Bridging the Domain Language Divide in Automatic Multiplatform Single Page Application Generation with Knowledge Graph

Le Yen Chi Pham[1], Duc Minh Le[2], Minh Khue Hoang[2], Dang Duc Anh Nguyen[2], Quang Tung Ta[2]
[1]Swinburne University of Technology, Hawthorn, Victoria, Australia
103430040@student.swin.edu.au
[2]Department of Information Technology, FPT University, Swinburne Vietnam, Hanoi, Vietnam
duclm20@fe.edu.vn, {khuehmswh00983, anhnddswh01109, tungtqswh01093}@fpt.edu.vn

***Abstract*—Automating the software development process has long been a central topic in software engineering. The advent of modern AI methods and tools, especially the recent rapid advancement of generative AI has brought renewed research interest in automation capability. A main limitation of the current AI-based approaches is that they are only able to generate code fragments and not the complete software module. Our recent works have proposed a novel software generation solution for a popular type of web frontend application, named single page application (SPA). Our works use annotation-based domain-specific languages (aDSLs) to express the SPA design specification and code generators to generate complete SPA software modules automatically. However, our works currently lack the ability to express the SPA specifications in higher-level languages, necessary to leverage the AI methods and tools. In this paper, we aim to make a first novel step in bridging this gap by using a knowledge graph to express the SPA specification, bringing it closer to the domain expert languages. We construct and implement the knowledge graph model in a Java software framework, named JDA. After that, we develop an algorithm named SPKGen to generate aDSL-based SPA specification from the knowledge graph model. Subsequently, we evaluate the effectiveness of the knowledge graph model against real-world SPA case studies. We assess the knowledge graph model's ability to accurately represent the SPA configurations. Further, we evaluate the effectiveness of our method in generating the SPA components and configurations from the knowledge graph. The results of these evaluations demonstrate the viability of using knowledge graph for capturing and realizing SPA.**

***Index Terms*—Knowledge graph, Domain-Driven-Design, Generative software development, Single Page Application**

I. Introduction

Domain-Driven Design (DDD) provides principles and practices for building complex software systems from a core domain model [1]. Generative programming techniques, such as code generation and metaprogramming, can help automate software development for these complex applications [2]. Additionally, Domain-Specific Languages (DSLs) allow for the creation of languages tailored to specific problem domains, which can be beneficial for automating the development of Single Page Applications (SPAs) [3]. Our previous work has proposed a novel method to automatically generate frontend applications for multiple SPA frameworks using annotation-based DSLs (aDSLs) and code generators implemented in the JDA framework [4]–[6]. However, a key limitation of our work is the lack of a formal mapping between the aDSL specifications and higher-level design models, such as those expressed in UML or knowledge graphs [7], [8]. Although knowledge graph has been used in software development in general, to the best of our knowledge, there have been no work researching the application of knowledge graph to SPA. This paper aims to bridge this gap by using knowledge graph to formally express the aDSL-based SPA specifications. Specifically, we propose a knowledge-graph-based language, named Software Configuration Graph Language (SCGL), for representing both the metamodel and model elements of SPA specification. To ease SPA application in SCGL, we define an algorithm named SPKGen, which transforms a software configuration graph, expressed in SCGL, into a software configuration class model. We leverage our recent research [5], [6] to automatically generate SPAs from this model. We evaluate the performance of SPKGen and the effectiveness of SCGL for expressing SPA design. The results demonstrate the overall effectiveness of our approach.

The rest of this paper is structured as follows. Section II provides an overview of the relevant concepts and prior research that serve as the foundation for this work. Section III defines SCGL as a knowledge-graph-based language for SPA specification. Section IV presents the SPKGen algorithm. Section V evaluates the effectiveness of SCGL and knowledge graph integration in automating SPA design and code generation. Finally, Section VI concludes the paper and outlines potential future research directions.

II. Background and Related Works

Our work presented in this paper lies at the interactions of these topic domains: (1) multiplatform SPA generation and (2) application of knowledge graph in the development of single page application (SPA).

