Prasad and Iyer *RPTEL (2022)17:22*
https://doi.org/10.1186/s41039-022-00192-0

Research and Practice in
Technology Enhanced Learning

RESEARCH Open Access

VeriSIM: A model-based learning pedagogy for fostering software design evaluation skills in computer science undergraduates

Prajish Prasad* and Sridhar Iyer

*Correspondence:
prajish.prasad@gmail.com

Interdisciplinary
Programme in Educational
Technology, Indian Institute
of Technology Bombay,
Mumbai, India

Abstract

Evaluating a software design is an important practice of expert software designers. They spend significant time evaluating their solution, by developing an integrated mental model of the software design and the requirements. However, sufficient emphasis has not been given on teaching and learning of evaluation practices in software design courses, and hence, graduating students find it difficult to critically analyse an existing design and improve upon it. In this paper, we describe a model-based learning pedagogy for teaching–learning of software design evaluation. Model-based learning has been extensively used in science education and entails helping students construct, refine, revise, evaluate, and validate scientific models. We argue that modelling practices in software design evaluation are analogous to these practices. We adapted the model-based learning paradigm and operationalised it into a technology-enhanced learning environment (TELE) for fostering software design evaluation skills in computer science undergraduates. We conducted a research study with 22 undergraduate students to explore how the TELE and its features help students effectively evaluate a given software design. Students attempted a pre-test and post-test which asked them to identify defects in the design. We used the content analysis method to identify categories of defects from student responses in the pre-test and post-test. We also analysed student interaction logs and conducted focus group interviews to identify how features in the TELE contributed towards student learning. Findings from the study showed that students' understanding of evaluation improved, from merely adding new functionalities and requirements, to a process which involved identifying alternate scenarios in the design which violate the given requirements. Students perceived that pedagogical features of the TELE were useful in helping them effectively evaluate software designs. Findings from the study provide evidence for the model-based learning paradigm as an appropriate pedagogy for software design and also opens the space for researchers to investigate model-based learning in other aspects of software design, such as designs of different types and varying complexities.

Keywords: Model-based learning, UML diagrams, Software design evaluation, Scenario modelling and tracing, Content analysis, Thematic analysis

Introduction

In the design phase of the software development cycle, designers come up with the conceptual design of a software system that satisfies the provided requirements. The conceptual design is often modelled using Unified Modelling Language (UML) diagrams (Rumbaugh et al. 2004). UML diagrams specify behaviours and scenarios of the system across various levels of abstraction. For example, the class diagram gives a structural view of the design, by providing the classes and their relationships, along with its data members and functions. The sequence diagram represents the behavioural view of the design, by describing how various objects pass messages with each other for a particular use case. It is essential that these diagrams represent the actual working of the system and satisfy the intended requirements of the system. A failure to do so can lead to inconsistent and incorrect designs, which can then trickle into the code as well. Hence, it is necessary that students develop the ability to accurately reason about a software system design and evaluate it against the given requirements.

When students graduate and enter the software industry, they often work on existing large and complex systems (Begel and Simon 2008a). They usually spend their first several months resolving bugs and writing additional features based on new requirements (Begel and Simon 2008b; Dagenais et al. 2010). When new requirements are provided, they need to have an integrated understanding of the design in order to add features into the design. This requires students to comprehend an already existing design, incorporate the required feature into the design, and evaluate if the design satisfies the intended goals.

Evaluating a software design is an important practice of expert software designers as well. They spend significant time evaluating their solution (Mc Neill et al. 1998). Experts create rich mental models of the design, on which they routinely perform mental simulations (Adelson and Soloway 1986). They also use various reasoning techniques such as simulating scenarios in the given problem (Tang et al. 2010; Guindon 1990), constraints consideration, and trade-off analysis (Guindon 1990), while reasoning about the design.

Considering the importance given to software design evaluation in the software industry, we can conclude that analysing and evaluating a design is an essential skill which needs to be incorporated in the curriculum. However, sufficient emphasis has not been given on teaching and learning of evaluation techniques and practices in a software design course, and hence, graduating students find it difficult to critically analyse an existing design and improve upon it (Brechner 2003). Learning to evaluate a software design is also non-trivial. It involves going beyond understanding software design concepts and requires developing certain cognitive processes in students, such as creating an accurate mental model of the design, simulating various scenarios in the design, and analysing how the design is satisfying/not satisfying the given requirements.

In this paper, we describe a technology-enhanced learning environment (TELE) for fostering software design evaluation skills in computer science undergraduates. We conducted a research study with 22 undergraduate students to explore how the TELE and its features are helping students effectively evaluate a given software design. The theoretical basis of the TELE is based on the model-based learning (MBL) paradigm, which has been used for designing activities to improve scientific modelling skills in students. We

argue that this paradigm and its pedagogical features can be adapted for teaching–learning of software design evaluation.

The paper is organised as follows. In the "Related work" section, we provide a background of software design evaluation, difficulties which students face, and various strategies which expert designers have used to evaluate software designs. Based on this background, in the "Theoretical basis of the pedagogy" section, we argue for the appropriateness of the model-based learning pedagogy for the teaching and learning of software design evaluation. This MBL pedagogy has been operationalised into a TELE, which we describe in the "Technology-enhanced learning environment for software design evaluation" section. We then describe the "Research design" and "Findings" of a study which we conducted to understand the effectiveness of the TELE and its features. We conclude by addressing certain limitations and providing implications and directions for future work.

