Publications

Available Online

J. Sci. Res. **15** (3), 637-649 (2023)

JOURNAL OF SCIENTIFIC RESEARCH

www.banglajol.info/index.php/JSR

Implementation of Test Suites using Enhanced State Chart Diagrams: A Case Study

K. Gupta[*], P. Goyal

Department of Computer Application and Information Technology, Shri Guru Ram Rai University, Dehradun-248001, India

Received 6 January 2023, accepted in final revised form 28 May 2023

Abstract

An essential part of software engineering is testing. A series of pre-testing tasks should be completed before performing testing tasks. Implementing test suites is one of the pre-testing phases. In this work, a case study has been used to implement the suggested method for implementing test suites. The strategy is based on an analysis of UML (Unified Modelling Language) enhanced state chart diagrams (SCDs). UML state chart diagram analysis, AD (activity diagram) conversion into a graph, AG (activity graph) simplification, and test suite implementation are all steps in the generating process.

Keywords: Software engineering; Software testing; UML state chart diagrams; Activity diagram; Test suites.

doi: http://doi.org/10.3329/jsr.v15i3.63787 J. Sci. Res. **15** (3), 637-649 (2023)

1. Introduction

One step in the software development life cycle is software artifact testing (SDLC). Some work must be completed as pre-testing operations, such as implementing the test suites [1]. Software testing is necessary to ensure that the software complies with its specifications, but if carried out improperly, it may result in even more problems. Putting test suites into place aids in locating the software's testing procedures that need improvement [2]. The test suites are necessary for software testers to use while they evaluate the application or software. Although test suites can produce excellent testing, they are expensive in terms of the time and resources they use. The problem has been explored from a wide variety of perspectives, although many of them lack comprehensive case studies.

A case study will be provided in this paper in a comprehensive manner. The case study was chosen to be the Hospital Management System (HMS) [3]. A number of software engineering approaches have been taught using this well-known case study. The approach will be thoroughly discussed using the chosen case study, going over each step and procedure. The outcomes will be presented in the end.

[*] *Corresponding author*: kritikagupta47@yahoo.co.in

There are various sections in this research article. The field literature review is covered in the next section. The strategy is then outlined in a section after that. Subsections in the next section explain the case study in more detail. The research paper's conclusion part serves as its final bow.

2. Literature Review

To implement the test suites, a variety of strategies have been suggested. The input used in different implementation strategies for test suites differs. It's crucial to consider how the suggested methodologies approach UML models, even if most of the literature concentrates on the implementation of test suites. A survey of the literature is included in this section for some of the earlier ideas.

The effectiveness of automated test suite implementation strategies for the Python project Atomic Simulation Environment used in the material sciences is examined by Trubenbach *et al.* [1]. The Principal Component Analysis approach by Li *et al*. [2] is suggested in this study for implementing multi-criteria test suites. It delivers better test suite implementation performance and achieves higher or equal coverage. Gupta *et al*. [3] suggested in this study the implementation of test suites by using a use case and activity diagram, but these test suites contain repeated information or edges. By addressing the issues caused by concurrency in object-oriented software, Khandai *et al*. [4] suggested a technique for creating test suites via sequence diagrams (SDs). In this study, Wang *et al*. [5] concentrated on applying evolutionary algorithms for automatically implementing test suites.

The automatic construction of software test suites and their differentiation using a genetic algorithm (GA) were proposed by Haga *et al*. [6] in this work. The conversion processes from UML state chart diagram (SCD) to Petri Net to build automatic test tree and those using SCD were detailed by Chang *et al*. in this paper [7]. The author of this study, Kaur *et al*. [8], include unique methods such as physically merging test path implementation and analysis criteria and automatically applying tools created using prefix-based and Chinese postman procedures. In this article, Jena [9] offered a technique for creating an AFT (Activity Flow Table) with AD and then converting that Table into an AFG (Activity Flow Graph). The writer implemented the test paths and applied the action analysis criterion for AFG (Activity Flow Graph) traversal. Li [10] examines the straightforward modeling technique in this article. The author implemented the test suites and employed Use Case Sequence Diagrams. Kumar et al. [11] proposed a new approach for dispersed testing in state-based OOPs (Object Oriented Programming) in this study. In this study, Elallaoui [12] presented a technique for using the scrum process to convert consumer sections into sequence diagrams, which are then recycled to create test suites. Park *et al*. [13] create the state diagram in this study using a tree-based methodology. They took a well-researched approach to the problem by creating a tool for the programmed generation of a State Transition Mapping Tree (STMT) to demonstrate the application of STMT. In this article, Elallaoui [14] established a prototype for SDs (sequence diagrams) that can be automatically translated via model-to-model conversion after this prototype-to-text translation.