A. Expressing SPA design using SCCL*

In our previous work, we proposed a fundamental metamodel for SPAs which is expressed using an annotation-based DSL called SCCL* [5]. This metamodel was derived from the requirements of four popular front-end frameworks: Angular, React, React Native, and Vue.js (referred to as the "ARNV" frameworks). The SPA metamodel describes the foundational architecture of SPAs, including key elements such as modules, components (main form, object form, object browser), routing, and backend communication. However, building SPAs using this metamodel remains complex, as modern front-end development often involves multiple visual components provided by third-party libraries, such as Bootstrap. To address this, we adopted a systematic approach that integrates front-end design patterns, allowing a more structured representation of visual components, whether custom-built or third-party. This approach streamlines development through code generation and automation.

This paper uses the CourseMan software example from our previous work, focusing on six key classes: SCCCourseMan, ModuleStudent, ModuleCourse, ModuleAddress, and ModuleEnrolment, which represent entities in a course management system. To simplify, we will illustrate SCCCourseMan and ModuleStudent, as the other modules follow similar implementations. This example serves as a benchmark for evaluating SCGL and SPKGen algorithms.

Listing 1. SCCCourseman (Source: [5])

```java
@RestController
@SystemDesc( appName="Courseman",dsDesc = @DSDesc(
   type="postgresql", dsUrl = "http://dataserver:5432/courseman",
   user = "admin", password = "password", connType=Client ),
modules = { ModuleMain.class, ModuleAdress.class,
   ModuleCourseModule.class, ModuleEnrolment.class,
   ModuleStudent.class }) public class SCCCourseMan { }
```

Listing 2. Student Module (Source: [5])

```java
@RestController
@ModuleDesc( name="Module: Student", model="Student",
   view=View.class, containmentTree = @CTree {
   root = "Student", edges = {
        @CEdge (p=Student.class, c=Address.class),
        @CEdge (p=Student.class, c= Enrolment.class) }})
class ModuleStudent {
   @AttributeDesc(label="Form: Student") private String title;
   @AttributeDesc(label="Id", type=JTextField.class)
   private int id;
   @AttributeDesc(label="Name", type=JTextField.class)
   private String name;
   @AttributeDesc(label="Gender", type=JComboField.class)
   private Gender gender;
   @AttributeDesc(label="Dob") private Date dob;
   @AttributeDesc(label="Address", type=JDataPanel.class)
   private Address address;
   @AttributeDesc(label="Form: Enrolment")
   private Collection <Enrolment> enrols; }
```

Listing 1 demonstrates the SCCL* configuration for SCCCourseMan and its relationship with various modules. Similarly, Listing 2 shows the SCCL* configuration for the Student module within the CourseMan software. These SCCL* configurations are written using Java's syntax. Example SPA views of the ModuleStudent of CourseMan application that are generated from the SCCL* configuration for the ARNV frameworks are presented in [5].

B. JDA: Generating multiplatform SPAs from SCCL*

The SPA development process in this approach begins by creating an SPA model using the SCCL* language, supported by the JDA framework [5]. This model is then input into the SPAGen code generator, a JDA module, which produces the actual SPA code based on the specified front-end platform.

In our previous research [5], [6], we have proposed a method for automatically generating frontend applications across multiple frameworks. This approach utilizes a set of annotation-based DSLs (aDSLs) to express the SPA model. We developed code generators in object-oriented programming languages (OOPLs) to parse the aDSL specifications and generate the corresponding frontend software. The aDSLs allow for the specification of the target frontend framework, such as Vue, React, Angular. The code generators incorporate well-defined UI design patterns to improve the usability of the generated software. We implemented our aDSLs and code generators within a Java software framework named JDA [4], [5].

However, a key limitation is the lack of a formal method to map the aDSL specifications to higher-level design models, such as those expressed in UML [7] or knowledge graphs [8]. Establishing a formal mapping between the aDSL specifications and these higher-level design models could provide significant benefits. It allows us to leverage the capabilities of our code generators not only for frontend software development.