Related work

Teaching–learning of software design

In typical Software Engineering courses, various Unified Modelling Language (UML) diagrams are taught, where each diagram describes a particular view of the system. While modelling designs, students are required to create models with multiple views, such as the structural view (e.g. using class diagrams) and the behavioural view (e.g. using sequence diagrams).

Various teaching strategies have been employed in teaching–learning of software design. Project-based learning is a common strategy used (Dym et al. 2005). Case studies are given to students who work in groups to come up with the design of the system (Chase et al. 2015). Active learning techniques like role play, pair programming, and peer learning have also been done in the classroom (Hu 2013; Quintana and Grados 2020). Teaching through games in the classroom is also another popular strategy (Jaramillo 2014; Baker et al. 2005). Although technology-enhanced learning environments have been used for software design (Hohmann et al. 1992), and for software design evaluation (Reddy et al. 2021), sufficient research has not been done on the effectiveness of these learning environments.

Various modelling tools have also been used in software design courses. Ciccozzi et al. conducted a survey with 47 instructors, asking them which tools they used to construct UML models in a software design course (Ciccozzi et al. 2018). The findings show that instructors use a wide variety of tools in their courses. The most commonly used ones were the Eclipse Modeling Framework (EMF),[1] Xtext,[2] Papyrus[3] and ATL,[4] Modelio[5], and Visual Paradigm.[6] Most of these tools are plugins in EMF and are open source. However, there have been concerns that these tools are too complex to use (Ciccozzi

[1] https://www.eclipse.org/modeling/emf/.

[2] https://www.eclipse.org/Xtext/.

[3] https://www.eclipse.org/papyrus/.

[4] https://www.eclipse.org/atl/.

[5] https://www.modelio.org/.

[6] https://www.visual-paradigm.com/

et al. 2018), and it is also unclear how these tools support software engineering education (Whittle et al. 2014).

Sufficient emphasis has also not been given to teaching–learning of software design evaluation. Although software engineering courses teach the basic syntax of diagrams in UML, important design evaluation aspects like well-formedness of models, and semantics of designs are not given sufficient importance (Westphal 2019). For example, while evaluating software designs, apart from checking syntactic issues in the design, students need to focus on developing an integrated understanding of the UML diagrams in order to build designs which are comprehensive and consistent.

Hence, we see that there are different perspectives while evaluating software designs. We explain these perspectives in the next subsection.

Software design evaluation

Evaluating a software design involves assessing the quality of such design models from different perspectives. Lindland et al. characterise software design evaluation based on checking the design for their syntactic, semantic, and pragmatic quality (Lindland et al. 1994). *Syntactic quality* refers to how well the models adhere to the rules of the language, i.e. the rules and syntax of UML. Hence, issues in the syntactic quality can be due to mistakes corresponding to the syntax and naming conventions in UML diagrams *(e.g. Are notations in class and sequence diagrams adhering to the standard rules and syntax of UML). Semantic quality* refers to how faithfully the modelled system is represented, i.e. if there is an accurate mapping between the model and the requirements. Issues in semantic quality can occur when the model lacks something that is present in the requirements, or the model includes something that is not present in the requirements. *Pragmatic quality* refers to how well a given design can be interpreted by different stakeholders, such as other team members, clients etc. There are several characteristics with regard to pragmatic quality, such as maintainability (how easy is the system to maintain), reusability (how can components of the design be used in building the model of another system), and complexity (how complex the system is to understand) (Nelson and Piattini 2012).

In this paper, we have focussed on software design evaluation from the *semantic quality* perspective. Detecting semantic deficiencies in designs is non-trivial, as it requires students to critically analyse the design and requirements, understand the relationship between different diagrams, develop an integrated model of the design, and evaluate the design against the given requirements. We also see that students face certain difficulties in effectively evaluating software designs for semantic quality. Prasad et al. conducted a study to understand how students evaluated a given design for semantic defects, and the difficulties they faced (Prasad and Iyer 2020a). The study found that although some students were able to identify semantic deficiencies in the design, many students added new functionalities and modified existing functionalities unrelated to the given requirements. Studies have also shown that students face difficulties in developing a consistent understanding among different diagrams (Sien 2011; Stikkolorum et al. 2016) and struggle to understand how the overall system specifications actually work based on these diagrams (Burgueño et al. 2018). They have difficulties in identifying the purpose and

relationship among these diagrams and tend to view the diagrams as existing in isolation (Stikkolorum et al. 2016).

Hence, we have scoped the work in this paper to develop a pedagogy which enables students to effectively evaluate software designs for *semantic deficiencies*. In the next subsection, we look at strategies expert software designers use for software design evaluation.

Strategies for software design evaluation

Strategies for evaluating software designs have been explored in the context of inspecting UML design diagrams in order to identify defects. Findings from our literature review show that experts use reading techniques like fast switching (Hungerford et al. 2004), horizontal and vertical reading (Travassos et al. 1999), and perceptual and conceptual processes (Kim et al. 2000) while evaluating design diagrams. We explain these strategies in detail below.

