6 and R9.8 shown in Fig. 18, where the behaviour names are Beep3Times and ThreeBeeps, respectively, and both behaviour names might be referred to the same thing but with the use of two different terms. The other is two different behaviour names (BeepContinuously, ContinuousBeep) used with the same component BEEPR, as shown in Fig. 17. It seems that both behaviour names referred to the same thing. In both cases, requirement analysts can fix the defects based on their domain knowledge.

After that, we check the redundancy defects. The two new RBTs created potential redundancies in the AIP system, as shown in Fig. 19. The RDI found two potential redundancies in the AIP system, and both involve the new requirements (R8 and R9). R9 has the same precondition, event, and postcondition as in R2 and creates potential redundancy. Similarly, R8 has the same precondition, event and postcondition as in R2, R5, and R9. Finally, we check the inconsistency defects. RDI has found one potential inconsistency (shown in Fig. 20). The detected inconsistency

Algorithm 1 Translate an IBT Into OWL

```
1:  Input: XML file of IBT
2:  Output: OWL file of IBT
3:  // Declare four meta-classes
4:  for each meta-class do
5:      Insert "Declaration (Class (:meta-name))"
6:  // Declare object properties
7:  for each object property do
8:      Insert "Declaration (Property (:property_name))"
9:  // Initialize empty node attributes (atts) list
10: $L\_atts := \emptyset$
11: // Read XML file
12: while not EOF do
13:     // Read node properties tag from XML file
14:     Node = XML_Node Tag
15:     // Check BT node attributes
16:     for each node att do
17:         if Node(att) not in $L\_atts$ then
18:             Declaration (Individual (:Node(att=)))
19:             Declaration (:Node(attribute) :meta-name))
20:             Add($L\_atts$, Node(att))
21:     // Declare node ID as an individual
22:     Declaration (Individual (:Node(NID)))
23:     Declaration (:Node(NID) :BTNodes))
24:     // Associate node attributes with the node ID
25:     for each node att do
26:         Insert "Property (:Node(NID) :Node(att))"
27:     // Use nextNode object property
28:     nextNode (Node(:SourceID) :Node(DestinationID))
```

involves the newly introduced requirement R9 along with R2. The detailed investigation of both requirements revealed that both requirements have the same precondition and event but different postconditions.

C. PERFORMANCE AND USER FEEDBACK

The performance of the RDI tool was evaluated on a Dell Latitude 5580. The translation of a 68-node XML file to OWL took 7 seconds, with defect checks taking under two seconds each. The ease of learning and user satisfaction were also assessed, with practitioners finding the model clear and beneficial for detecting requirements defects.

This evaluation demonstrates the BERDD approach's potential in defect detection. Further studies are planned to validate the approach across a wider range of case studies.

VIII. DISCUSSION AND LIMITATIONS

This section explores the broader implications and constraints of our research, including scalability and contrasts with existing studies.

A. COMPARISON WITH EXISTING STUDIES

Research in requirements defects detection broadly falls into two categories: formal and semi-formal approaches. Formal methods involve translating software requirements into languages like first-order logic or propositional logic for defect detection. Semi-formal approaches utilize representations like UML or BT. Additionally, NLP techniques are also employed for defect detection.

TABLE 6. AIP Functional Requirements.

No	Requirement
R1	The system is turned on when the batteries are put in and is turned off when the batteries are out.
R2	To start the pump, when in the stopped state, the start-stop button is held down until it beeps three times and (...) is displayed on the screen.
R3	Every time the system pumps 1 ml drug, and when the battery is low, it sends a single beep alarm.
R4	After a set time pump activates to pump 1ml of a drug through the line.
R5	When the volume reaches 5ml, the system does three beeps and displays a low volume message every 1ml as it counts down to empty.
R6	When there is no drug left, the pump enters stopped mode, and the system sounds a continuous beeping alarm.
R7	When the line is blocked, or air is detected, the pump is stopped and the beeper a continuous beep.
R8	When the battery is low, the system sends three beeps and displays a low battery message on the screen.
R9	To stop the pump, when in running state, the start-stop button is held down until it beeps three times and (...) is displayed on the screen.

Our approach shares similarities with ontology-based requirements defects detection, such as Nguyen et al. [6] and Nentwich et al. [8], who utilized DL and first-order logic, respectively. In contrast, semi-formal methods predominantly use UML, as seen in the approaches by El-Attar and Miller [74] and Kaneiwa and Satoh [48].

