Int. j. inf. tecnol. (January 2024) 16(1):159–168
https://doi.org/10.1007/s41870-023-01471-9

ORIGINAL RESEARCH

BECIA: a behaviour engineering-based approach for change impact analysis

Sajid Anwer[1,3] · Lian Wen[1,2] · Shaoyang Zhang[4] · Zhe Wang[1,2] · Yong Sun[5]

Received: 19 April 2023 / Accepted: 28 August 2023 / Published online: 15 September 2023
© The Author(s) 2023

Abstract This paper introduces Behaviour Engineering-based Change Impact Analysis (BECIA), a novel approach to Change Impact Analysis (CIA). BECIA enables visualization of change impacts from modified requirements on architecture design. It also introduces the Change Impact Indicator (CII) metric, quantifying impact by calculating the Kolmogorov Complexity (KC) of design models. Our contributions include: an algorithm to convert behavior trees to composition trees, a requirements components dependency network, and a metric for quantifying change impacts.

Keywords Requirements change · Change impact analysis · Changeability · Behaviour trees · Composition trees · Kolmogorov complexity

✉ Lian Wen
l.wen@griffith.edu.au

Sajid Anwer
Sajidanwer786@gmail.com

Shaoyang Zhang
zhsy@chd.edu.cn

Zhe Wang
zhe.wang@griffith.edu.au

Yong Sun
sunyong@chd.edu.cn

1 School of Information and Communication Technology, Griffith University, Brisbane, Australia

2 Institute for Integrated and Intelligent Systems (IIIS), Griffith University, Brisbane, Australia

3 National University of Computer and Emerging Sciences, Islamabad, Pakistan

4 School of Information Engineering, Chang'an University, Xi'an, China

5 Information and Network Management, Chang'an University, Xi'an, China

1 Introduction

Software requirements frequently evolve during the development life cycle, with up to 50% changing before system deployment [1]. These changes can be driven by stakeholder needs, business goals, or technological advancements [2]. While these new requirements may only occupy a small portion of the system, they can significantly impact other requirements and critical design artifacts, including the architecture. Identifying these change impacts can be challenging due to the system's size and complexity [3].

Studies show that 85–90% of software system budgets are spent on operation and maintenance [4], underlining the need for systematic and automated approaches to identifying change impacts for cost-effective software development [5–7]. This has given rise to Change Impact Analysis (CIA), which various techniques have addressed [8, 9], including on requirements [10], architecture [11], source code [2], and a combination of them [12].

However, existing approaches often only provide a rough set of potentially impacted elements and use natural languages, making them difficult to process with automation tools [13]. Additionally, they do not usually provide measures to quantify change impacts, making cost estimation challenging [14, 15].

To tackle these issues, we propose a Behaviour Engineering-based Change Impact Analysis (BECIA) that addresses these limitations by harnessing the advanced features of Behaviour Engineering (BE) [16]. We introduce a new model, the Requirements Components Dependency Network (RCDN), and algorithms to transform Integrated Behaviour Trees into Integrated Composition Trees and then to RCDNs. These can be automated to improve efficiency and reduce errors. We also present a Change Impact Indicator (CII) to quantify change impacts, using the concept

of Kolmogorov Complexity [17] to estimate a component's complexity. We evaluate our approach with final year projects of undergraduate students.

The remainder of this paper presents BECIA's key elements (Sect. 2), a workflow example and evaluation (Sects. 3 and 4), and a comparison with existing approaches (Sect. 5), followed by our conclusion and future work (Sect. 6)."

2 Key elements of BECIA

This section introduces BECIA's four key elements: the conversion of an Integrated Behaviour Tree (IBT) to an Integrated Composition Tree (ICT), the transformation of an ICT to a Requirements Components Dependency Network (RCDN), the transition from an IBT to Component Impact Diagrams (CIDs), and the calculation of the Change Impact Indicator (CII).

2.1 Conversion of an IBT into an ICT

The original Behaviour Engineering (BE) methodology translates Behaviour Trees (BTs) and Composition Trees (CTs) directly from requirements. This process helps identify requirement defects and terminological ambiguities, which proves useful when requirements are poorly written, or the design team lacks familiarity with them.

In this research, we propose an algorithm for automating the conversion of an IBT into an ICT. This automation has three benefits: it ensures the completeness and correctness of the IBT, saves human effort during change management, and reduces human errors, guaranteeing consistency between the diagrams.

We illustrate the conversion process using a simple abstract example of a system with four requirements, as shown in Fig. 1. The IBT for this system includes five components (C1 to C5) and eight behaviours (B1 to B8). All three-leaf nodes include a reversion sign ("^"), indicating a return to the closest parent node of the same behaviour.