Hungerford et al. conducted a study with 12 experienced developers who were asked to perform individual reviews on a software design (Hungerford et al. 2004). The results indicate that reading techniques that rapidly switch between the two design diagrams are the most effective. The authors claim that these "fast switching" strategies help in building relationships between diagrams and create a better mental model.

Travassos et al. formulated a set of reading techniques, termed as traceability-based reading, which help students integrate information across diagrams (horizontal reading) and also between diagrams and textual requirements (vertical reading) (Travassos et al. 1999). In horizontal reading, students focus their attention on a design diagram (such as class diagram) and inspect it with respect to another diagram (such as sequence diagram). This ensures that they are building an integrated and correct mental model of the design. In vertical reading, the design diagrams are read with respect to the textual requirements and use cases. For example, a requirement is read and the specific sequence diagram which realises this requirement is analysed. If there are certain entities missing in the sequence diagram based on the given requirement, it can be added. Using these horizontal and vertical reading techniques, students were asked to report defects in the given design diagrams. Results from their study show that the techniques did lead to defects being detected.

Kim et al. conducted a study to explore the cognitive processes which one uses to understand a system represented by multiple diagrams (Kim et al. 2000). The authors claim that reasoning with multiple design diagrams involves performing effective perceptual and conceptual processes. Perceptual processes while reasoning with design diagrams involve linking relevant information from different diagrams. A conceptual process is used to generate, refine, and validate hypotheses based on the provided design diagrams. Based on the analysis of participants interacting with multiple diagrams, successful participants' perceptual integration processes involved making several round-trip transitions, enabling them to relate information from different diagrams. By doing so, they were able to develop an integrated mental model of the design. Conceptual integration processes involved creating and refining several hypotheses while inspecting several diagrams. Based on these hypotheses, participants refined and adapted their mental model of the design as well.

Based on these studies, we can infer that inspecting design diagrams involves focusing on relevant information in these diagrams in order to create an effective mental model of the design (Winn 1994). A mental model is a "mental structure that represents some aspect of one's environment" (Sorva 2013). It is an abstract representation of a system that enables us to describe the underlying mechanisms of systems, answer questions about the system, as well as predict future system states (Schumacher and Czerwinski 1992). The studies mentioned above show that effective software design evaluation happens when designers integrate information between design diagrams and build an accurate mental model of the design (Hungerford et al. 2004; Travassos et al. 1999). They generate several scenarios based on the requirements and the design (Tang et al. 2010; Guindon 1990) and simulate how the system carries out specific scenarios to fulfil the stated requirements (Adelson and Soloway 1986; Zannier et al. 2007). They mentally simulate the control flow and data flow of various scenarios in the design (Guindon 1990). Control flow refers to what methods are called, which components call which functions, how a function is reached, etc. Data flow refers to the change in data variable values on the basis of method calls. In other words, they imagine all changes that occur in different diagrams for various scenarios and examine which scenarios and conditions violate the given requirement.

Summary

To summarise, we see that sufficient emphasis has not been given in designing pedagogies for software design evaluation, especially for the design's semantic quality. Findings from studies show that effective evaluation of a software design involves developing an integrated mental model of the design, and simulating various scenarios in these mental models in order to identify defects. However, students face difficulties in developing such an integrated understanding, and in identifying relevant scenarios in the design.

This background forms the basis for the model-based learning pedagogy for software design evaluation, which we explain in the next section.

Theoretical basis of the pedagogy

In the proposed pedagogy, we aim to scaffold students to identify relevant scenarios and construct an effective model of these scenarios. As they identify and model various scenarios in the design, they will be able to uncover defects in the design. In order to support students to identify and construct models of relevant scenarios in the design, we draw the theoretical basis of our pedagogy from the model-based learning paradigm in science education. We argue for its appropriateness in the context of software design and adapt effective affordances and pedagogical features to inform the design of the pedagogy for software design evaluation.

Model-based learning in science education

Modelling is a central practice in the discipline of science (Papaevripidou and Zacharia 2015). When scientists observe a phenomena, they construct models which try to explain the process at a sufficient level of abstraction. The model serves as a portrayal of a scientist's current understanding of the phenomena. The model is then tested by observations in the real world and refined based on these observations (Hestenes 2010).

Since the modelling process is an important practice of scientists, it is essential that students are explicitly taught this process. Scientists have emphasised the importance of integrating this modelling process into the teaching–learning of science (Hestenes 1987, 1992). An important goal of science education is to enable students to develop powerful models for making sense of their daily experiences of biological, physical, and chemical phenomena (Clement 2000). A prominent paradigm towards this goal is model-based learning. Simply put, model-based learning involves using modelling for learning purposes. It is defined as "*a dynamic, recursive process of learning by constructing mental models of the phenomenon under study. It involves the formation, testing, and subsequent reinforcement, revision, or rejection of those mental models*" (Buckley et al. 2010). Model-based learning provides students opportunities to interact with physical, representational, or computer models which describes the working of a scientific phenomenon. As students engage in constructing scientific models, they mimic modelling practices of scientists and also develop conceptual knowledge of the phenomena.