Knowledge graphs, in particular, offer a powerful way to represent and reason about the relationships and domain knowledge within software applications. Mapping aDSLs to a knowledge graph model would capture contextual information between components and business logic, improving code generation and enabling semantic queries. This formal mapping could also advance generative AI in software engineering.

C. Applications of Knowledge Graph in SPA Construction

A knowledge graph (KG) is a structured representation of knowledge, typically used to organize information in a machine-readable format. It is a graph database connecting entities, concepts, and their relationships. In a knowledge graph, entities are represented as nodes, and the relationships between them are represented as edges [9]. Knowledge graphs have numerous applications, particularly in Artificial Intelligence (AI). They facilitate tasks such as question answering, recommendation systems, and semantic search by organizing structured knowledge to enable efficient information retrieval and analysis, enhancing various aspects of artificial intelligence [10]. By capturing and representing diverse types of knowledge and relationships, KGs can also be highly beneficial throughout the software development lifecycle [8], particularly in requirements engineering (modeling relationships between requirements and system components [11]), in software design and architecture (visualizing architecture and dependencies [12]), and in maintenance and evolution: tracking changes and bug reports linked to code.

Although KG has been widely used in software development, its application in expressing the metamodel of aDSLs for frontend applications remains unexplored. To verify this, we conducted two Google Scholar searches. The first query - "Knowledge Graph" AND (SPA OR "single page application") AND (generation OR construction OR development—returned no relevant studies among the top 200 results, focusing either *(i)* KG construction for domain-specific knowledge-base (e.g. tourism, procurement value chain, online tutoring assistant, question answering, etc.) or *(ii)* KG engineering (including automatic KG generation, visualisation, exploration and editing, query answering over KG, etc.). The second query extends the first to cover the frontend: “Knowledge Graph” AND (SPA OR “single page application” OR “frontend” OR “front end”) AND (generation OR construction OR development). Among the first 200 results, this second search returns a few additional articles concerning the general application of KG to software development (e.g. [13]), but none specifically addressed SPA or frontend generation.

This paper marks a key step in applying KG to SPA development, particularly for multi-platform SPAs that require uniform representation of cross-platform features. Specifically, we focus on aDSL-based software generation and propose expressing aDSL specifications in a knowledge graph format. This approach leverages KG to represent SCCL* for SPA generation, laying the groundwork for integrating generative AI with existing multiplatform code generators.

III. Representing SCCL* in SCGL

We use KG to express the SPA design specification embedded in SCCL*. We call the new language **Software Configuration Graph Language (SCGL)**. The benefit of KG is to represent the underlying language model (a.k.a the metamodel) and its models in the same graph. A model represents a software model that is instantiated from the metamodel for a specific problem domain. LHS of Figure 1 shows a diagram to visualize the design of SCGL using nodes and their relationships. Within the scope of this paper, we focus on the core elements of SCCL*. We use the popular Neo4j graph modeling language[1] as the base KG language.

To express the SCCL* model in Neo4j, we capture the concepts and relationships in SCCL* then map them to Neo4j node and relationship types. We create the necessary node labels and relationship types to match the entities and associations defined in the SCCL*'s metamodel. SCGL's metamodel expresses the design and constraints of SCCL*. In SCGL, a model defines a SCC model for a specific software.

Key relationships between entities, such as inheritance, composition, containment, and annotation, correspond to Neo4j relationships like "INSTANCE_OF," "HAS,", "HAS_CONTAINMENT_TREE" and "ATTACH_TO," preserving semantic connections and cardinality constraints. However, Neo4j does not natively support modeling with constraints, so we layered the graph design into three levels, each with specific rules to structure SCGL.

The central entities in the metamodel were directly translated into node labels in Neo4j, ensuring an accurate representation of concepts and their taxonomic structure. For example, the SCCCourseMan node is an SCG model for CourseMan. The target SPA configuration consists of a generation configuration (an instance of RFSGenDesc), a system configuration (an instance of SystemDesc), and module configurations, each represented by an instance of ModuleDescriptor with a data model (LHS of Figure 1).