Before constructing the ICT, we identify the system component serving as the root node. The system component name is denoted within a double-lined rectangle. Processing each behaviour node in the IBT involves:

1. Checking the component name and creating a component node in the ICT if it's not already there. A simple ICT includes one system component, with all other components as direct children. More complex systems may have more intricate composition trees.
2. Checking the node's behaviour and requirement link. This step involves three possible scenarios:

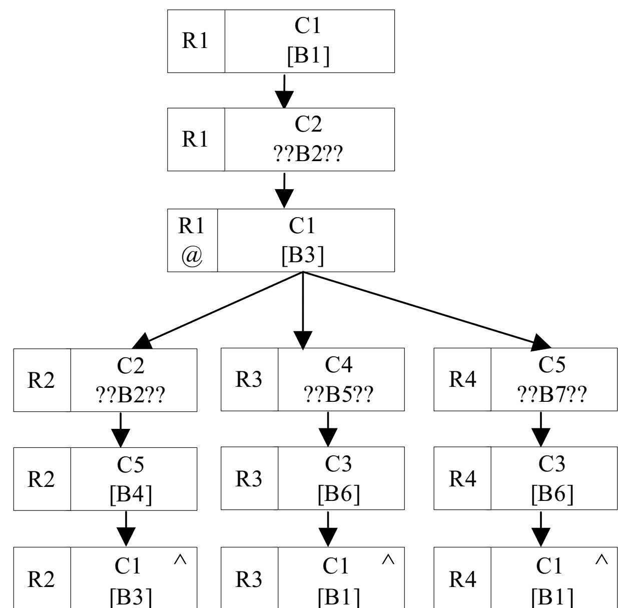

Fig. 1 A simple example IBT

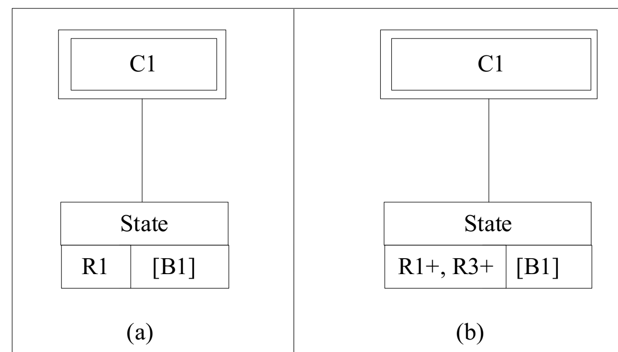

Fig. 2 Step to model example ICT

a. If the ICT doesn't have a node with the same behaviour under the component, a new node is created (Fig. 2a).
b. If the ICT has a node with the same behaviour under the component, but the requirement link is not in the node's CT, the new requirement link from the BT node is added (Fig. 2b).
c. If the ICT has a node with the same behaviour and the requirement link is already in the CT's requirement link cell, no changes are made (Fig. 3).

An important point in step 2 is the use of a plus sign " + " with the requirement ID if the Requirement Behaviour Tree (RBT) changes a component's state. For instance, in the ICT node for C1 [B3] (Fig. 3), the requirement link

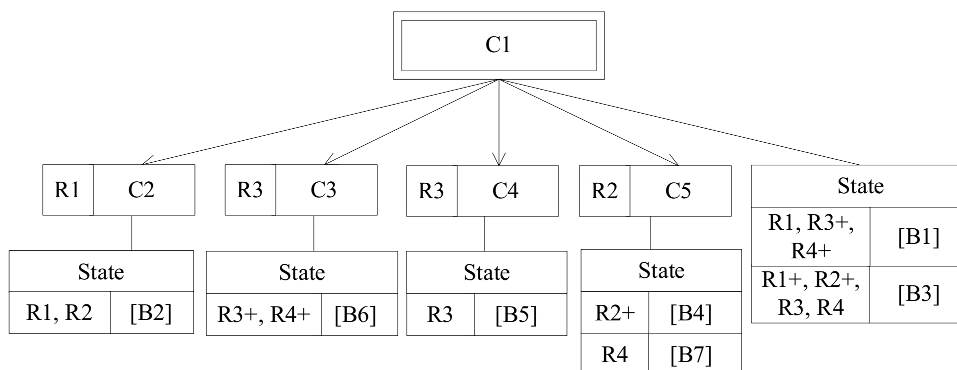

Fig. 3 Example ICT

for R1 has a plus sign as R1 changes the state of C1 from B1 to B3.

After following steps 1 and 2, we generate the ICT for all four requirements (Fig. 3). Notably, the IBT's integration node shows both requirement links in the ICT, as this node is a postcondition of one requirement and a precondition of the other (as shown with the "@" sign in Fig. 1, integrating R1 with R2, R3, and R4). Therefore, the behaviour B3 of component C1 is associated with requirements links R1, R2, R3, and R4 in the ICT (Fig. 3).

2.2 Conversion of an ICT into RCDN

Individual functional requirements, as initially presented, do not highlight their interdependent relationships. Therefore, a thorough requirements analysis is necessary to identify and visually represent these relationships.

Traditional techniques manually identify these relationships, usually illustrating the requirements relationship in the problem domain. For instance, in a use case diagram, two related use cases like "create an order" and "pay an order" are depicted. However, since these diagrams are constructed prior to considering implementation, they often fail to reflect the relationship between two requirements in the solution domain.

Our new model captures dependency relationships among requirements based on their associations with components. The logic is straightforward: if two requirements involve the same component, they are closely related; conversely, if two requirements don't share components, they are relatively independent. To achieve this, we introduce a new Requirements Engineering (RE) model called Requirements Components Dependency Network (RCDN), which connects requirements through the components involved in their implementation. Hypergraph models have shown great effectiveness in capturing higher-order relationships [18].

An RCDN is a hypergraph $H = (V, E)$, where V is the set of requirements (vertices) and E is the set of components (hyperedges). Requirements $v_1, \ldots, v_n$ are connected by a component e if all these requirements involve e. We use hypergraphs as one component (hyperedge) can connect more than two requirements (vertices), effectively capturing higher-order relationships.