Vemuri [15] presented a strategy using the probabilistic method in this paper for identifying actors and use cases. For UML state chart diagrams, Bengal [16] created the coverage of the branch and Modified Condition/Decision Coverage (MC/DC) methods to calculate the coverage of code. Kumar *et al.* [17] provide a strategy in this study for identifying changes at the syntax and semantic levels by merging UML diagrams such as class diagrams, use cases, and activity diagrams. UML Activity Diagrams are used in this research by Verma *et al.* [18] to construct test suites and explain the user registration process. Ahmad *et al.* [19] give a thorough overview of the most recent methods for MBT (model-based testing) using a UML activity diagram (AD) in the article. Cvetkovic *et al.* [20] examined various methods for developing test suites in this work, employing either a single UML model or a collection of diagrams. Panda *et al.* [21] used a UML SCD through the Firefly and Differential Algorithm (DA), two methodologies that were inspired by nature, to create Model-Based Testing. Hamza [22] suggested an approach in this study that might be created on a use case analysis model. The generation method includes the study of UML use case diagrams, an AD transformation process, an AD simplification process, and an information extraction process.

Jena [23] defined the test suites in this study using a UML Activity diagram. Input from the system's AD was utilized to create the graphs. The writers then create test suites while taking into account potential test pathways. Tiwari *et al.* [24] demonstrate how to use UML diagrams, such as activity, state, and sequence diagrams, to streamline test suites. In the testing phase, models are developed using the Unified modeling language in order to implement test suites. Alsmadi [25] proposed a technique for test suite implementation estimates that is influenced by the principles of evolutionary algorithms. In this study, Kansomkeat *et al.* [26] proposed a method for autonomously collecting control flow data from decision facts and watching situations in ADs. Models for condition-classification bushes make use of this reality. Following design, test suites Table and suites are put into practice. An innovative approach for the automatic generation of test suites via XML process by ADs was presented by Boghdady *et al.* in this study [27].

3. Methodology

The five steps of the proposed methodology for test suites implementation are depicted in Fig. 1. The foundation of this proposed methodology will be the UML state chart diagrams (SCD), which are used to develop Hospital Management Systems (HMS). The necessary data is gathered from each HMS UML state chart object and integrated to implement a single activity diagram (SAD). The technique is dependent on a collection of SCOs on which the SAD will be executed. The SAD is then transformed into an Activity Graph (AG). A set of nodes and edges for linking those nodes that are indicated sequentially conversing to the information fetched from the previously implemented UML

state chart objects will be present in an AG. Following this, we modified the activity graph (AG) by deleting odd nodes like join, merge, and fork and implemented a simplified activity graph (SAG), which we then traversed to obtain the test suites. The final phase compares the traversing costs of the test suites created by AG and SAG. Each procedure will be broken into steps in the following section using the HMS as the approach.
The six steps in this process are as follows:

(i) Designing State Chart Objects (SCOs) for each Hospital Management System (HMS) object using a UML State Chart Diagram.
(ii) Combining all these SCOs into a Single Activity Diagram (SAD).
(iii) Converting SAD into Activity Graph (AG).
(iv) Simplifying the AG into Simplified Activity Graph (SAG).
(v) Implementing test suites by AG and SAG and comparing both test suites.

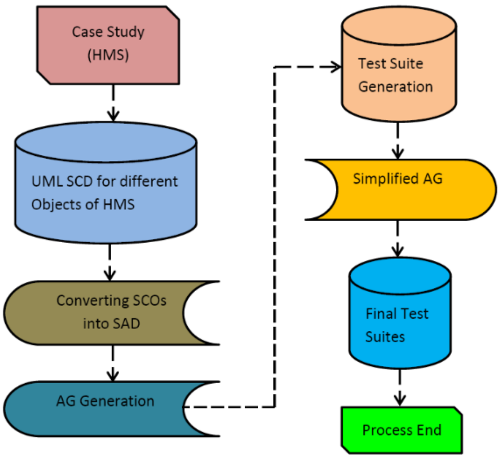


Fig. 1. Proposed framework for test suites implementation.