Comparing these with our approach, we observe:

- UML-based studies often emphasize inconsistencies across various UML models, while BE focuses on a minimal set of coherent notations.
- Most of the existing approaches including tools that used formal languages translate software requirements from natural language directly into formal languages; this process requires an expert-level understanding of these languages [13]. Due to the lack of such understanding, the customers who provide the requirements may find it difficult to understand formal languages, while the engineers may not have the domain knowledge to interpret the customers' requirements correctly. That means the customer may not be able to verify if the formal representation is an accurate reflection of their requirements. However, in this work, we first translate natural language requirements into the semi-formal modelling language, i.e. BT and then convert BT into

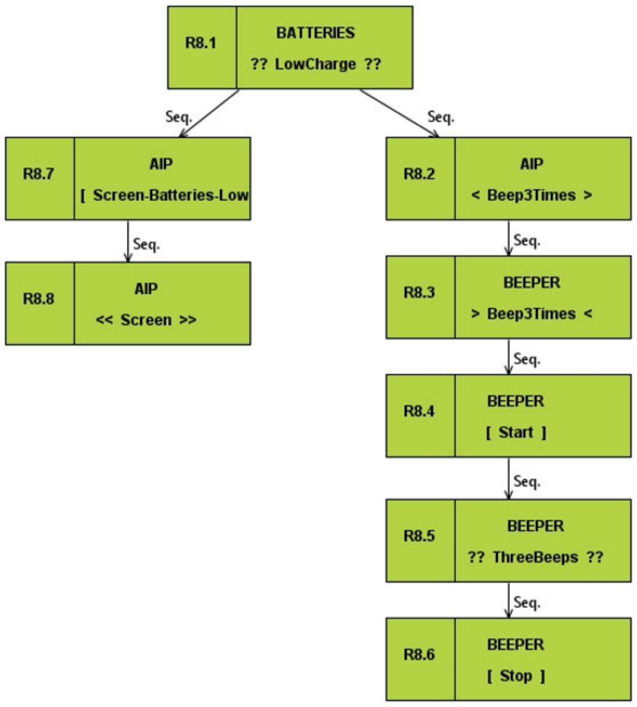


FIGURE 13. AIP-R8 RBT.

formal language, i.e. OWL. Customers and requirements analysts easily understand the semi-formal language used as a bridge, which overcomes the above-discussed shortcomings.

- To perform requirements defects detection effectively, good tool support is needed [75], and it becomes more important while using formal languages. Most of the existing approaches that use formal languages defined requirements defects through natural language descriptions, which might have ambiguous interpretations. Due to this, it seems to be difficult to formalise defects, and as a result, it becomes challenging to develop an automated tool. However, in this work, we first formalise defects through semi-formal language, making it easy to design queries to detect them through an automated tool.
- Unlike many studies that focus solely on inconsistency defects, our work also addresses incompleteness, ambiguity, and redundancy.

B. IS BERDD SCALABLE?

Considering scalability, both Behaviour Tree and SPARQL queries, the core components of our approach, have shown promising results in larger systems. For example, BT has been effectively used in complex systems like satellite control [27] and car-fleet management [76]. SPARQL queries, according to Sejdiu et al. [77], have demonstrated efficient computation even with large data sets. Thus, intuitively, BERDD appears scalable for larger systems.

BERDD is designed for defect detection during requirement analysis but can also be effective in the RCM process.

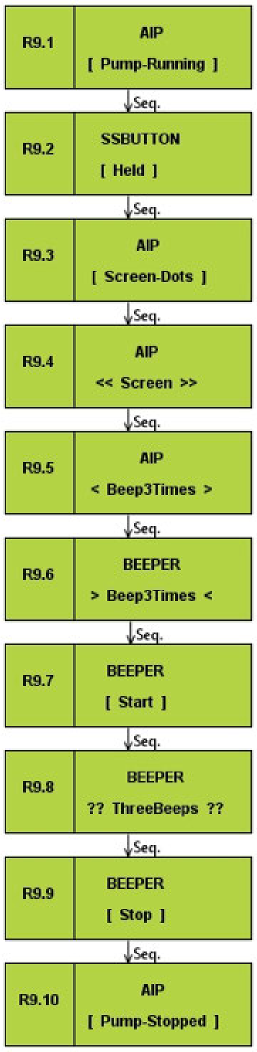
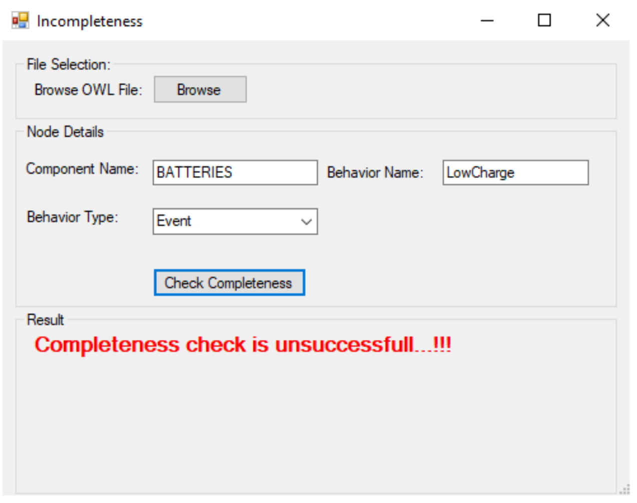


FIGURE 14. AIP-R9 RBT.