To illustrate the ICT-to-RCDN conversion algorithm, we return to the previous subsection's example, with its ICT depicted in Fig. 3. The algorithm starts with the root component and travels through all components. Once a component is selected, the process follows these steps:

1. Check the component name. If the component is not yet in the RCDN, a new component is created (represented as a circle).
2. Check the requirement links under the state section of the ICT node for that component. This check presents three potential situations:
   a. If the RCDN does not have that requirement, a new requirement is created (represented as a rectangle). If there is a '+' sign next to the requirement, a strong connection (solid line) between the requirement and component is added. Otherwise, a weak connection (dotted line) is added. For example, in component C1, the first requirement is R1, so we create a new requirement R1 and connect it with a strong line to component C1 (Fig. 4a).

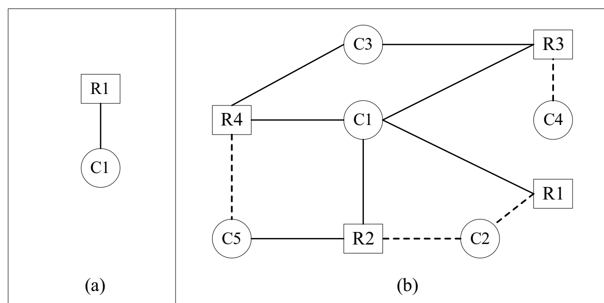

Fig. 4 The RCDN for the Example ICT

b. If the RCDN already has the requirement, it is only connected to the component based on the connection type.
c. If the RCDN has the requirement, and it's connected to that component with a weak connection, but another instance of the same requirement with different behavior requires a strong connection, the weak connection is replaced with a strong one.

After traversing all nodes in the ICT, we achieve a complete RCDN, as shown in Fig. 4b.

2.3 Transformation of an IBT into CID

In this subsection, we delve into another fundamental component of BECIA: the conversion of an IBT into a CID. While the specifics of the algorithm are already documented [19], we will provide a CID for reference here.

In our work, a CID is used to evaluate whether a component has undergone any changes and, if applicable, to identify the nature of these changes. We continue with the example introduced earlier. Figure 5 displays the CID of component C1, derived from the IBT shown in Fig. 1. This CID effectively maps any changes in the component's structure or behavior.

2.4 Calculation of change impact indicator

One of the features of BECIA is not just the identification of which components and requirements will be influenced by a proposed change, but also quantifying the change impact through a newly introduced Change Impact Indicator (CII).

Changes, according to requirements change taxonomies [20–22], can be classified into three categories: adding new requirements, deleting an existing requirement, and modifying an existing requirement. In BECIA, we first identify a set of components from the RCDN that may be affected due to the proposed change. Then we use CIDs to quantify the actual impact on each component. The set of impacted components is defined as follows:

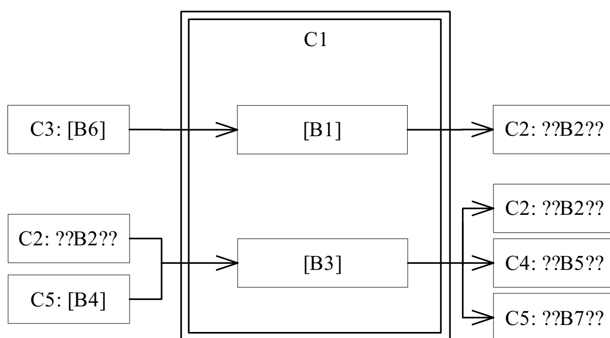

Fig. 5 The CID for Component C1

$$\Theta = \Theta_n \cup \Theta_m \cup \Theta_d, \tag{1}$$

where Θ represents the set of impacted components,, Θ_n is the set of new components, Θ_m is the set of modified components, and Θ_d is the set of deleted components.

To quantify the change impact on a component, we propose a method to estimate the description complexity of the component. Following this, the change impact on the entire system is defined as the sum of the change impacts on all components.

From a CID, we see that each component contains a number of states (or interfaces), and each state is linked to a number of incoming and outgoing components. As shown in Fig. 5, component C1 has two states [B1] and [B3]. State [B1] has one incoming and one outgoing component, whereas state [B3] has two incoming components and three outgoing components.

Kolmogorov Complexity (KC) [17] has been widely studied and accepted as a universally applicable complexity measure in mathematics. It has found various applications in software engineering, often with positive results [23].

In our approach, we employ KC to measure the complexity of individual impacted components, using it as a change impact indicator for the component. We then estimate the overall change impact by aggregating the change impacts on all impacted components. The general formula to estimate an upper boundary of KC of an integer n is as follows:

$$K(n) \leq \log^{*} n + c \tag{2}$$

We simplify the computation for practical purposes and use $\log(n+1)$ to estimate the complexity of non-negative integer value n.

$$K(n) \leq \log(n+1) + c \tag{3}$$

In a CID, each component comprises a number of states; therefore, the complexity of a component can be estimated as the sum of all states' complexities. Let Let S be a state in component C,, n_c is the number of states in C. The complexity of state S can be estimated as:

$$K(S) \leq \log\left(n_c + 1\right) \log\left(n_{s_i} + n_{s_o} + 1\right) \tag{4}$$

We've ignored the constant c when calculating the complexity of a state, as the value is only significant when compared to others, and the constant c could be omitted as it appears in all compared items.