Modelling a phenomenon requires knowledge of specific modelling practices (Papaevripidou and Zacharia 2015). It entails constructing, refining, revising, evaluating, and validating scientific models. Learning to model a phenomenon involves investigating it, collecting evidence, and bringing one's observations and experiences from the physical world. These conceptions and domain-specific knowledge result in the construction of an initial working model *(model formulation)*. The initial model is modified by restructuring and tuning various elements in the model. During model revisions, several intermediate models are generated. Students go back and forth between the intermediate models and compare it with the phenomenon or system *(model comparison)*. These comparisons are done by performing mental simulation on these models and validating whether it predicts the corresponding phenomenon accurately *(model evaluation)*. *Hence, model formulation, model comparison, and model evaluation are essential practices required during model-based learning* (Papaevripidou and Zacharia 2015).

Adapting model-based learning for software design evaluation

The term "model" has been used in science education to refer to physical or biological systems. Learning of scientific modelling requires students to have knowledge of components (objects, processes and entities) of the system and interactions between these components. Analogously, software design models also comprise components, such as objects, variables, and methods. These components interact with each other to describe the system behaviour. For example, an object's member function can call a function of another object, resulting in changes in variable values in the system. Students require knowledge of these components and how they interact with each other to understand the functioning of a software system. As students develop their understanding of the components and their interactions, they are able to effectively model a software system.

The practices involved in modelling scientific phenomenon can be adapted to software design as well. A common characteristic of modelling activities is that these models are constructed, refined, and evaluated to describe the phenomenon under observation. The central tenet of model-based learning is that the process of constructing and manipulating these mental models causes a deeper and integrated understanding of scientific concepts required to understand the phenomenon. We argue that similar mental modelling

processes occur while evaluating software designs as well. Experts create an initial mental model of the given software design and the requirements *(model formulation)*. They then identify various scenarios in their model, compare it with the design *(model comparison and evaluation)*, and validate their models with the given design, by examining if there are any defects or deficiencies *(model validation)*.

Hence, we argue that the model-based learning paradigm can serve as the basis for teaching–learning of evaluating a given software design. We describe the model-based learning pedagogy in the next subsection.

The model-based learning pedagogy for software design evaluation

We propose the VeriSIM pedagogy for the teaching and learning of software design evaluation. VeriSIM stands for "**Veri**fying designs by **SIM**ulating scenarios". The key idea of the VeriSIM pedagogy is that *"scaffolding students to identify and construct models of relevant scenarios in the design can lead to effective evaluation of the design diagrams against the given requirements."*

The VeriSIM pedagogy has two key phases. Phase 1 incorporates the design tracing strategy, which trains students to construct a model of a given scenario. In this phase, the model is similar to a state diagram. This model enables learners to simulate the control flow and data flow of the execution of a given scenario.

Phase 2 incorporates the scenario branching strategy, which trains students to identify various scenarios for each requirement. In this phase, the model is a scenario branching tree. As learners construct the tree and traverse through the tree, they are able to identify relevant scenarios for each requirement.

Hence, in the VeriSIM pedagogy, the state diagram and the scenario branching tree are the models which learners construct to perform effective software design evaluation. We explain these models in detail below.

Phase 1: Modelling scenarios using design tracing

Design tracing is an adaptation of the tracing strategy used in programming. Program tracing is the process of emulating how a computer executes a program (Fitzgerald et al. 2005). While tracing, programmers visualise how control flows and data values change during the execution of a program. In design tracing, students trace the control flow and data flow across different diagrams for a given scenario of system execution. Hence, in Phase 1, the model which students construct is a model of the scenario execution, which is similar to a state diagram. The values in the states correspond to the values of relevant variables, and the transitions correspond to different parts of the scenario.

We illustrate the design tracing strategy with an example. Consider the case of an ATM system design. A requirement for this design is *"When the user enters the ATM and inputs the correct PIN, the user can withdraw money from his/her account. If the balance is less than Rs. 1000, withdrawal is denied"*. For this requirement, a probable scenario can be that the *"User withdraws an amount greater than balance"*. Using design tracing, students identify different data attributes from the class diagram, which are added to the states, as shown in Fig. 1. They identify the purpose of different parts of the sequence diagram, and these form the transitions of the state diagram. Based on the transitions, the values in the states keep changing. For example, in Fig. 1, the final transition in the

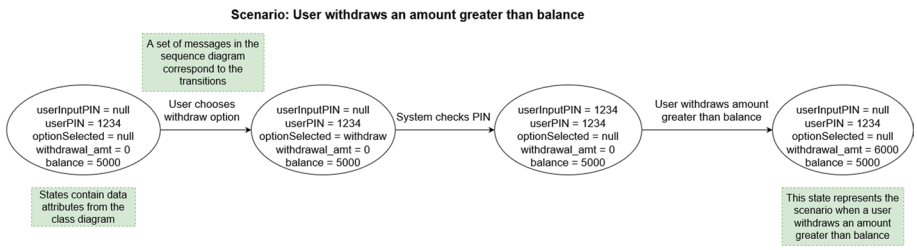


Fig. 1 The model state diagram for a given scenario generated using the design tracing strategy