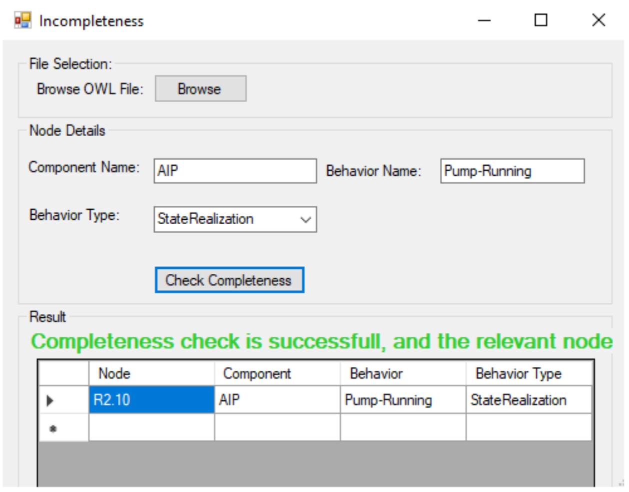


Component Name: BATTERIES
Behavior Name: LowCharge
Behavior Type: Event

Completeness check is unsuccessfull...!!!

FIGURE 15. AIP-R8 incompleteness detection.

It can identify defects in new requirements introduced during RCM, applying equally well in this phase as in requirements elicitation and analysis.


Incompleteness

File Selection:
Browse OWL File: Browse

Node Details
Component Name: AIP
Behavior Name: Pump-Running
Behavior Type: StateRealization

Check Completeness

Result
Completeness check is successfull, and the relevant node

Node	Component	Behavior	Behavior Type
R2.10	AIP	Pump-Running	StateRealization

FIGURE 16. **AIP-R9 incompleteness detection.**

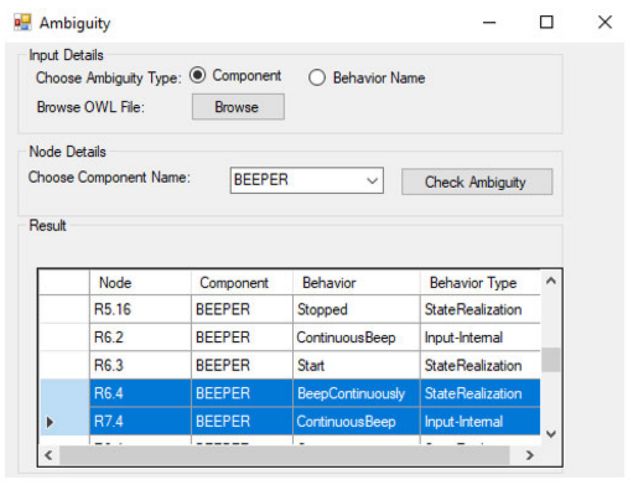
Algorithm 2 SPARQL Query to Check Incompleteness Defect

```
1:  PREFIX rdf: < http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2:  PREFIX owl: < http://www.w3.org/2002/07/owl#>
3:  PREFIX xsd: < http://www.w3.org/2001/XMLSchema#>
4:  PREFIX rdfs: < http://www.w3.org/2000/01/rdf-schema#>
5:  PREFIX my: < http://www.semanticweb.org/s5105389/ontologies/2018/8/untitled-ontology-9#>
6:  SELECT ?cname ?bname ?btype
7:  WHERE {
8:  ?node rdf:type my:BTNodes.
9:  ?node my:hasComponentName ?cname.
10: ?node my:hasComponentName ?bname.
11: ?node my:hasBehaviorType ?btype.
12: FILTER (str (strafter(str(?cname), @prefix)) = @parm1)
13: FILTER (str (strafter(str(?bname), @prefix)) = @parm1)
14: FILTER (str (strafter(str(?btype), @prefix)) = @parm1)}
```

C. LIMITATIONS

Our approach, while promising, is not without limitations:

- Coverage: While addressing the four most common defects, other types of defects remain uncovered.
- Incompleteness: We focus on individual requirements incompleteness and do not address the incompleteness of overall system requirements.
- Ambiguity: Our approach primarily handles one type of ambiguity, leaving out other forms such as unclear statements.
- Inconsistency and Redundancy: Only certain aspects of these defects are covered.
- Ambiguity Detection: This requires domain knowledge, leading to potential subjective interpretations.


Ambiguity

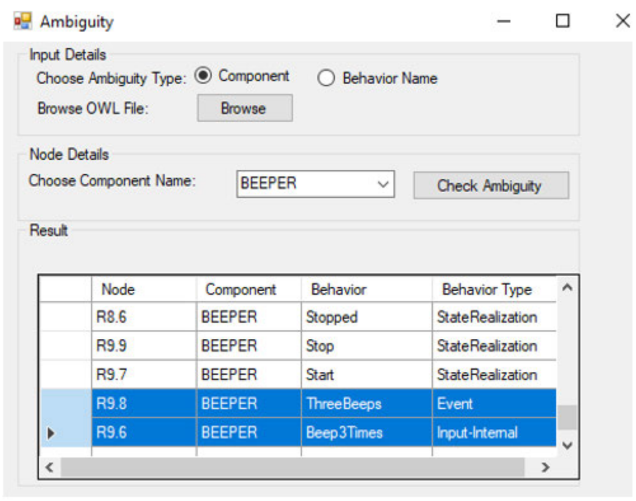
Input Details
Choose Ambiguity Type: [✓ Component] Behavior Name
Browse OWL File: Browse