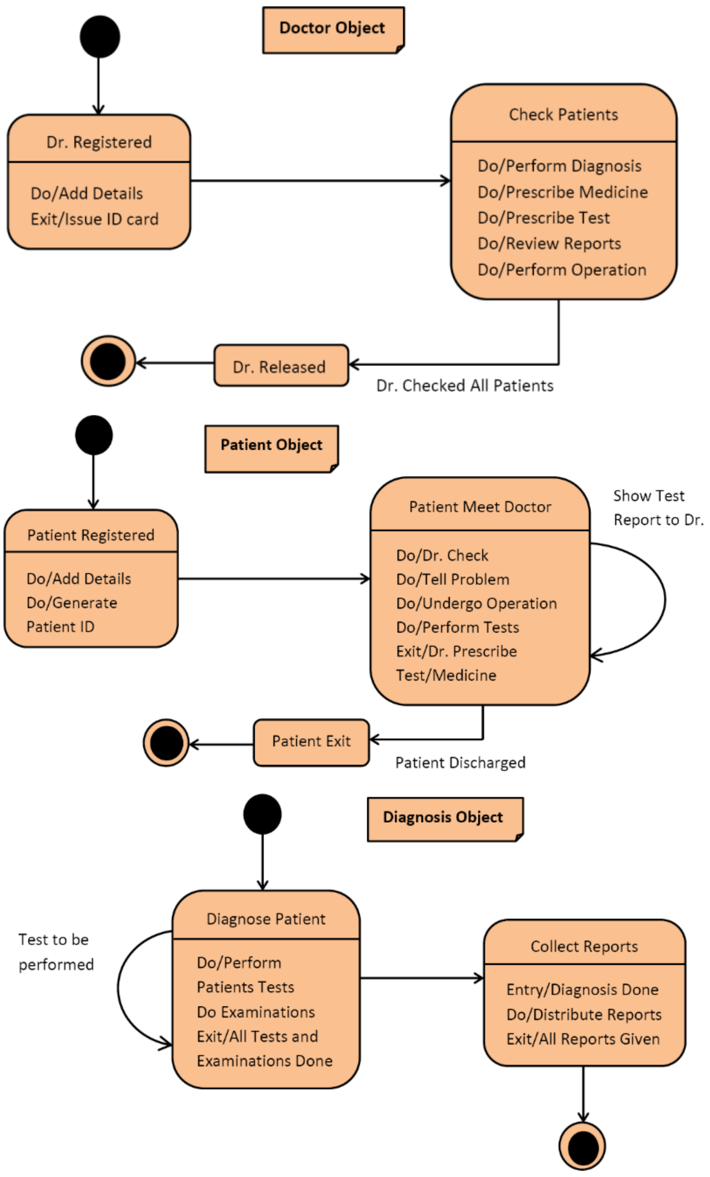
3.1. *Designing of state chart objects (SCOs) for each object of HMS using UML SCD (State Chart Diagram)*

The fundamental definition of SCDs has been covered before going into detail about the first step of the procedure. The SCD (State Chart Diagram) is well-defined mathematically as follows:
An SCD (State Chart Diagram) is a Five-tuple $SCD = (S, E, GC, S_i, S_f)$, where

(i) A finite collection of states called $S = \{s_{1,}s_2, \dots, s_n\}$ are those in which an object responds to events uniformly.
(ii) A finite sequence of changes from one state to another brought on by the input is represented by $E = \{e_{1,}e_2, \dots, e_n\}$.
(iii) $GC = \{gc_{1,}gc_2, \dots, gc_n\}$ A finite number of guard conditions, and gc_i corresponds to e_i, $Cond$ is a mapping from e_i to gc_i such that $Cond(e_i) = gc_i$.
(iv) $f \subseteq \{S \times E\} \cup \{E \times S\}$ is the flow relation between the States and Events.

(v) The starting state is $S_i \in S$, and the final state is $S_f \in S$. There is only one instance of event $e \in E$ such that $(S_i, e) \in f$, and for any instance of the event $e' \in E, (e', S_i) \notin f$ and $(S_f, e') \notin f$.

This step implements the State Chart Diagram (SCD) for several HMS objects, including (Doctor Object, Patient Object, Diagnosis Object, Pharmacy Object, and Payment Object). And each object is a container for the states and transitions of other objects. The many HMS objects that have been implemented are shown in Fig. 2 using a UML state chart diagram.



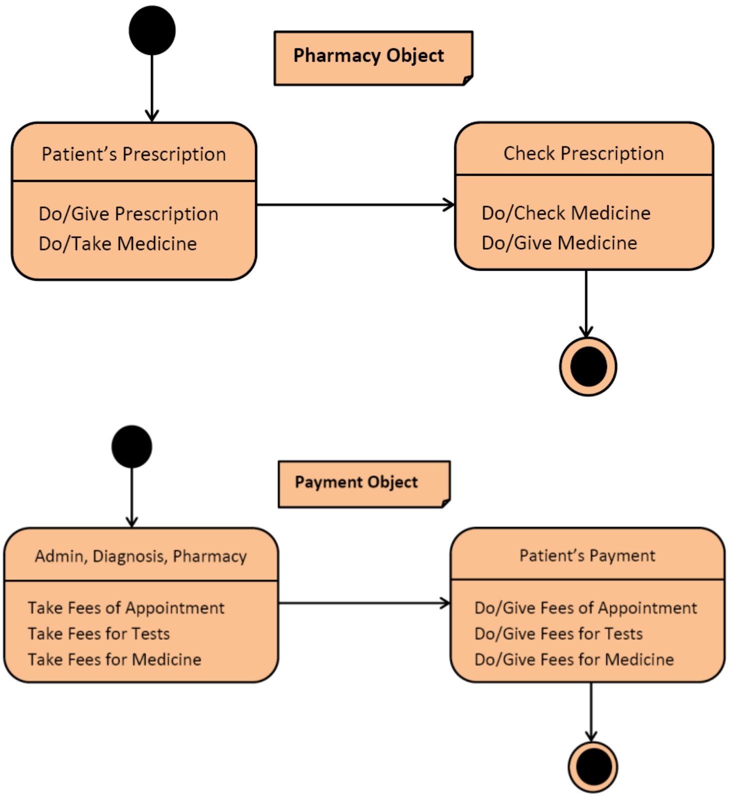


Fig. 2. Different objects implementation using state chart diagram for HMS.