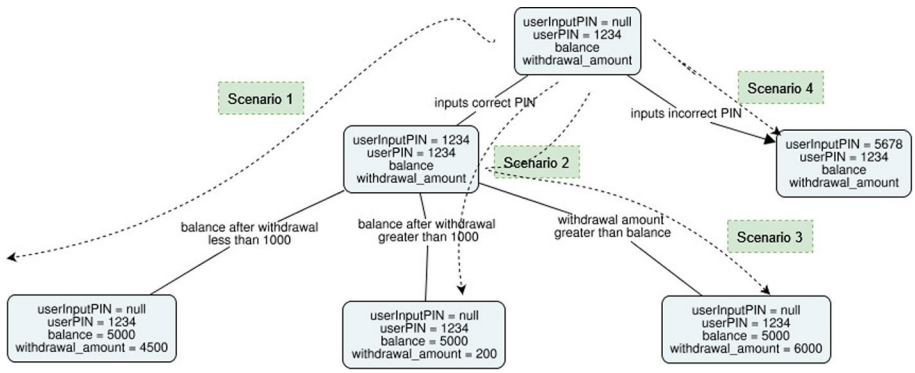


Fig. 2 The scenario branching tree for a given requirement

state diagram results in a change in the value of *withdrawal_amount*. The final state corresponds to when the user enters the withdrawal amount greater than the balance.

As students construct the state diagram, they simulate the execution of the given scenario and also simulate the change in control flow and data flow. Hence, constructing such state diagram models for other scenarios in the design scaffolds learners to simulate the execution of these scenarios in the design.

Phase 2: Identifying scenarios using scenario branching

Once students are trained to construct models of an already given scenario, the scenario branching strategy scaffolds them to identify different scenarios for each requirement in the design. Scenario branching is adapted from cognitive mapping strategies which have been widely used in requirement analysis (Montazemi and Conrath 1986). In scenario branching, students analyse each requirement and break it down into units called sub-goals. For each sub-goal, they identify relevant variables and different possibilities for these variables. They represent these different possibilities for each sub-goal in a visual tree-like representation, known as a "scenario branching tree". The tree captures the identified sub-goals and different values for each of these variables. After constructing the tree for a given requirement, students identify scenarios by traversing from the root to a leaf in the tree. Hence in this phase, the scenario branching tree serves as the model which enables learners to identify relevant scenarios for each requirement.

We illustrate the scenario branching strategy with the help of an example. Consider the minimum balance requirement *"When the user enters the ATM and inputs the*

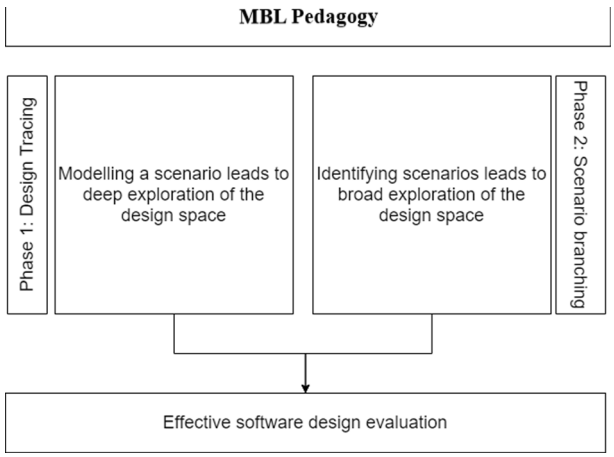


Fig. 3 The model-based learning (MBL) pedagogy for software design evaluation

correct PIN, the user can withdraw money from his/her account. If the balance is less than Rs.1000, withdrawal is denied". The sub-goals for this requirement are: *"User inputs correct PIN", "Withdrawal of amount from account"*. Based on these sub-goals, students are scaffolded to construct the scenario branching tree, as shown in Fig. 2. The scenarios which can be identified by traversing the tree are: *"(1) User inputs correct PIN, and balance after withdrawal is less than 1000, (2) User inputs correct PIN, and balance after withdrawal is greater than 1000, (3) User inputs correct PIN, and withdrawal amount is greater than balance, (4) User inputs the incorrect PIN"*. Students can then examine whether these scenarios are present in the design. Hence, the scenario branching strategy enables students to identify scenarios which do not satisfy the given requirements.

Summary of the VeriSIM pedagogy

We hypothesise that as students identify scenarios using the scenario branching strategy and construct models of specific scenarios by tracing the control flow and data flow using the design tracing strategy, they will be able to evaluate the given software design better. First, by identifying different scenarios for each requirement, students broadly explore the design solution space. Second, by tracing each of these scenarios, they are forced to think deeply about the flow of data and events for each scenario. Hence, both the broad exploration of the design by identifying scenarios and simulating the data and control flow for each scenario can help in an improved understanding of the design and enable students to effectively evaluate the design against the given requirements. A summary of the MBL pedagogy is shown in Fig. 3.

Technology-enhanced learning environment for software design evaluation

Studies have shown that technology support and scaffolding for model-based learning are essential for student learning (Fretz et al. 2002). Students provided with scaffolds (such as prompts, hints, and visualisations) as they interact with models have shown to perform better than those who were not provided any scaffolds (Buckley et al. 2010; Azevedo et al. 2011). Hence, we have operationalised the VeriSIM pedagogy into a technology-enhanced learning environment (TELE).