[1]https://neo4j.com/use-cases/knowledge-graph/

A. Annotation, Class, Attribute, and Method node types

These node types represent various Java elements in SCGL. The other nodes may have the "IS_A" relationship with the element nodes to specify their type in Java. The Annotation nodes represent different types of Java annotations. They have an "IS_A" relationship with the "Annotation" node. Examples of annotation type nodes include "RFSGenDesc", "SystemDesc", and others, each named after the specific annotation they represent. The Class nodes represent Java classes. These class nodes have an "IS_A" relationship with the "Class" node.

B. Relationships

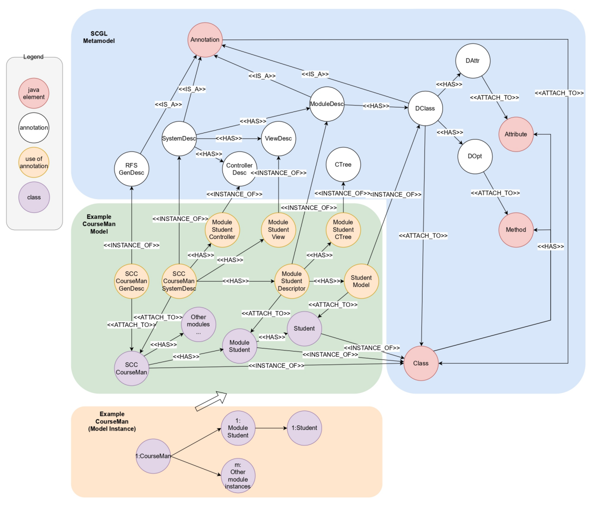
"ATTACH_TO" relationship with a class node, indicating which class the annotation is attached to. In addition, "INSTANCE_OF" relationship with an annotation type node, indicating the type of annotation. Depending on the type of annotation instance, there may be additional relationships. First, "SystemDesc" annotation instances have one or more "HAS_MODULE" relationships with module class nodes. For example, a "SystemDesc" for "SCCCourseMan" may have "ModuleMain", "ModuleStudent", etc. Second, "ModuleDescriptor" annotation instances have exactly one "HAS_MODEL" relationship with a domain class node, indicating the domain model for that module. They also have a "HAS_VIEW" relationship with a "ViewDesc" node and a "HAS_CONTROLLER" relationship with a "ControllerDesc" node. Third, "CTree" (Containment Tree) annotation instances have one or more relationships “HAS_CONTAINMENT_EDGE“ with other modules via domain class nodes. For example, a "CTree" for "ModuleEnrolment" may contain "Student", "Module".

C. Metamodel

The metamodel forms the foundation of SCGL, defining the structural elements and relationships within the SCCL* metamodel. It encompasses both core aDSL elements and SPA-specific components, including SetupDesc, ModuleDesc, ViewDesc, ControllerDesc, DClass, and DAttr. These elements describe various software components and their interactions, ensuring a structured and consistent approach to modeling software configurations.

D. Model

The Model is an instance of the Metamodel, conforming to the design and constraints established in the Metamodel layer. This layer represents the specific configuration of the software, including various components and their relationships. LHS of Fig.1 shows an example model for the CourseMan SPA example. It includes nodes such as SCCCourseManGenDesc, SCCCourseManSystemDesc, ModuleStudentDescriptor, and Student Model. These define the specific entities and interactions in the software application, such as course management, module management, and student-related functionality.



Algorithm 1 SPKGen Algorithm

```
function EXTRACT_KNOWLEDGEGRAPH(KG_Connection)
    classNodes ← queryClassNodes(KG_Connection)
    for all classNode in classNodes do
        className ← classNode.getProperty('name')
        classPackage ← classNode.getProperty('package')
        outputFilePath ← classPackage + className
        path ← classPackage + className
        if path ≠ null then
            copyFile(path, outputFilePath)
        else
            continue
        end if
        classModel ← new ClassModel()
        classModel.name ← className
        classModel.package ← classPackage
        classModel.annotations ← new List<string>
        annotationPairs ← queryAnnotations(knowledgeGraph, classNode)
        for all (annotationNode, annotationTypeNode) in annotationPairs do
            annotationTemplate ← getAnnotationTemplate(annotationTypeNode.name)
            annotationModel ← AnnotationModel.nodeToModel(annotationNode, annotationTypeNode.name)
            annotationBoundOutput ← bind(annotationTemplate, annotationModel)
            classModel.annotations.add(annotationBoundOutput)
            writeToFile(outputFilePath, bind(classTemplate, classModel))
        end for
    end for
end function
```