3.2. *Combining different objects of HMS implemented by SCD into the SAD*

Before delving deeper into this process step, we first discussed the fundamental definition of an activity diagram. The AD (Activity Diagram) is well-defined mathematically as follows:

An AD is a Six-tuple $D = (A, T, F, C, A_i, A_f)$, where

(i) A is called a finite collection of activity states, where $A = \{A_1, A_2, \dots, A_n\}$.

(ii) T is called a finite collection of complete transitions, where $T = \{T_1, T_2, \dots, T_n\}$.

(iii) $GC = \{GC_1, GC_2, \dots, GC_n\}$ A finite number of guard conditions exists, and GC_i corresponds to T_i, and $Cond$ is a mapping from T_i to GC_i such that $Cond(T_i) = GC_i$.

(iv) $F \subseteq \{A \times T\} \cup \{T \times A\}$ is the flow relation between Activities and Transitions.

(v) The starting state is $A_i \in A$, and the last state is $A_f \in A$. There is exactly one transition $t \in T$ so that $(A_i, t) \in F$, and for any $t' \in T, (t', A_i) \notin F$ t'∈T, and $(A_f, t') \notin F$.

The transition is depicted by multiple merges, join, and fork conditions, and each state in SCD is represented by a separate activity. There are multiple activities shown in different ways for a single state. SCO symbols that have been transformed into activity symbols are depicted in Fig. 3. Fig. 4 illustrates how the process of integrating various HMS objects into a SAD, as performed by SCD, is accomplished. This is as follows:

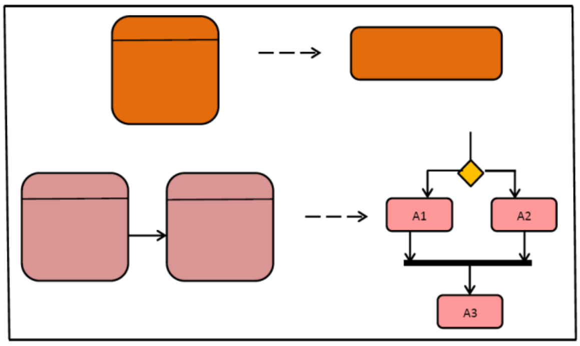


Fig. 3. Symbols conversion of state chart into an activity diagram.

For each object of HMS, there is a UML state chart diagram, in which each state represents the different activities accomplished by the object, and the transition is used to represent the flow of states in performing activities. All the SCOs of HMS are combined into a single activity diagram (SAD). For example, in Fig. 2, different objects of HMS, like doctor and Diagnosis objects, are connected through a join node. In the state diagram single state, for example, S1 (dr. registered) and S2 (check patient) show various activities in a single state in one object in this doctor object, but in the combining procedure, these several activities are represented by different activity nodes like A1, A2, A3, A4, etc. which are shown in Fig. 5. These activities are combined through join/fork/merge (J, F, and M) edges. Finally, all the activities are connected to the end node (E).

The objects that are interrelated to each other in the case study HMS and implemented using SCD are represented differently in the SAD, as presented in Fig. 4.

3.3. *Conversion of SAD into AG*

This implemented SAD of HMS is then converted into an intermediary graph called Activity Graph (AG), which is shown in Fig. 5. After that; weight is assigned to every edge for computing the traversing cost of all test suites. The graph implemented by this single activity diagram (SAD) is very complex. A complexity-simplified activity graph (SAG) is implemented to eliminate unusual nodes.