After estimating the complexity of individual states, the complexity of a component can be calculated as:

$$K(C) = \sum_{S\in\Phi} K(S) \tag{5}$$

where $K(C)$ is the Complexity of component C and Φ is the set of states of component C.

The individual component's complexity is then used to calculate the impact of the proposed change on system components. We use δ_C to denote the change impact factor on component C. For a new component, we define the change impact factor as the component's KC, calculated in Eq. (5).

$$\delta_C = K(C) \tag{6}$$

For a modified component, its states are classified into three different sets: newly introduced states, modified states (some of which may have the number of incoming and outgoing components altered), and deleted states:

$$\Phi = \Phi_n \cup \Phi_m \cup \Phi_d \tag{7}$$

where Φ is the set of all states in component C, which equals the union of newly introduced states Φ_n, the set of modified states Φ_m, and the set of deleted states Φ_d.

$$\delta_C = \sum_{S\in\Phi_n} K(S) + \sum_{S\in\Phi_m} \max\left(K\left(S'\right) - K(S), 0\right) \tag{8}$$

where δ_C is the complexity of a modified component, $K(S)$ is the Complexity of the new state and also Complexity before the change, $K\left(S'\right)$ is the complexity of a modified state after the change. We use max to avoid negative values. If complexity is equal to a negative value, we consider it as zero.

Finally, the overall impact of the proposed change can be quantified as follows:

$$\Delta = \sum_{C\epsilon\Theta} \delta_C \tag{9}$$

where Δ is the overall change impact factor, and Θ is the set of impacted components defined in Eq. (1).

3 BECIA workflow with an example

The previous section detailed all the key elements in BECIA. This section will first explain the BECIA workflow by combining all these elements and then provide an example to illustrate the process.

3.1 BECIA workflow

The workflow of BECIA is shown in Fig. 6 and comprises seven steps. Step 1 falls within the problem domain, steps 2–4 are in the analysis domain, and steps 5–7 lie in the solution domain of the software development life cycle.

The initial three steps are standard practices in the BE approach that have been introduced in the background section and detailed in previous papers [19, 24]. The last four steps have been explained in the preceding section. Now, we will illustrate these steps with an example.

3.2 Impact analysis example

This section extends the simple example introduced in Sect. 3 to demonstrate the BECIA workflow. We will reuse the IBT example shown in Fig. 1 and then add a new RBT based on an assumed new requirement. Afterwards, we will carry out steps 3–7 of BECIA based on the IBT and RBT.

3.2.1 Step 2,3-Draw/Modify RBTs and Integrate them with the IBT

Based on the assumed new requirement, we have created a new RBT(R5) in step 2, as illustrated in Fig. 7. Then, in step 3, we merge this RBT with the IBT from Fig. 1. The resulting integration is shown in Fig. 8. The two trees are united through the root node, which is drawn with a thick border. To highlight the changes, we use a grey background color to depict new nodes, and a light grey color with a pattern fill to illustrate modified nodes.

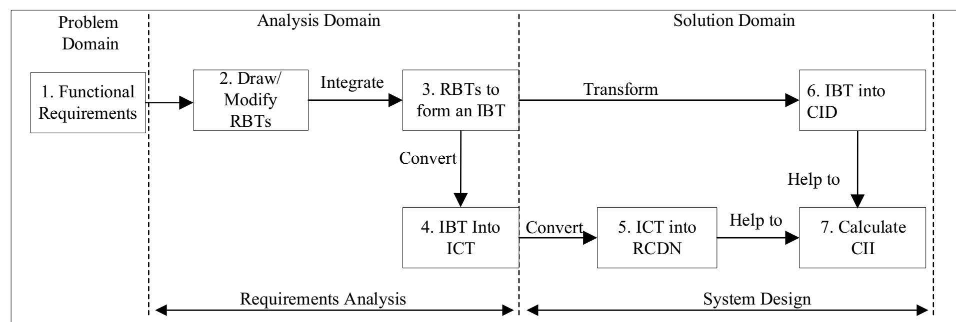

Fig. 6 BECIA Workflow

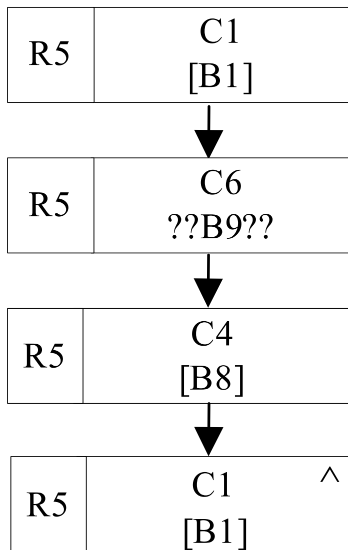

Fig. 7 R5 RBT

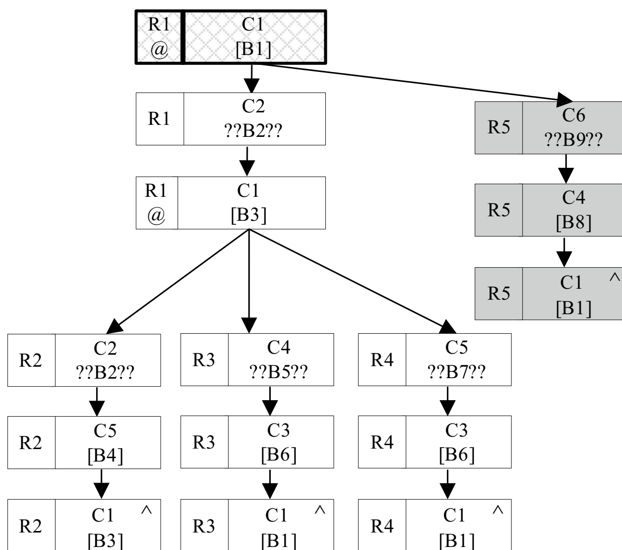

Fig. 8 Updated IBT

3.2.2 *Step 4 convert an IBT to an ICT*

During step 4, we illustrate the process to convert the updated IBT into an ICT. Since the original IBT shown in Fig. 1 already has an associated ICT displayed in Fig. 3, we only need to process the updated section in the IBT to modify the associated ICT accordingly.