Fig. 1. SCGL structure and CourseMan example implementation and SPKGen Algorithm.

IV. SPKGen: From SCGL to SCCL*

To ensure that the Neo4j knowledge graph schema aligns with SCCL*, we developed a method to query the graph and reconstruct the configurations as SCCL* in Java. This algorithm, called SPKGen (Knowledge-Graph-Based Single Page Application Generation), generates SPAs from a knowledge graph written in SCGL by transforming the graph into SCCL*. The transformed SCCL* is then used, along with the JDA framework, to generate the target SPA. This paper focuses on the essential elements of SCCL* supported by SCGL, with additional features planned for future work.

We use the Cypher query language[2] of Neo4j to query the KG of SCGL to retrieve the design information. This allows us to programmatically extract the node labels, relationship types, and their associated properties that represent the schema using our algorithm (RHS of Figure 1). The detailed description and the steps of SPKGen are described next.

A. Query all class nodes from the knowledge graph

The algorithm starts by querying the knowledge graph to retrieve all the class nodes. This is done using a Cypher query like MATCH [n:CLASS]-[:IS_A]->[e:JAVA_ELEMENT name: "Class"] RETURN n.

B. Check each class node for the "path" property

The algorithm checks if the node has a "path" property for each class node retrieved in the previous step. If the "path" property exists, it means this class will not be generated. Instead, it will be programmatically copied over to the output directory from the Java file indicated by the "path" property. If the "path" property does not exist, the algorithm proceeds to the next step to generate this class.

C. Generate the classes

For each class node that needs to be generated (i.e., does not have the "path" property), the algorithm retrieves the class name, inserts it into a Freemarker template, and queries the knowledge graph to obtain the annotations and types.

For each annotation and its type, the algorithm selects the appropriate Freemarker template. If the type is "RFSGenDesc", the template is populated directly with data from the annotation node. For "SystemDesc", the algorithm queries all classes related via -[:HAS_MODULE]->(n:CLASS), assigning the resulting nodes to the "modules" property while retrieving other properties from the annotation node. For "ModuleDescriptor", it queries the class linked via -[:HAS_MODULE]->(n:CLASS). If the KG is correct, there is only one `n`, a domain class, whose name is included in the ModelDesc. Other properties are either static or retrieved from the annotation node. Similarly, for "CTree", the algorithm queries classes linked via [:HAS_CONTAINMENT_EDGE]->(n:ANNOTATION), assigning the resulting nodes to the "edges" property while extracting other details from the annotation node. The final output is a string binding the annotation template with the data, which is appended to the $annotations section of the class template. The completed class is then written to the Java source file.

V. Evaluations

The primary focus of our evaluation is to assess the effectiveness of the proposed approach in meeting the SCGL requirements outlined in Section IV. To that end, we have formulated the following questions to guide our evaluation:

RQ1: (SCGL) What is the extent to which SCGL correctly expresses the SPA design through SCCL*?

RQ2: (SPKGen) What is the effectiveness of the SPKGen algorithm in generating the SCC* for a given SPA?

RQ3: (SCGL for code generation) How effective is SCGL in generating SPA code?

RQ4: (Effectiveness of Knowledge Graph) - What are the benefits of integrating knowledge graphs in enhancing generative AI for code generation?

We have implemented SCGL and SPKGen as part of JDA, and the code is available online in the GitHub repository:

[2]https://neo4j.com/product/cypher-graph-query-language/

https://github.com/jdomainapp/jdm-scgl. Our implementation uses Java and adds the following third-party libraries to JDA: (i) `org.neo4j.driver`: This Neo4j library provides a driver that helps connecting to the AuraDB and query the knowledge graph from it; (ii) `org.freemarker`: This library allows creating template for the generated code file.