Node Details
Choose Component Name: BEEPER
Check Ambiguity

Result

Node	Component	Behavior	Behavior Type
R5.16	BEEPER	Stopped	StateRealization
R6.2	BEEPER	ContinuousBeep	Input-Internal
R6.3	BEEPER	Start	StateRealization
R6.4	BEEPER	BeepContinuously	StateRealization
R7.4	BEEPER	ContinuousBeep	Input-Internal

FIGURE 17. **AIP ambiguity detection.**


Ambiguity

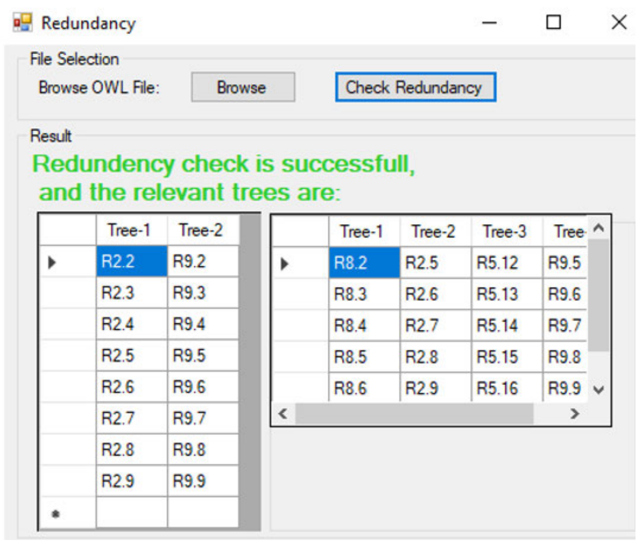
Input Details
Choose Ambiguity Type: [✓ Component] Behavior Name
Browse OWL File: Browse

Node Details
Choose Component Name: BEEPER
Check Ambiguity

Result

Node	Component	Behavior	Behavior Type
R8.6	BEEPER	Stopped	StateRealization
R9.9	BEEPER	Stop	StateRealization
R9.7	BEEPER	Start	StateRealization
R9.8	BEEPER	ThreeBeeps	Event
R9.6	BEEPER	Beep3Times	Input-Internal

FIGURE 18. **AIP ambiguity detection.**


Redundancy

File Selection
Browse OWL File: Browse
Check Redundancy

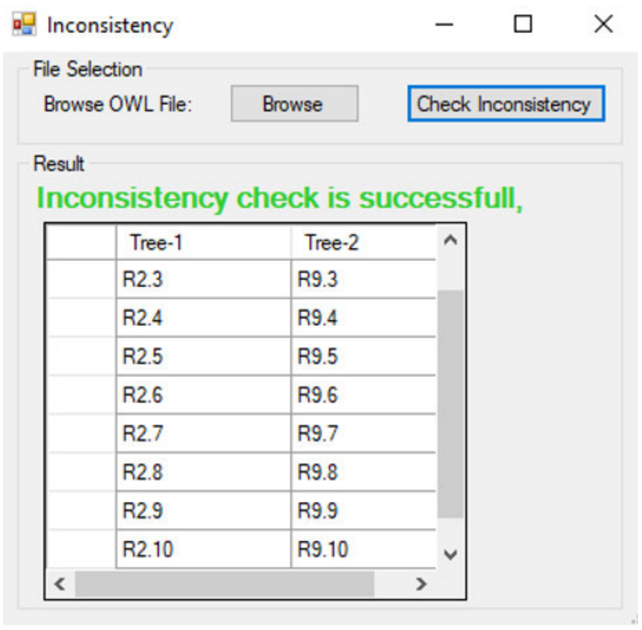
Result
Redundancy check is successfull,
and the relevant trees are:

Tree-1	Tree-2
R2.2	R9.2
R2.3	R9.3
R2.4	R9.4
R2.5	R9.5
R2.6	R9.6
R2.7	R9.7
R2.8	R9.8
R2.9	R9.9

Tree-1	Tree-2	Tree-3	Tree-
R8.2	R2.5	R5.12	R9.5
R8.3	R2.6	R5.13	R9.6
R8.4	R2.7	R5.14	R9.7
R8.5	R2.8	R5.15	R9.8
R8.6	R2.9	R5.16	R9.9

FIGURE 19. **AIP redundancy detection.**

- Case Study Limitation: The approach has been tested on a relatively simple system, necessitating further evaluation on larger case studies.