We read each node one by one from the updated IBT and incorporate it into the existing ICT based on the two steps defined in Sect. 2.1. For example, we first read the root node of R5, which is an integration node with R1, so we update the state section of the C1. The node with behavior B1 already exists, so we add a new requirement ID in the left cell, make the node boundary bold, and fill it with a cross pattern to highlight this node as a changed node. Conversely, if no node exists for the changed behavior, we model a new node with a grey background, as shown for one node of component C4 in Fig. 9. If any new component appears in the changed requirement, we then create a node for the new component and connect it to the root of the tree, as we did for component C6.

3.2.3 *Step 5 Convert an ICT to an RCDN*

During step 5, we update the RCDN based on the changes in the ICT. This is achieved by following the same process discussed in Sect. 2.2. Given that this change is to add a new requirement (R5), we first model it in Fig. 10 in a rectangle filled with a grey colour. After that, we add a new component C6, which is represented in a circle also filled with grey colour, and then we connect it with R5.

Subsequently, we search for updated components and discover one new node in component C4. This node changes the component state, so a strong connection is formed between R5 and C4. We represent changed components/requirements with a bold boundary filled with a cross pattern, as depicted in Fig. 10. Lastly, we also find one new requirement ID in the existing state of component C1, so we connect requirement R5 to component C1. This completes the conversion of the updated ICT into the RCDN.

3.2.4 *Steps 6,7-Project out CIDs from an IBT and calculate change impact*

Step 6 involves projecting out Component Interaction Diagrams (CIDs) from the updated Interacting Behavior Tree (IBT). We only need to project CIDs for the components potentially impacted by the change. To identify these components, we refer to the Requirement-Component Dependency Network (RCDN), which highlights all newly introduced and modified components.

As per Fig. 10, a new component C6 and two modified components (C1 and C4) have been identified. Thus, we project out the CIDs of these three components from the IBT presented in Fig. 8. The CID for C4 (illustrated in Fig. 11) reveals a newly added state in component C4, with one outgoing interface also altered. The CID of C1 (shown in Fig. 12) reveals multiple interface updates resulting from this change.

Next, we examine the possible indirect impacts based on the relationships between a changed requirement and other requirements. Since R3 and R5 are competing requirements, we need to check R3. Given that R3 hasn't changed, there's no need to assess components connected to R3.