A. The expressiveness of SCGL for SPA design

To assess the effectiveness of the SCGL knowledge graph in representing the SCCL* software configuration model, we developed a systematic approach using a mapping table. Table II (Fig 2) establishes a clear correspondence between the key SCCL* concepts and their equivalent representations in the SCGL knowledge graph. The one-to-one correspondence between the SCGL and SCCL* concepts facilitates the translation and integration of software configuration data between the two languages. The table thus serves as a reference for developers and researchers working with SCGL and SCCL* to ensure that the SCGL knowledge graph can effectively capture and represent the software configuration information as defined in the SCCL* model.

B. SPKGen Testing

To evaluate the effectiveness of the proposed SPKGen algorithm, we adopted the CourseMan example from our previous work as a benchmark. In this evaluation, we created a SCG model to represent the CourseMan application. Then, we used the SPKGen algorithm to help JDA to generate the CourseMan application implementation (Listing 3 and 4 in Figure 2). We compare the generated result with the original CourseMan in Section II. The comparison showed that the generated application supports all the basic elements listed in Table II in Figure 2.

C. SPKGen Performance

We further conduct a time complexity evaluation of the SPKGen algorithm to assess its computational efficiency and scalability. These performance characteristics are critical for the practical application of the SPKGen algorithm in real-world software development scenarios.

Theorem V.1. ***(SPKGen Performance)*** *The time complexity of our SPKGen algorithm is quadratic to the number of class nodes in the input graph.* □

Proof. Assuming the time complexity of writing to a file is $O(1)$, the overall time complexity of the SPKGen algorithm can be broken down as follows.

In the first step, the algorithm queries all class nodes in the Neo4j knowledge graph. Let the number of class nodes be denoted as n, and the time complexity for this step is $O(n)$. Next, the algorithm loops through the list of class nodes, which also incurs a time complexity of $O(n)$. For each class node, it performs several sub-steps. First, it reads basic class information such as the name and package from the node, which is $O(1)$. Then, it checks for the existence of a "path" property, again $O(1)$. If the property exists, the class is copied from the original file (with time complexity $O(1)$), otherwise, the algorithm proceeds to generate the class. Class generation involves creating a class model object, which is $O(1)$, querying the class annotations (with time complexity $O(m)$, where m represents the number of annotations), and for each annotation, creating and binding a model object to a template, also $O(m)$. Finally, the class model is bound to a template and written to the output file, with a time complexity of $O(1)$.

Assuming that k nodes have the "path" property, and $n-k$ nodes do not, the overall time complexity of the SPKGen algorithm is the sum of the time complexities of the individual steps, resulting in:

$$\begin{aligned} &O(n) + O\left(k \cdot (1+1+1) + (n-k) \cdot (1+m+m+1)\right) \\ &\qquad = O(n) + O(2nm + 2n + k - 2km) \end{aligned}$$

This simplifies to $O(nm)$, which can be approximated to $O(n^2)$, since in typical software, n is generally greater than m. □

D. Effectiveness of Knowledge Graph

An experiment was conducted to assess the impact of Knowledge Graphs (KG) on code generation using ChatGPT and Gemini. The goal was to determine whether SCGL-enhanced GenAI produces more optimized and accurate outputs than models relying solely on natural language prompts.

Methodology: The experiment involved generating frontend React code for modules such as Student, Address, Course, Enrollment, and StudentClass. Each AI model was tested three times by calling its API to ensure consistency and reliability. The generated code was analyzed for key technical features and compared across models. The results are available in a GitHub repository https://github.com/jdomainapp/doc-scgl-soict2024, organized into two folders: **CourseManScenario1** and **CourseManScenario2**, corresponding to the two different conditions.

Scenario 1: Natural Language Prompts ChatGPT was provided with prompts written entirely in natural language, without additional contextual information.