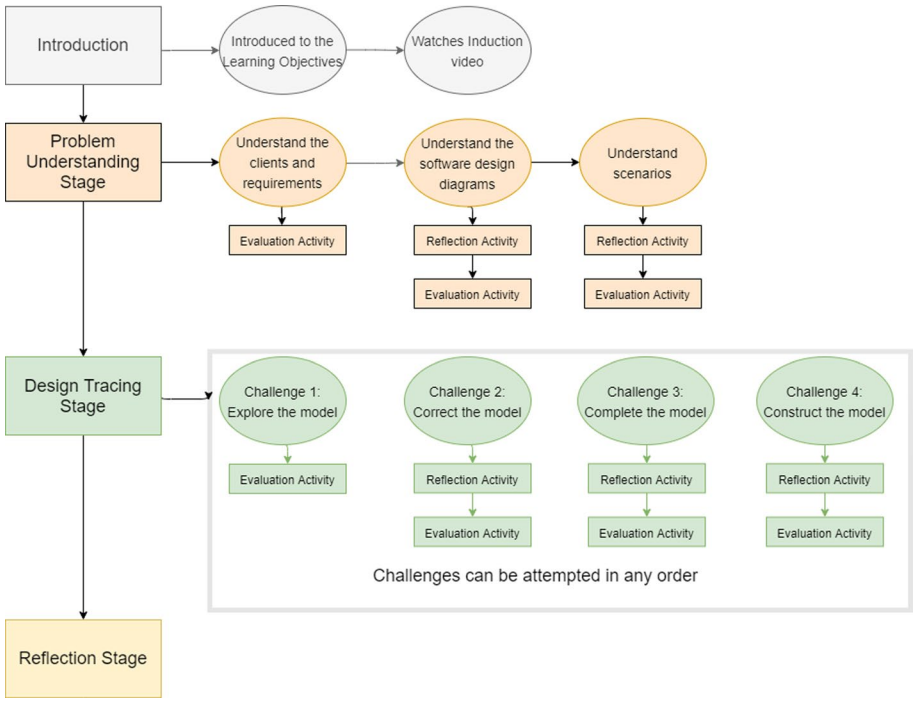


Fig. 4 Overview of learning activities in Module 1 of the TELE

The TELE has two modules corresponding to the two phases of the VeriSIM pedagogy. In the first module, students are introduced to the design tracing strategy and go through activities which enable them to trace the given scenarios in the design. The first module of the TELE is a self-learning module. All information and scaffolds required for students to perform design tracing are present in the learning environment itself.

In the second module, learners are introduced to the scenario branching strategy. Learners generate alternate scenarios in a given design using a mapping (Cmap) tool. This module is partly guided by the instructor. Students are provided with a worksheet which contains the requirements and design diagrams for a given problem. The instructor explains the scenario branching strategy and facilitates alternate scenarios creation using the Cmap tool.

Module 1: Design tracing module

Module 1 is a web-based learning environment that trains learners to apply the design tracing strategy. In Module 1, learners go through different stages that contain challenges. As learners attempt these challenges, they gain points and acquire skills. An overview of the stages and challenges in Module 1 is shown in Fig. 4. The three stages are: Problem Understanding Stage, Design Tracing Stage, and Reflection Stage. In the Problem Understanding Stage, learners are introduced to the requirements and the software design of an "Automated Door Locking System". In the Design Tracing Stage, learners are introduced to the design tracing strategy. This stage comprises four challenges in increasing order of complexity. In each challenge, a different scenario of the same design is provided. Learners are free to attempt the challenges in any order. In all the challenges

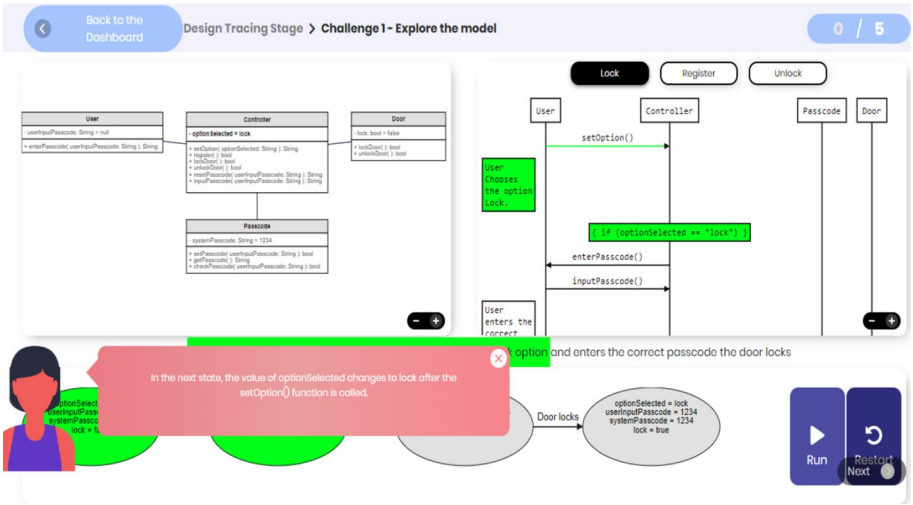


Fig. 5 The agent explaining the state diagram execution for Challenge 1

in the Design Tracing Stage, the objective of the learner is to trace the scenario using the state diagram. The state diagram has to match the expert model for the learner to successfully complete the challenge. Each state in the state diagram is compared with the corresponding state in the expert model, and appropriate feedback is provided by the system. Finally, in the Reflection Stage, learners reflect on their overall learning and how design tracing will be useful for them in the future. Summary videos of the stages in the TELE can be found at this link - https://prajishprasad.github.io/verisim.html.