Inconsistency check is successfull,

	Tree-1	Tree-2
	R2.3	R9.3
	R2.4	R9.4
	R2.5	R9.5
	R2.6	R9.6
	R2.7	R9.7
	R2.8	R9.8
	R2.9	R9.9
	R2.10	R9.10

FIGURE 20. **AIP inconsistency detection.**

- Applicability to Non-Functional Requirements: The normal formed RBT is more suited for functional requirements and may not adequately handle non-functional aspects like cross-platform compatibility.

In conclusion, while BERDD offers a significant advancement in requirements defects detection, further research and more extensive case studies are needed to fully explore its capabilities and limitations.

IX. CONCLUSION & FUTURE WORK

This paper introduces a novel approach for automatically detecting software requirements defects, substantially enhancing coverage compared to existing methodologies. The approach incorporates two key elements:

First, we formalize four major requirements defects (inconsistency, redundancy, incompleteness, and ambiguity) using a normalized behaviour tree. This formalization is grounded in the Inertia Axiom, which we introduce to define a normal formed Requirement Behaviour Tree (RBT). Algorithms based on this formalism are then developed for defect detection.

Second, we develop the Behaviour Engineering-based Requirements Defects Detection (BERDD) approach. In this framework, natural language software requirements are modeled using behaviour trees (BTs), which are then translated into the Web Ontology Language (OWL). This translation process effectively bridges the gap between natural language requirements and their formal representations, facilitating verification.

To validate our approach, we created the Requirements Defects Identifier (RDI) tool. RDI utilizes SPARQL queries to interrogate the OWL representations of requirements and identify defects. When combined with the iRE tool, RDI offers comprehensive support for requirements analysts, requiring no prior knowledge of SPARQL. Our application of this tool to a simple system demonstrates its efficacy in detecting all four types of requirements defects.

Looking ahead, our future work aims to:

- Extend the coverage of this approach by incorporating additional types of requirements defects.
- Empirically validate the proposed approach and developed tool through industry professional surveys.
- Investigate the applicability of this approach in the context of global software development.
- Explore the modeling of requirements defects using composition trees, which represent the static aspects of software-intensive systems.

APPENDIX A OWL FILE

Available at http://www.beworld.org/BERDD/OWL-File.html

APPENDIX B SPARQL QUERIES

Available at http://www.beworld.org/BERDD/SPARQL-Queries.html

APPENDIX C AIP-IBT FILE

Available at http://www.beworld.org/BERDD/AIP-IBT.html

REFERENCES

[1] I. Sommerville, *Software Engineering*, 9th ed. Pearson Education India, 2011.

[2] S. Jayatilleke and R. Lai, "A systematic review of requirements change management," *Inf. Softw. Technol.*, vol. 93, pp. 163–185, Jan. 2018.

[3] B. Bernardez, M. Genero, A. Duran, and M. Toro, "A controlled experiment for evaluating a metric-based reading technique for requirements inspection," in *Proc. 10th Int. Symp. Softw. Metrics*, 2004, pp. 257–268.

[4] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Found. Empirical Softw. Eng., Legacy Victor R Basili*, vol. 426, no. 37, pp. 426–431, 2005.

[5] S. Anwer, L. Wen, and Z. Wang, "A systematic approach for identifying requirement change management challenges: Preliminary results," in *Proc. Eval. Assessment Softw. Eng.*, Apr. 2019, pp. 230–235.

[6] T. H. Nguyen, B. Q. Vo, M. Lumpe, and J. Grundy, "KBRE: A framework for knowledge-based requirements engineering," *Softw. Quality J.*, vol. 22, no. 1, pp. 87–119, Mar. 2014.

[7] A. Goknil, I. Kurtev, K. van den Berg, and J.-W. Veldhuis, "Semantics of trace relations in requirements models for consistency checking and inferencing," *Softw. Syst. Model.*, vol. 10, no. 1, pp. 31–54, Feb. 2011.

[8] C. Nentwich, W. Emmerich, A. Finkelsteiin, and E. Ellmer, "Flexible consistency checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 28–63, Jan. 2003.

[9] S. Banerjee and A. Sarkar, "Domain-specific requirements analysis framework: Ontology-driven approach," *Int. J. Comput. Appl.*, vol. 44, no. 1, pp. 23–47, Jan. 2022.

[10] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, 2012.

[11] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Trans. Softw. Eng. Methodology*, vol. 14, no. 3, pp. 277–330, Jul. 2005.

[12] M. J. Cresswell, *Logics and Languages*. Evanston, IL, USA: Routledge, 2016.

[13] M. Gogolla, "Benefits and problems of formal methods," in *Proc. Int. Conf. Reliable Softw. Technol.* Cham, Switzerland: Springer, 2004, pp. 1–15.

[14] D. Schlimm, "Formal languages in logic. A philosophical and cognitive analysis," *Hist. Philosophy Log.*, vol. 35, no. 1, pp. 108–110, Jan. 2014.