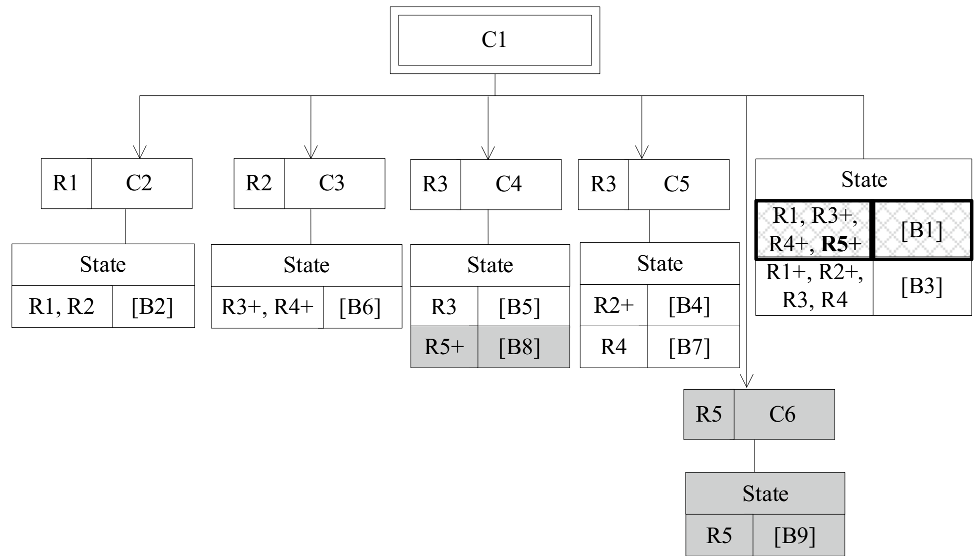

Fig. 9 Updated ICT

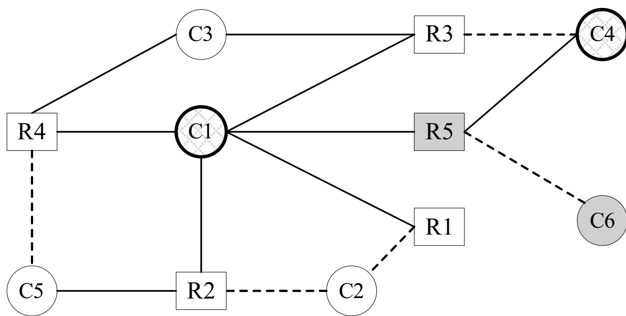

Fig. 10 Updated RCDN

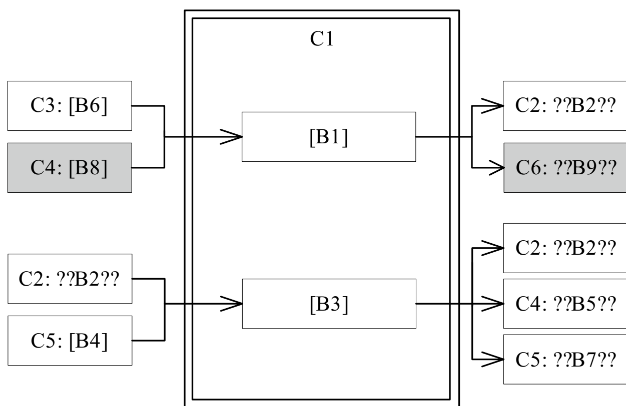

Fig. 11 The CID for C1

Having identified the actually impacted components, we now measure the complexity of these components. Using the CID of these components and Eq. (5), we calculate the complexity of each state in a new component. The CID of

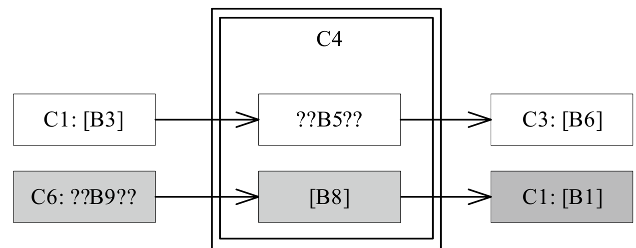

Fig. 12 The CID for C4

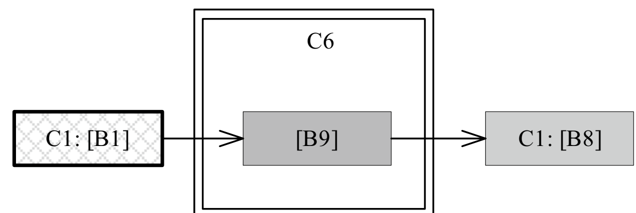

Fig. 13 The CID for Component C6

C6, shown in Fig. 13, allows us to compute the complexity of its state 'B9':

$$K(\mathrm{B9}) \leq \log(1+1)\log(1+1+1) = 1 \times 1.58 = 1.58$$

As there is only one state in C6, the state complexity equals the component complexity. Then, using Eq. (6), we find the change impact of C6 to be:

$$\delta_{C6} = 1.58$$

Next, we calculate the complexity of the modified component. We start with C4 (CID in Fig. 12), which has two

states: one unchanged state (B5) and one new state (B8). For the unchanged state, the complexity is zero, and for the new state, the complexity is:

$$K(\text{B8}) \leq \log(2+1)\log(1+1+1) = 1.58\text{x}1.58 = 2.51$$

After measuring the complexity of all the states of C4, we can calculate the change impact for the modified component (C4) using Eq. (8):

$$\delta_{C4} = (0 + 2.51) = 2.51$$

Similarly, for component C1 (CID in Fig. 11), the unchanged state (B3) has a complexity of zero, and the modified state (B1) has a complexity of:

$$K(\text{B1}) \leq \log(2+1)\log(2+2+1) = 1.58\text{x}2.32 = 3.68$$

So, the change impact for C1 can be calculated as follows:

$$\delta_{C1} = (3.68 + 0) = 3.68$$

After measuring the change impact of the new and modified components, using Eq. (9), we can calculate the change impact indicator (Δ):

$$\Delta = 1.58 + 2.51 + 3.68 = 7.77$$

This complexity value is on a logarithmic scale in the bits unit. The Change Impact Indicator (CII) unit is a bit, which quantifies how many bits of information are needed to describe the change. This absolute value can be meaningless without comparison. The comparison provides an understanding of the magnitude of the change impact on the software.

4 Evaluation

In this section, we evaluate the effectiveness of our BECIA approach using final year projects from undergraduate students at the FAST-National University of Computer and Emerging Sciences (NUCES), Chiniot-Faisalabad campus. FAST-NUCES is one of the top computer science universities in Pakistan [25].

We engaged three groups consisting of nine practitioners in applying BECIA, asking them to critically evaluate the proposed approach against "ease of learning" and "user satisfaction" metrics.

- Ease of Learning: All three groups rated the model as clear and easy to understand. They felt the division of the proposed approach into self-explanatory steps assisted them in their understanding. This is encouraging as it indicates that our model is both concise and comprehensive.
- User Satisfaction: All practitioners agreed that, in general, our model would be useful within the software industry. They believed that the Change Impact Analysis (CIA) approach is clear and can effectively estimate the proposed change's impact on other system components.

Based on this preliminary evaluation, we are confident that the proposed approach can assist in estimating the impact of a proposed change on other system components. However, we also recognize the necessity for further evaluation of the proposed approach through additional case studies. This would help ensure that our model can be effectively and reliably applied in different contexts and scales of software development.