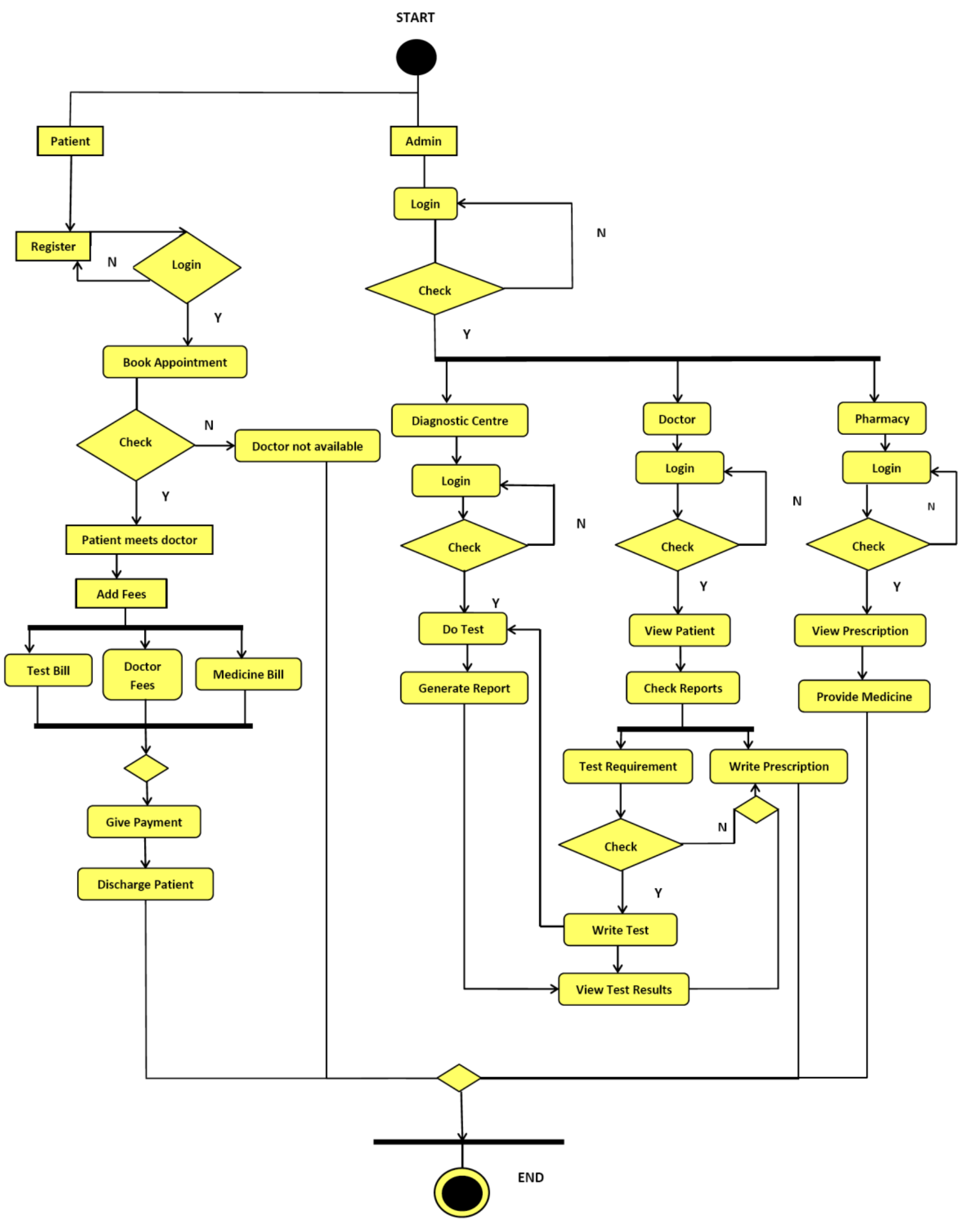


Fig. 4. Combining different objects of HMS into a single activity diagram.

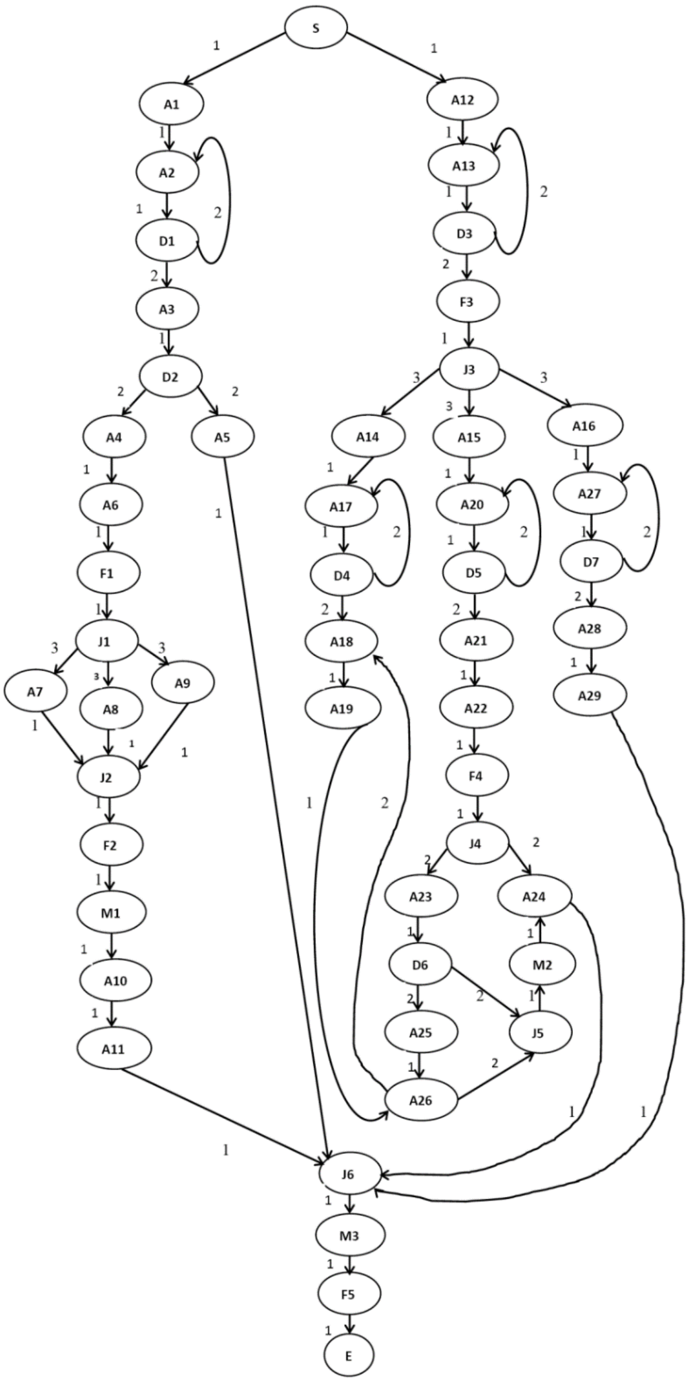


Fig. 5. Conversion of single activity diagram of HMS into activity graph.