Each stage comprises challenges, which are followed by evaluation and/or reflection activities (Ge and Land 2004). The reflection activities make students reflect on what they have done and learnt in a challenge. The evaluation activities test them on what they learnt in the challenge. A pedagogical agent serves as a guide to the learner and helps understand the different features. The agent also provides the goal of each challenge to the learner and provides appropriate feedback when they encounter certain errors in the challenges.

The key stage in Module 1 is the Design Tracing Stage. We now explain each challenge in the Design Tracing Stage.

Challenge 1: Explore the model

In this challenge, learners are introduced to the state diagram for the first time. They are presented with the correct state diagram which describes the given scenario. The agent describes the design tracing strategy and the model, i.e. the state diagram, and its different parts, such as the states and the transitions. The agent then suggests learners to click the "Run" button to view the execution of the model. For each click of the "Run", the appropriate state is highlighted in green, and appropriate messages are provided by the agent. The agent describes the changes in the design diagrams when control transfers to a particular state, as shown in Fig. 5. After the execution reaches the final state, learners can proceed to an evaluation activity. The evaluation activity contains questions which

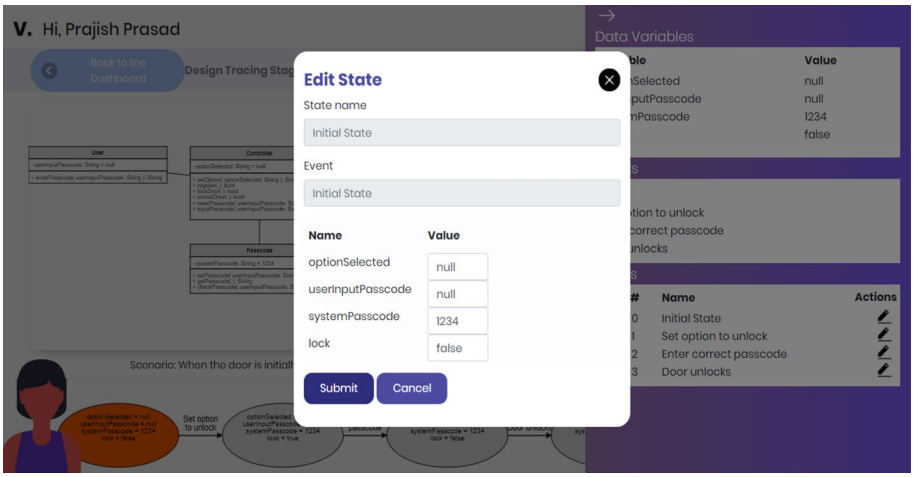


Fig. 6 In Challenge 2, learners can change values of variables in appropriate states

test learners' understanding of the state diagram (Example question:. *The state diagram contains relevant data variables from the ____ diagram*). After completing the evaluation activity, learners can proceed to Challenge 2.

Challenge 2: Correct the model

In this challenge, learners are presented with another scenario and an incorrect state diagram which models this scenario. The agent explains the objective of this challenge, i.e. learners are expected to fix errors in the given state diagram. They can click the "Run" button at any time to validate the model. If a state contains an error, and learners click on "Run", the agent indicates an error in the state by highlighting that state in red and asks learners to trace the state change of variables in the highlighted state to identify the error. To modify the state, learners can click on the "Edit" button, which opens the model attribute space, where they can edit the appropriate state by changing the appropriate values (see Fig. 6). For example, for the first state in Challenge 2, learners are required to change the value of "lock" from "false" to "true", since the door is initially at the locked state. Learners are free to edit states and check the execution of the model at any time during the challenge.

After learners correct all the states in the state diagram, they can click on "Run" to validate their model. They then move onto a reflection activity, which asks them to summarise what they learnt and what they found difficult in Challenge 2. After answering the reflection question, they then attempt an evaluation activity. In the evaluation activity, they are provided with an incorrect state diagram. They attempt multiple-choice questions which ask them to identify incorrect states and also choose the appropriate correct states. After attempting this evaluation activity, learners move on to Challenge 3.