5 Related work and comparison

Much of the existing research focuses on executing Change Impact Analysis (CIA) in source code relative to other Software Development Life Cycle (SDLC) phases. According to Kretsou [26], 62% of the current research concentrates on performing CIA in source code, followed by 22% in the design phase, 14% in architecture, and a mere 2% in requirements. In the context of CIA across various development phases, our approach introduces a method to comprehend change impact, starting from requirements to design and architecture. This covers phases of the SDLC not extensively explored before.

Several studies trace changes from one software artifact to other artifacts. For instance, from source code to software design, Hammad et al. [27] demonstrated a methodology that tracks design evolution based on source code changes.

Considering change propagation from requirements to design, Al-Saiyd and Zriqat [28], Sudin and Kristensen [29], and Kchaou et al. [30] presented various traceability-based approaches.

In terms of techniques employed for CIA, according to Kilpinen [31], CIA research can be divided into two groups: traceability-based and non-traceability or dependency matrix-based methods. Some studies also employ a combination of these approaches.

While Spilkerman [32] and Goknil et al. [33] proposed formal semantics of requirements relations [34] for understanding change impact under traceability methods, Hassine et al. [35] and Ali and Lai [35] presented non-traceability approaches.

Analyzing the above-discussed approaches for CIA, several conclusions can be drawn:

- While various techniques such as dependency graph [30], dynamic slicing [36], distance measure [37], and reverse

Table 1 The comparisons between BECIA and existing CIA approaches

Aspect	Existing CIA approaches	BECIA
Source code	Majority (62%)	N/A
Design	Some (22%)	Focus
Architecture	Some (16%)	Focus
Requirements analysis	Few (2%)	Focus
Time & Cost	High	Low
Automation	Usually difficult	Easy
Quantify impact	Usually not	Yes

engineering [38] have been used for change propagation across different software artifacts, traceability-based approaches have inherent issues such as time and cost required to establish traceability links [39]. and the challenge in managing excessive links between artifacts automatically through software [40].

- Our approach offers traceability benefits due to the inherent properties of the modeling language, BT, used to model system requirements [41]. We exploit these properties and propose two algorithms to systematically transform system requirements from one software artifact to another.
- Regarding quantifying change impact, except a few in source code traceability analysis [2, 35, 42], most existing studies execute CIA without quantifying the change impact. Our approach, in contrast, estimates change impact at the architectural level, identifies impacted components, and estimates the complexity of these components.
- Our approach is fully automatic except for the first step of translating functional requirements into RBTs. All other steps are based on well-defined rules and processes, which provide visible and verifiable traceability across all different design artifacts.

The comparisons between BECIA and existing CIA approaches are summarised in Table 1.

6 Conclusion and future work

In this paper, we have introduced a novel behavior engineering-based approach to conduct change impact analysis, named BECIA. BECIA utilizes an IBT to model system requirements and offers algorithms to convert an IBT into an ICT, and subsequently, the ICT into an RCDN. The RCDN assists in identifying a preliminary set of architectural elements potentially impacted by a proposed change. We then use the CID derived from the IBT and RCDN to investigate which components are genuinely impacted and require modification due to the proposed change. Lastly, we introduce a CII metric to quantify change impact, providing more objective evidence to estimate the development cost for a suggested change.

We believe our approach will aid practitioners in understanding the change impact of functional requirements on design and architecture and establish traceability among different types of software artifacts.

However, the proposed CIA method does have its limitations: firstly, it leverages BE models to estimate change impact, making it inapplicable to other modeling languages. Secondly, it only encapsulates requirements that can be described as "behaviors" and may not be adept at handling certain non-functional requirements such as constraints.

For future work, we intend to create a prototyping tool to augment the proposed approach further and assist in automating it. We also plan to conduct a survey with industry practitioners to empirically validate the proposed approach. The applicability of the proposed framework in the global software development paradigm is another avenue worthy of future exploration.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

Data availability Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Bhatti MW, Hayat F, Ehsan N, Ishaque A, Ahmed S, Sarwar SZ (2010) An investigation of changing requirements with respect to development phases of a software project. In: International Conference on Computer Information Systems and Industrial Management Applications (CISIM), 2010: IEEE, pp 323–327.
2. Jayatilleke S, Lai R, Reed K (2018) A method of requirements change analysis. Requir Eng 23(4):493–508
3. Anwer S, Wen L, Wang Z (2019) A systematic approach for identifying requirement change management challenges: preliminary results. In: Proceedings of the Evaluation and Assessment on Software Engineering, 2019, pp 230–235
4. Erlikh L (2000) Leveraging legacy system dollars for e-business. IT Professional 2(3):17–23
5. Arnold RS, Bohner SA (1993) Impact analysis-towards a framework for comparison. In: Conference on Software Maintenance, 1993, IEEE, pp 292–301