[15] V. Langenfeld, A. Post, and A. Podelski, "Requirements defects over a project lifetime: An empirical analysis of defect data from a 5-year automotive project at Bosch," in *Proc. Int. Work. Conf. Requirements Eng., Found. Softw. Quality*. Cham, Switzerland: Springer, 2016, pp. 145–160.
[16] K. Ahmed, L. Wen, and A. Sattar, "IRE: A semantic network based interactive requirements engineering framework," in *Proc. 2nd World Conf. Complex Syst. (WCCS)*, Nov. 2014, pp. 171–177.
[17] K. Ahmed, T. Myers, L. Wen, and A. Sattar, "Detecting requirements defects utilizing a mathematical framework for behavior engineering," 2014, *arXiv:1401.5198*.
[18] S. Zafar, N. Farooq-Khan, and M. Ahmed, "Requirements simulation for early validation using behavior trees and datalog," *Inf. Softw. Technol.*, vol. 61, pp. 52–70, May 2015.
[19] J. H. Hayes, "Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project," in *Proc. 14th Int. Symp. Softw. Rel. Eng.*, 2003, pp. 49–59.
[20] R. G. Dromey, "Climbing over the 'no silver bullet' brick wall," *IEEE Softw.*, vol. 23, no. 2, pp. 119–120, Mar./Apr. 2006.
[21] L. Wen, D. Tuffley, and T. Rout, "Using composition trees to model and compare software process," in *Int. Conf. Softw. Process Improvement Capability Determination*. Cham, Switzerland: Springer, 2011, pp. 1–15.
[22] R. G. Dromey, "Genetic software engineering-simplifying design using requirements integration," in *Proc. IEEE Work. Conf. Complex Dyn. Syst. Archit.*, Jun. 2001, pp. 251–257.
[23] C. Gonzalez-Perez, B. Henderson-Sellers, and G. Dromey, "A metamodel for the behavior trees modelling technique," in *Proc. 3rd Int. Conf. Inf. Technol. Appl. (ICITA)*, 2005, pp. 35–39.
[24] R. Colvin and I. J. Hayes, "A semantics for behavior trees," Univ. Queensland, St Lucia, QLD, Australia, Tech. Rep. SSE-2010-03, 2010.
[25] L. W. Chan, R. Hexel, and L. Wen, "Integrating non-monotonic reasoning into high level component-based modelling using behavior trees," in *Proc. SoMeT*, 2012, pp. 21–40.
[26] R. Colvin, L. Grunske, and K. Winter, "Probabilistic timed behavior trees," in *Proc. Int. Conf. Integr. Formal Methods*. Cham, Switzerland: Springer, 2007, pp. 156–175.
[27] *Behavior Engineering World*. Accessed: Dec. 1, 2023. [Online]. Available: http://www.beworld.org/BE/
[28] S. Shlaer and S. J. Mellor, *Object Lifecycles: Modeling the World in States*. USA: Yourdon Press, 1992.
[29] R. G. Dromey, "From requirements to design: Formalizing the key steps," in *Proc. 1st Int. Conf. Softw. Eng. Formal Methods*, 2003, pp. 2–11.
[30] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *J. Web Semantics*, vol. 5, no. 2, pp. 51–53, Jun. 2007.
[31] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 reasoner," *J. Automated Reasoning*, vol. 53, no. 3, pp. 245–269, Oct. 2014.
[32] L. Wen and R. G. Dromey, "From requirements change to design change: A formal path," in *Proc. 2nd Int. Conf. Softw. Eng. Formal Methods*, Beijing, China, 2004, pp. 104–113.
[33] L. Giordano, V. Gliozzi, N. Olivetti, and G. L. Pozzato, "Semantic characterization of rational closure: From propositional logic to description logics," *Artif. Intell.*, vol. 226, pp. 1–33, Sep. 2015.
[34] O. Corcho and A. Gmez-Prez, "A roadmap to ontology specification languages," in *Proc. Int. Conf. Knowl. Eng. Knowl. Manag.* Cham, Switzerland: Springer, 2000, pp. 80–96.
[35] D. Tsarkov and I. Horrocks, "FaCT++ description logic reasoner: System description,' in *Automated Reasoning*, Berlin, Germany: Springer, 2006, pp. 292–297.
[36] D. Dermeval, J. Vilela, I. I. Bittencourt, J. Castro, S. Isotani, P. Brito, and A. Silva, "Applications of ontologies in requirements engineering: A systematic review of the literature," *Requirements Eng.*, vol. 21, no. 4, pp. 405–437, Nov. 2016.
[37] I. L. Margarido, J. P. Faria, R. M. Vidal, and M. Vieira, "Classification of defect types in requirements specifications: Literature review, proposal and assessment," in *Proc. 6th Iberian Conf. Inf. Syst. Technol.*, 2011, pp. 1–6.
[38] S. Robertson and J. Robertson, *Mastering the Requirements Process: Getting Requirements Right*. Reading, MA, USA: Addison-Wesley, 2012.
[39] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*. London, U.K.: Pearson, 1998.
[40] H. Yang, A. de Roeck, V. Gervasi, A. Willis, and B. Nuseibeh, "Analysing anaphoric ambiguity in natural language requirements," *Requirements Eng.*, vol. 16, no. 3, pp. 163–189, Sep. 2011.
[41] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," *Inf. Softw. Technol.*, vol. 45, no. 14, pp. 993–1009, 2003.
[42] S. K. Chang, *Handbook of Software Engineering and Knowledge Engineering*. Singapore: World Scientific, 2001.
[43] A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzé, "Intrinsic redundancy for reliability and beyond," in *Present and Ulterior Software Engineering*. Berlin, Germany: Springer, 2017, pp. 153–171.
[44] X. Chen, Z. Zhong, Z. Jin, M. Zhang, T. Li, X. Chen, and T. Zhou, "Automating consistency verification of safety requirements for railway interlocking systems," in *Proc. Int. Requirements Eng. Conf. (RE)*, 2019, pp. 308–318.
[45] A. Artale and E. Franconi, "A temporal description logic for reasoning about actions and plans," *J. Artif. Intell. Res.*, vol. 9, pp. 463–506, Dec. 1998.
[46] J. Chanda, A. Kanjilal, and S. Sengupta, "UML-compiler: A framework for syntactic and semantic verification of UML diagrams," in *Distributed Computing and Internet Technology*. Berlin, Germany: Springer, 2010, pp. 194–205.
[47] A. H. Khan and I. Porres, "Consistency of UML class, object and statechart diagrams using ontology reasoners," *J. Vis. Lang. Comput.*, vol. 26, pp. 42–65, Feb. 2015.
[48] K. Kaneiwa and K. Satoh, "On the complexities of consistency checking for restricted UML class diagrams," *Theor. Comput. Sci.*, vol. 411, no. 2, pp. 301–323, Jan. 2010.
[49] M. Kamalrudin, J. Hosking, and J. Grundy, "MaramaAIC: Tool support for consistency management and validation of requirements," *Automated Softw. Eng.*, vol. 24, no. 1, pp. 1–45, Mar. 2017.
[50] S. Liu, J. Sun, Y. Liu, Y. Zhang, B. Wadhwa, J. S. Dong, and X. Wang, "Automatic early defects detection in use case documents," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2014, pp. 785–790.
[51] P. Kroha, R. Janetzko, and J. E. Labra, "Ontologies in checking for inconsistency of requirements specification," in *Proc. 3rd Int. Conf. Adv. Semantic Process.*, 2009, pp. 32–37.
[52] J. Muskens, R. J. Bril, and M. R. Chaudron, "Generalizing consistency checking between software views," in *Proc. 5th Work. IEEE/IFIP Conf. Softw. Archit.*, Nov. 2005, pp. 169–180.
[53] A. Kossiakoff and W. N. Sweet, *Systems Engineering: Principles and Practices*. Hoboken, NJ, USA: Wiley, 2003.
[54] J. Holt, *UML for Systems Engineering: Watching the Wheels*. Edison, NJ, USA: IET, 2004.
[55] R. G. Dromey, "Formalizing the transition from requirements to design," in *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*. Singapore: World Scientific, 2006, pp. 173–205.
[56] T. Myers, *The Foundation for a Scaleable Methodology for Systems Design*. Australia: Griffith Univ., 2010.
[57] A. Mavin, P. Wilksinson, S. Gregory, and E. Uusitalo, "Listens learned (8 lessons learned applying EARS)," in *Proc. IEEE 24th Int. Requirements Eng. Conf. (RE)*, Sep. 2016, pp. 276–282.
[58] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Automated checking of conformance to requirements templates using natural language processing," *IEEE Trans. Softw. Eng.*, vol. 41, no. 10, pp. 944–968, Oct. 2015.
[59] S. F. Tjong and D. M. Berry, "The design of SREE—A prototype potential ambiguity finder for requirements specifications and lessons learned," in *Proc. Int. Work. Conf. Requirements Eng., Found. Softw. Quality*. Cham, Switzerland: Springer, 2013, pp. 80–95.
[60] H. Hasso, M. Dembach, H. Geppert, and D. Toews, "Detection of defective requirements using rule-based scripts," in *Proc. REFSQ Workshops*, 2019, pp. 1–5.
[61] H. Femmer, D. Méndez Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," *J. Syst. Softw.*, vol. 123, pp. 190–213, Jan. 2017.
[62] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, A. Fantechi, and S. Gnesi, "Detecting requirements defects with NLP patterns: An industrial experience in the railway domain," *Empirical Softw. Eng.*, vol. 23, no. 6, pp. 3684–3733, Dec. 2018.
[63] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E.-V. Chioasca, and R. T. Batista-Navarro, "Natural language processing (NLP) for requirements engineering: A systematic mapping study," 2020, *arXiv:2004.01099*.