Scenario 2: Knowledge Graph-Enhanced Prompts This scenario extended the experiment by incorporating structured knowledge from knowledge graphs, capturing high-level and detailed relationships between the modules.

Results:

TABLE I
COMPARISON OF SCENARIO 1 AND 2 RESULTS

Aspects	Scenario 1	Scenario 2
Total Prompts Used	3	2
Parent-child Relationships (binary)	0	1
Code Repetition (%)	15%	5%
API Integration (binary)	0	1
Modules Generated	5	7
Validation Coverage (%)	50%	90%
Gemini Prompts and Results	Link to Google Colab	Link to Google Colab
GPT Prompts and Results	Link to Google Colab	Link to Google Colab
GitHub Repository	Link to CourseManScenario1	Link to CourseManScenario2

In Scenario 1, generating complete modules required multiple iterations due to incomplete fields and unclear relationships, particularly in complex structures like parent-child hierarchies. Without structured guidance, some models produced

Listing 3. Generated SCCCourseman

```
@SystemDesc( appName="Courseman", dsDesc = @DSDesc( type="
     postgresql", dsUrl="http://dataserver:5432/domainds", user
      = "postgres", password = "postgres"),
modules = { ModuleMain.class, ModuleCourseModule.class,
 ModuleEnrolment.class, ModuleStudent.class,
 ModuleAdress.class, ModuleStudentClass.class})
public class SCCCourseMan { }
```

Listing 4. Generated Student Module

```
@ModuleDesc(name="ModuleStudent",
    modelDesc = @ModelDesc(model = Student.class),
viewDesc = @ViewDesc( formTitle = "Form: Student",
    domainClassLabel = "Student", imageIcon = "Student.png"),
controllerDesc = @ControllerDesc(controller = ControllerBasic.
     class), type = ModuleType.DomainData)
public class ModuleStudent {
@AttributeDesc(label = "Student class") private String title;
@AttributeDesc(label="Id",type=JTextField.class)private int id;
@AttributeDesc(label = "Name", type = JTextField.class)
private String name;
@AttributeDesc(label = "Gender", type=JComboField.class)
private Gender gender;
@AttributeDesc(label = "Dob") private Date dob;
@AttributeDesc(label = "Address", type=JDataPanel.class)
private Address address;
@AttributeDesc(label = "Enrolments")
private Collection <Enrolment> enrols; }
```

TABLE II. SCGL and SCCL* OOPL and Software Configuration Elements

No.	SCGL	SCCL*
OOPL Elements		
1	"Annotation" node type, "ATTACHED_TO" relationship	Annotation
2	"Class" node type	Class
3	"Attribute" node type, "HAS" relationship	Attribute
4	"Method" node type, "HAS" relationship	Method
Software Configuration Elements		
5	"RFSGenDesc" node	RFSGenDesc annotation
6	"SystemDesc" node	SystemDesc annotation
7	"ModuleDesc" node	ModuleDescriptor annotation
8	"ViewDesc" node	ViewDesc annotation
9	"ControllerDesc" node	ControllerDesc annotation
10	"DClass" node	DClass annotation
11	"DAttr" node	DAttr annotation
12	"DOpt" node	DOpt annotation

Fig. 2. Listings of Generated SCCCourseman and Student Modult and Table of Configuration Elements.

simplified forms instead of distinct modules. In Scenario 2, incorporating a Knowledge Graph (KG) reduced iterations, improved accuracy, captured hierarchical relationships, minimized redundancy, and enabled API integration. Nested forms were also generated more effectively, streamlining the process.

This experiment shows that KG-enhanced GenAI significantly improves code generation by enhancing accuracy, reducing manual intervention, and better understanding module relationships. Further integration of KG into GenAI could enhance its practical application in software development.

Discussion: The evaluation, combined with results from JDA and SCGL, highlights the significant benefits of using KG to structure code generation. While JDA-generated code offers a superior user interface compared to GPT-generated code (see folders CourseManJDA and CourseManScenario2 in our GitHub repository), both approaches demonstrate strengths in performance, including code optimization and module generation, as shown in JDA's evaluation and Table I for GenAI.