6. Arif M, Mohammad CW, Sadiq M (2023) UML and NFR-framework based method for the analysis of the requirements of an information system. Int J Inf Technol 15(1):411–422
7. Khan T, Faisal M (2023) An efficient Bayesian network model (BNM) for software risk prediction in design phase development. Int J Inf Technol 15(4):2147–2160
8. Ibrahim S, Idris NB, Munro M, Deraman A (2005) A requirements traceability to support change impact analysis. Asian J Inform Tech 4(4):345–355
9. Khurshid S, Shrivastava AK, Iqbal J (2021) Effort based software reliability model with fault reduction factor, change point and imperfect debugging. Int J Inf Technol 13(1):331–340
10. Zhang H et al (2014) Investigating dependencies in software requirements for change propagation analysis. Inf Softw Technol 56(1):40–53
11. Rostami K, Heinrich R, Busch A, Reussner R (2017) Architecture-based change impact analysis in information systems and business processes. In: IEEE International Conference on Software Architecture (ICSA), 2017: IEEE, pp 179–188
12. Yazdanshenas AR, Moonen L (2012) Fine-grained change impact analysis for component-based product families. In: 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp 119–128
13. Goknil A, Kurtev I, Berg Kvd (2016) A rule-based change impact analysis approach in software architecture for requirements changes. arXiv preprint arXiv:1608.02757
14. Angerer F, Grimmer A, Prähofer H, Grünbacher P (2019) Change impact analysis for maintenance and evolution of variable software systems. Autom Softw Eng 26(2):417–461
15. Sharma S, Vijayvargiya S (2022) Modeling of software project effort estimation: a comparative performance evaluation of optimized soft computing-based methods. Int J Inf Technol 14(5):2487–2496
16. Dromey RG (2006) Formalizing the transition from requirements to design. In: Liu Z (ed) Mathematical frameworks for component software: models for analysis and synthesis. World Scientific, pp 173–205
17. Li M, Vitányi P (2008) An introduction to Kolmogorov complexity and its applications. Springer
18. Zhou D, Huang J, Schölkopf B (2006) Learning with hypergraphs: clustering, classification, and embedding. Adv Neural Inform Process Syst 19:1601–1608
19. Lian W, Dromey RG (2004) From requirements change to design change: a formal path. In: Proceedings of the Second International Conference on software engineering and formal methods, 2004, pp 104–113
20. Jayatilleke S, Lai R (2013) A method of specifying and classifying requirements change. In: Proceedings of the 22nd Australian Conference on software engineering, 2013, pp 175–180
21. Jayatilleke S, Lai R, Reed K (2018) Managing software requirements changes through change specification and classification. Comput Sci Inf Syst 15(2):321–346
22. McGee S, Greer D (2011) Software requirements change taxonomy: evaluation by case study. In: 19th International Requirements Engineering Conference, 2011: IEEE, pp 25–34
23. Gates AQ, Kreinovich V, Longpre L (1998) Kolmogorov complexity justifies software engineering heuristics. Departmental Technical Reports (CS). 558. https://scholarworks.utep.edu/cs_techrep/558
24. Ahmed K, Wen L, Sattar A (2014) iRE: a semantic network based interactive requirements engineering framework. In: Second World Conference on complex systems (WCCS), 2014: IEEE, pp 171–177
25. FAST-NUCES https://cfd.nu.edu.pk/. Accessed 13 Sept 2023
26. Kretsou M, Arvanitou E-M, Ampatzoglou A, Deligiannis I, Gerogiannis VC (2020) Change impact analysis: a systematic mapping study. J Syst Softw 174:110892
27. Hammad M, Collard ML, Maletic JI (2011) Automatically identifying changes that impact code-to-design traceability during evolution. Softw Qual J 19(1):35–64
28. Al-Saiyd N, Zriqat E (2015) Analyzing the impact of requirement changing on software design. Eur J Sci Res 136:1–11
29. Sudin MN, Ahmed-Kristensen S (2011) Change in requirements during the design process. In: 18th International Conference on engineering design, 2011, pp 200–208
30. Kchaou D, Bouassida N, Ben-Abdallah H (2017) UML models change impact analysis using a text similarity technique. IET Softw 11(1):27–37
31. Kilpinen MS (2008) The emergence of change at the systems engineering and software design interface. University of Cambridge
32. Spijkerman W (2010) Tool support for change impact analysis in requirement models: exploiting semantics of requirement relations as traceability relations. University of Twente
33. Goknil A, Kurtev I, Van Den Berg K, Spijkerman W (2014) Change impact analysis for requirements: a metamodeling approach. Inf Softw Technol 56(8):950–972
34. Goknil A, Kurtev I, van den Berg K, Veldhuis J-W (2011) Semantics of trace relations in requirements models for consistency checking and inferencing. Softw Syst Model 10(1):31–54
35. Ali N, Lai R (2016) A method of requirements change management for global software development. Inf Softw Technol 70:49–67
36. Lallchandani JT, Mall R (2011) A dynamic slicing technique for UML architectural models. IEEE Trans Software Eng 37(6):737–771
37. Briand LC, Labiche Y, O'Sullivan L, Sówka MM (2006) Automated impact analysis of UML models. J Syst Softw 79(3):339–352
38. Canfora G, Di Penta M (2007) New frontiers of reverse engineering. In: Future of Software Engineering (FOSE'07), 2007: IEEE, pp 326–341
39. Bashir MF, Qadir MA (2006) Traceability techniques: A critical study. In: *10th* IEEE International Multitopic Conference, INMIC, 2006, pp 265–268
40. Verhanneman T, Piessens F, Win BD, Joosen W (2005) Requirements traceability to support evolution of access control. In: Presented at the Proceedings of the workshop on Software engineering for secure systems—building trustworthy applications, St. Louis, Missouri, 2005
41. Dromey RG (2001) Genetic software engineering-simplifying design using requirements integration. In: IEEE Working Conference on Complex and Dynamic Systems Architecture, 2001, pp 251–257
42. Saraf I, Iqbal J (2019) Generalized software fault detection and correction modeling framework through imperfect debugging, error generation and change point. Int J Inf Technol 11(4):751–757