3.4. *Simplifying AG to remove the complexity of AG*

The implemented AG using AD remains complicated. The complication looks when several nodes are converted, and each joins/merge/fork transition flow edges into nodes. The simplification is proposed to eliminate these J/M/F nodes, which are used for join, merge, and fork, respectively. The AG, represented in Fig. 5, has 17 paths, but the traversing cost of these paths is high, as shown in Table 1, and implemented test suites are lengthy and time-consuming and take up more space in memory. Hence, the simplification process implements SAG by removing extra nodes that are joined, merged, and forked, and every activity is connected by its prior activities in the end; all activities are connected through the end node. The SAG, represented in Fig. 6, has 12 paths, and the traversing cost of these paths is low, as shown in Table 2 and. Implemented test suites are small, which a less time-consuming process is and takes less space in memory.

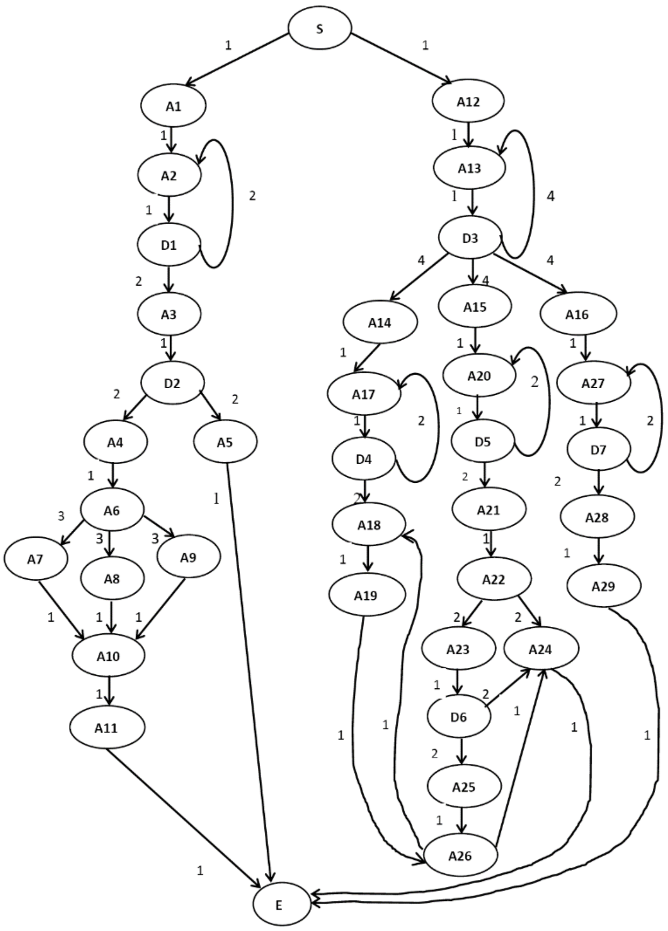
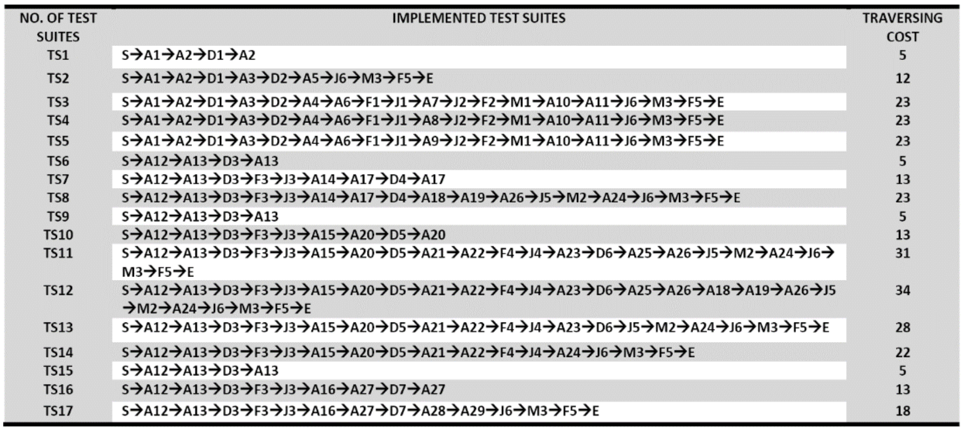


Fig. 6. Simplified activity graph of HMS.