SCGL stands out by effectively representing complex relationships, leading to more optimized code with less redundancy. This underscores the value of SCGL and the broader potential of KG in improving automatic code generation, particularly for large-scale, complex projects. The integration of SCGL with KG enhances efficiency and expands the possibilities for applying KG in code generation.

VI. Conclusion

This paper presented a novel approach to automating the SPAs development process by using knowledge graph technology to express software design specifications. Our key contributions are the development of a knowledge-graph-based language (SCGL) that can capture the essential elements of SPA design, and the implementation of this language in a software framework (JDA) to generate SPAs. This framework allows domain experts to define SPA specifications in SCGL and automatically generates the complete SPA. This work represents an important step towards bridging the gap between domain requirement language and design specifications in SPA development. Future extensions include enriching SCGL with SPA design patterns and investing in the use of AI-powered software engineering tools with SCGL.

Acknowledgement This work was conducted in the Software Engineering Innovation Lab at Swinburne Vietnam. The authors would like to thank Mr. Van Cong Nguyen for his contributions in the earlier stages of this research.

References

[1] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley Professional, 2004.
[2] K. Czarnecki, "Generative Programming: Methods, Techniques, and Applications Tutorial Abstract," in *Software Reuse: Methods, Techniques, and Tools*, C. Gacek, Ed. Berlin, Heidelberg: Springer, 2002, pp. 351–352.
[3] M. Fowler and T. White, *Domain-Specific Languages*. Addison-Wesley Professional, Sep. 2010.
[4] D. M. Le, D.-H. Dang, and H. T. Vu, "jDomainApp: A Module-Based Domain-Driven Software Framework," in *Proc. 10th Int. Symp. on Information and Communication Technology (SOICT)*. New York, USA: ACM, 2019, pp. 399–406.
[5] D. M. Le, A. P. Nguyen, L. Q. Tran, H. T. Le, and H. V.-A. Tong, "Generating Multi-platform Single Page Applications: A Hierarchical Domain-Driven Design Approach," in *Proc. Int. Conf. 11th SOICT 2022*, ser. SoICT '22. Hanoi, Vietnam: ACM, 2022, pp. 344–351.
[6] D. M. Le, C. V. Nguyen, L. Q. Tran, and K. M. Hoang, "Modularly Generating Multi-platform Single Page Applications with FrontEnd Patterns," in *Intelligent Systems and Networks*, ser. LNCS. Singapore: Springer Nature, 2024, pp. 552–562.
[7] Object Management Group (OMG), "Unified Modeling Language version 2.5.1," 2017, accessed: 2024-08-22.
[8] L. Wang, C. Sun, C. Zhang, W. Nie, and K. Huang, "Application of knowledge graph in software engineering field: A systematic literature review," *Information and Software Technology*, vol. 164, p. 107327, Dec. 2023.
[9] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, and A. Wahler, "Introduction: What Is a Knowledge Graph?" in *Knowledge Graphs: Methodology, Tools and Selected Use Cases*, D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, and A. Wahler, Eds. Cham: Springer International Publishing, 2020, pp. 1–10.
[10] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A Survey on Knowledge Graphs: Representation, Acquisition, and Applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, Feb. 2022, conference Name: IEEE Transactions on Neural Networks and Learning Systems.
[11] J. Kasser, "A framework for requirements engineering in a digital integrated environment (fredie)," 01 2000.
[12] M. Ali Babar, T. Dingsøyr, P. Lago, and H. Vliet, *Software Architecture Knowledge Management*, Jan. 2009, journal Abbreviation: Software Architecture Knowledge Management Publication Title: Software Architecture Knowledge Management.
[13] Z.-Q. Lin, B. Xie, Y.-Z. Zou, J.-F. Zhao, X.-D. Li, J. Wei, H.-L. Sun, and G. Yin, "Intelligent Development Environment and Software Knowledge Graph," *J. Comput. Sci. Technol.*, vol. 32, no. 2, pp. 242–249, Mar. 2017.