[64] F. Dalpiaz, A. Ferrari, X. Franch, and C. Palomares, "Natural language processing for requirements engineering: The best is yet to come," *IEEE Softw.*, vol. 35, no. 5, pp. 115–119, Sep. 2018.
[65] B. Wei, J. Sun, and Y. Wang, "A knowledge engineering approach to UML modeling (S)," in *Proc. SEKE*, 2018, pp. 60–63.
[66] M. Sadowska and Z. Huzar, "Representation of UML class diagrams in OWL 2 on the background of domain ontologies," *E-Inform. Softw. Eng. J.*, vol. 13, no. 1, pp. 63–103, 2019.
[67] K. Siegemund, E. J. Thomas, Y. Zhao, J. Pan, and U. Assmann, "Towards ontology-driven requirements engineering," in *Proc. Workshop Semantic Web Enabled Softw. Eng. 10th Int. Semantic Web Conf. (ISWC)*, 2011, pp. 1–15.
[68] K. Verma and A. Kass, "Requirements analysis tool: A tool for automatically analyzing software requirements documents," in *Proc. Int. Semantic Web Conf.* Cham, Switzerland: Springer, 2008, pp. 751–763.
[69] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 782–791.
[70] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 931–942.
[71] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2015, pp. 43–54.
[72] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 156–166.
[73] *DotNetRef*. Accessed: Dec. 1, 2023. [Online]. Available: https://www.dotnetrdf.org/
[74] M. El-Attar and J. Miller, "Producing robust use case diagrams via reverse engineering of use case descriptions," *Softw. Syst. Model.*, vol. 7, no. 1, pp. 67–83, Nov. 2007.
[75] Y. Xie, T. Tang, T. Xu, and L. Zhao, "Research on requirement management for complex systems," in *Proc. 2nd Int. Conf. Comput. Eng. Technol.*, vol. 1, Apr. 2010, pp. 113–116.
[76] D. Powell, "Behavior engineering—A scalable modeling and analysis method," in *Proc. 8th IEEE Int. Conf. Softw. Eng. Formal Methods*, Sep. 2010, pp. 31–40.
[77] G. Sejdiu, D. Graux, I. Khan, I. Lytra, H. Jabeen, and J. Lehmann, "Towards a scalable semantic-based distributed approach for SPARQL query evaluation," in *Semantic Systems. The Power of AI and Knowledge Graphs*. Cham, Switzerland: Springer, 2019, pp. 295–309.