3.5. *Test suites implementation using activity graph (AG) before modification*

AG is delivered as input for implementing test suites. The AG is traversed using the graph traversal algorithm like breadth-first search to implement test suites. Here the number of implemented test suites is 17, and their traversing cost is high, as shown in Table 1.

Table 1. Implemented test suites for HMS before modification in AG.

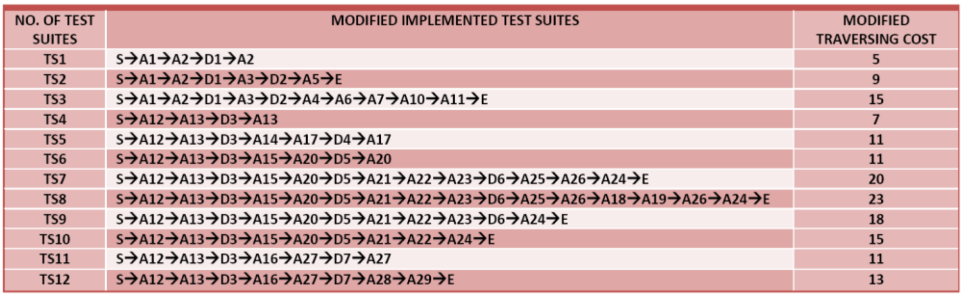


NO. OF TEST SUITES	IMPLEMENTED TEST SUITES	TRAVERSING COST
TS1	S→A1→A2→D1→A2	5
TS2	S→A1→A2→D1→A3→D2→A5→J6→M3→F5→E	12
TS3	S→A1→A2→D1→A3→D2→A4→A6→F1→J1→A7→J2→F2→M1→A10→A11→J6→M3→F5→E	23
TS4	S→A1→A2→D1→A3→D2→A4→A6→F1→J1→A8→J2→F2→M1→A10→A11→J6→M3→F5→E	23
TS5	S→A1→A2→D1→A3→D2→A4→A6→F1→J1→A9→J2→F2→M1→A10→A11→J6→M3→F5→E	23
TS6	S→A12→A13→D3→A13	5
TS7	S→A12→A13→D3→F3→J3→A14→A17→D4→A17	13
TS8	S→A12→A13→D3→F3→J3→A14→A17→D4→A18→A19→A26→J5→M2→A24→J6→M3→F5→E	23
TS9	S→A12→A13→D3→A13	5
TS10	S→A12→A13→D3→F3→J3→A15→A20→D5→A20	13
TS11	S→A12→A13→D3→F3→J3→A15→A20→D5→A21→A22→F4→J4→A23→D6→A25→A26→J5→M2→A24→J6→ M3→F5→E	31
TS12	S→A12→A13→D3→F3→J3→A15→A20→D5→A21→A22→F4→J4→A23→D6→A25→A26→A18→A19→A26→J5 →M2→A24→J6→M3→F5→E	34
TS13	S→A12→A13→D3→F3→J3→A15→A20→D5→A21→A22→F4→J4→A23→D6→J5→M2→A24→J6→M3→F5→E	28
TS14	S→A12→A13→D3→F3→J3→A15→A20→D5→A21→A22→F4→J4→A24→J6→M3→F5→E	22
TS15	S→A12→A13→D3→A13	5
TS16	S→A12→A13→D3→F3→J3→A16→A27→D7→A27	13
TS17	S→A12→A13→D3→F3→J3→A16→A27→D7→A28→A29→J6→M3→F5→E	18

3.6. *Test suites implementation using simplified activity graph (SAG) after modification in AG*

SAG is delivered as input for implementing modified test suites. The SAG is traversed using the graph traversal algorithm like breadth-first search to implement modified test suites. Here the number of implemented test suites is 12, as shown in Table 2, and their traversing cost is less compared to the traversing cost of test suites in Table 1.

Table 2. Final implemented test suites for HMS after modification in AG, i.e., SAG.



NO. OF TEST SUITES	MODIFIED IMPLEMENTED TEST SUITES	MODIFIED TRAVERSING COST
TS1	S→A1→A2→D1→A2	5
TS2	S→A1→A2→D1→A3→D2→A5→E	9
TS3	S→A1→A2→D1→A3→D2→A4→A6→A7→A10→A11→E	15
TS4	S→A12→A13→D3→A13	7
TS5	S→A12→A13→D3→A14→A17→D4→A17	11
TS6	S→A12→A13→D3→A15→A20→D5→A20	11
TS7	S→A12→A13→D3→A15→A20→D5→A21→A22→A23→D6→A25→A26→A24→E	20
TS8	S→A12→A13→D3→A15→A20→D5→A21→A22→A23→D6→A25→A26→A18→A19→A26→A24→E	23
TS9	S→A12→A13→D3→A15→A20→D5→A21→A22→A23→D6→A24→E	18
TS10	S→A12→A13→D3→A15→A20→D5→A21→A22→A24→E	15
TS11	S→A12→A13→D3→A16→A27→D7→A27	11
TS12	S→A12→A13→D3→A16→A27→D7→A28→A29→E	13

4. Conclusion

In order to apply a method of developing test suites, the Hospital Management System (HMS) has been used as a case study in this work. Because it contains many typical scenarios already present in other case studies, the selected case study (HMS) is a good choice for such a methodology. In particular, software testers will find the case study presented in this paper useful. The work in this paper is limited by a few issues. If the order of the state chart objects' occurrences is taken into account, the implemented test suites may vary and also did not consider the possibility of software engineers using several diagrams to implement the same software requirements in this case study.

It is possible to further reduce the size of developed test suites by recycling an appropriate optimization technique, such as Meta-Heuristic algorithms by using this heuristic approach will try to improve more implemented test suites in future.

References

1. D. Trubenbach, S. Muller, and L. Grunske, ACM 6 (2022). https://doi.org/10.1145/3526072.3527523
2. D. Li, W. E. Wong, S. Pan, L. S. Koh, S. Li, and M. Chau, IEEE Access **10**, 85518 (2022). https://doi.org/10.1109/ACCESS.2022.3198694
3. K. Gupta and P. Goyal, AIP Conf. Proc. **2576**, ID 050012 (2022). https://doi.org/10.1063/5.0105807
4. M. Khandai, A. A. Acharya, and D. P. Mohapatra, IEEE **11**, 157 (2011). https://doi.org/10.1109/ICECTECH.2011.5941581
5. P. Wang, X. Hu, N. Qiu, and H. Yang, Recent Adv. Computer Sci. Inform. Eng. **125**, 489 (2012). https://doi.org/10.1007/978-3-642-25789-6_66
6. H. Haga and A. Suehiro, IEEE **12**, 23 (2012). https://doi.org/10.1109/ICCSCE.2012.6487127
7. C. Chang, C. Lu, W. C. Chu, and X. Huang, IEEE **13**, 511 (2013). https://doi.org/10.1109/COMPSACW.2013.116
8. P. Kaur and G. Gupta, Int. J. Comp. Sci. Mol. Comp. **2**, 302 (2013).
9. K. Jena, S. K. Swain, and D. P. Mohapatra, IEEE **14**, 621 (2014). https://doi.org/10.1109/ICICICT.2014.6781352
10. Y. Li and L. Jiang, IEEE **14**, 1067 (2014). https://doi.org/10.1109/ICCSE.2014.6926626
11. V. K. K. S. and S. Mathew, Proc. Comp. Sci. **46**, 859 (2015). https://doi.org/10.1016/j.procs.2015.02.155
12. M. Elallaoui, K. Nafil, and R. Touahni, IEEE **16**, 65 (2015). https://doi.org/10.1109/SITA.2015.7358415
13. J. Park and Y. B. Park, IEEE **16**, 1 (2016). https://doi.org/10.1109/PlatCon.2016.7456784
14. M. Elallaoui, K. Nafil, and R. Touahni, IEEE **16**, 65 (2016). https://doi.org/10.1109/CIST.2016.7804972
15. S. Vemuri, S. Chala, and M. Fathi, IEEE **17**, 1 (2017). https://doi.org/10.1109/CCECE.2017.7946792
16. A. Dutta, S. Godboley, and D.P. Mohapatra, IEEE **17** (2017). https://doi.org/10.1109/INDICON.2017.8487872
17. P. K. Arora and R. Bhatia, Proc. Comp. Sci. **125**, 747 (2018). https://doi.org/10.1016/j.procs.2017.12.096
18. U. Verma, R. K. Rambola, and P. Meshram, Rev. Business Tech. Res. **15**, 102 (2018).
19. T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres, Comp. Sci. Rev. **33**, 98 (2019). https://doi.org/10.1016/j.cosrev.2019.07.001
20. J. Cvetkovic and M. Cvetkovic, Physica A **525**, 1351 (2019). https://doi.org/10.1016/j.physa.2019.03.101
21. M. Panda and S. Dash, IEEE **8**, 179167 (2020). https://doi.org/10.1109/ACCESS.2020.3026911
22. Z. A. Hamza and M. Hammad, IEEE **20**, 1 (2020). https://doi.org/10.1109/IEEECONF51154.2020.9319979

23. S. S. Panigrahi, P. K. Sahoo, B. P. Sahoo, A. Panigrahi, and A. K. Jena, IEEE **21**, 263 (2021). https://doi.org/10.1109/ESCI50559.2021.9396999
24. R. G. Tiwari, A. P. Srivastava, G. Bhardwaj, and V. Kumar, IEEE **21**, 457 (2021). https://doi.org/10.1109/ICIEM51511.2021.9445383
25. I. Alsmadi, IEEE (2010). https://doi.org/10.1109/CCECE.2010.5575262
26. S. Kansomkeat P. Thiket, and J. Offutt, IEEE **10**, 62 (2010). https://doi.org/10.1109/ICSTE.2010.5608913
27. P. N. Boghdady, N. L. Badr, M. A. Hashim, and M. F. Tolba, IEEE **11**, 289 (2011). https://doi.org/10.1109/ICCES.2011.6141058