SAJID ANWER received the bachelor's degree in computer science from the COMSATS Institute of Information Technology, Lahore, Pakistan, the master's degree in computer science from the King Fahd University of Petroleum and Minerals, and the Ph.D. degree from the School of ICT, Griffith University. He is an Assistant Professor with the Department of Software Engineering, FAST-National University of Computer and Emerging Sciences, Chiniot-Faisalabad Campus. Prior to joining Griffith University, he was a Lecturer with the King Fahd University of Petroleum and Minerals, Saudi Arabia. His research interests include global software engineering, software requirements engineering, behavior engineering, and software security.

LIAN WEN (LARRY) received the bachelor's degree in mathematics from Peking University, the master's degree in EE from the Chinese Academy of Space Technology, and the Ph.D. degree in software engineering from Griffith University. He is a Lecturer with the School of ICT, Griffith University. He was a software engineering and a project manager with different IT companies. His current research interests include behavior engineering, complex systems, scale-free networks, software change and evolution, logic programming, answer set programing.

MAHMOOD UL HASSAN received the B.S. degree (Hons.) in computer science from Hazara University, Pakistan, the M.S. degree in computer science from COMSATS University, Pakistan, and the Ph.D. degree in computer science from the IIC University of Technology, Cambodia. He is currently an Assistant Professor with the Department of Computer Skills, Deanship of Preparatory Year, Najran University, Najran, Saudi Arabia. He has more than 40 publications in the area of computer networks, image processing, and cloud computing, in well reputed international journals and conferences. His research interests include ad hoc networks, connectivity and coverage restoration in the wireless networks, image processing, and cloud computing.

ZHE WANG (JACK) is a Senior Lecturer with Griffith University, Australia. His research has led to over 40 publications in influential journals and conferences. His research interests include semantic technologies, artificial intelligence, and ontology-based systems.

AMIN A. AL-AWADY received the B.S., M.S., and Ph.D. degrees in computer science from the Faculty of Computer Science and Technology, Technical University of Wroclaw, Poland. He is currently an Assistant Professor with the Department of Computer Skills, Deanship of Preparatory Year, Najran University, Najran, Saudi Arabia. He has more than 15 publications in the area of database systems, computer networks, and cloud computing, in well reputed international journals and conferences. His research interests include database and distributed database systems, ad hoc networks, connectivity and coverage restoration in the wireless networks, and cloud computing.

YAHYA ALI ABDELRAHMAN ALI received the bachelor's degree in computer science from the University of Science and Technology, the master's degree in computer science from University Technology Malaysia (UTM), in 2007, and the Ph.D. degree in computer science (with excellent academic achievements) from the Sudan University of Science and Technology. He was the Head of the Programming Department, Computer Center, Sudan University of Science and Technology. Currently, he is an Assistant Professor with the College of Computer Science and Information System, Najran University (NU), Saudi Arabia. His research interests include natural language processing (NLP), cloud computing, and image processing.

• • •