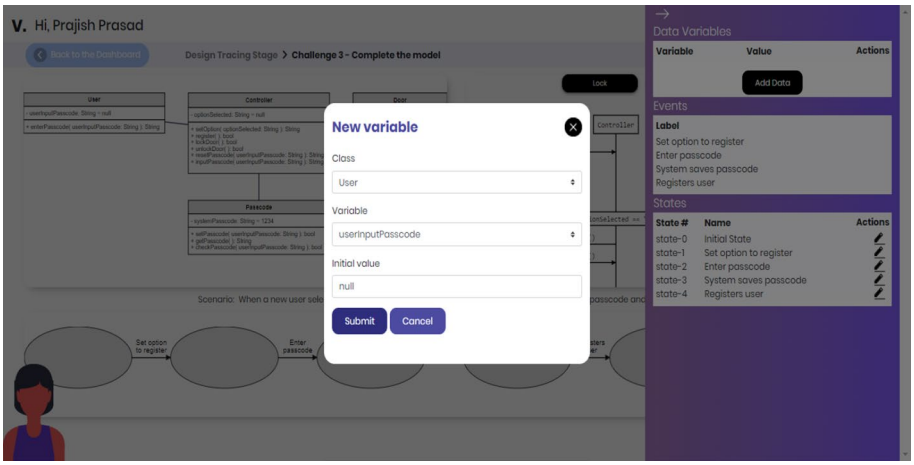


Fig. 7 In Challenge 3, learners add relevant data variables to the state diagram

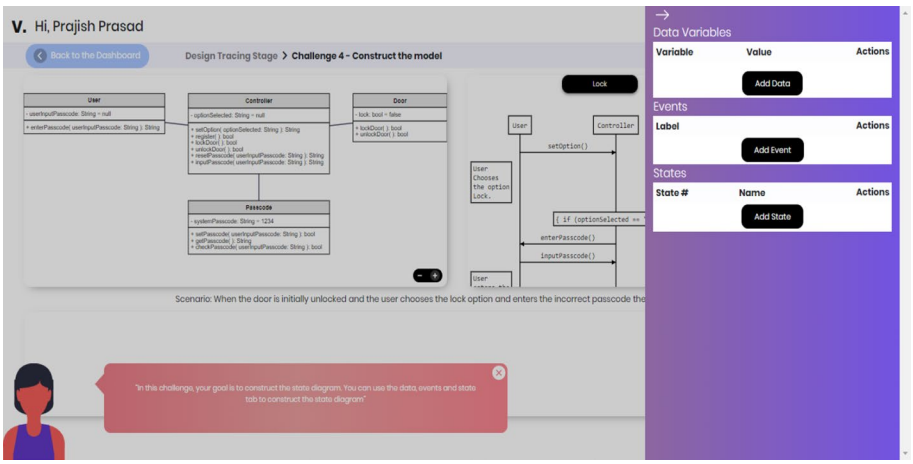


Fig. 8 In Challenge 4, learners construct the entire state diagram from scratch

Challenge 3: Complete the model

In Challenge 3, learners are provided with the state diagram containing relevant events, but the data variables and their values are missing. Learners can use the "Edit" feature to add data variables in the data tab (see Fig. 7) and edit appropriate states to reflect the change in values of variables in these states. Similar to previous challenges, they can execute the state diagram at any time to get feedback on their model. After completing the state diagram with appropriate data variables and values, they move on to reflection and evaluation activities, similar to the ones after Challenge 2.

Challenge 4: Construct the model

When learners start challenge 4, they are provided with an empty model. They have to add relevant data variables, events, and states to model the given scenario (see Fig. 8).

(a) Example: Requirement 1:

A user with a valid account can register his/her ATM and set a PIN if he/she has not set a PIN yet. The PIN should be of length 4 and should contain only numbers.

Step 1: Identifying sub-goals in the requirement

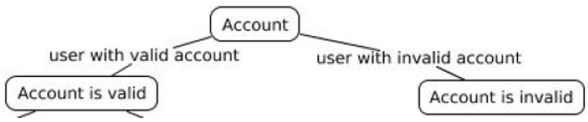
Identify the sub-goals in the requirement. For example, in requirement 1, the sub-goals are:

a. User with valid account
b. Sets a PIN if a PIN hasn't been set yet
c. PIN should be of length 4 and should contain only numbers

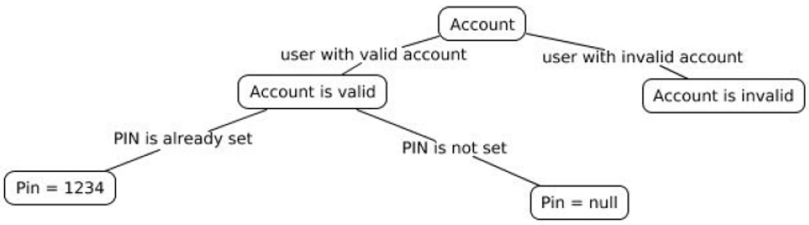
(b) Step 2: For each sub-goal, identify relevant variables. Identify different possibilities of these variables. Use the concept map to come up with alternate scenarios

Consider the example for requirement 1

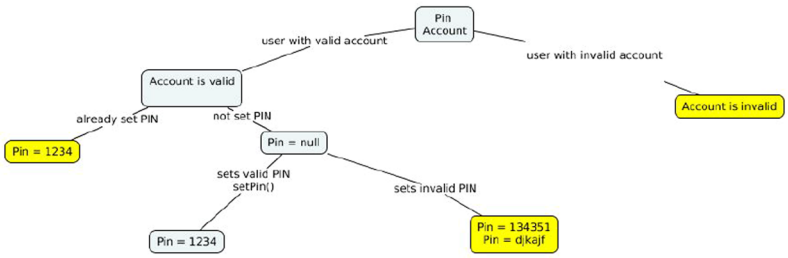
a. User with valid account - The account object can be either set or not set. Hence, two scenarios are possible, either the user has a valid account or an invalid account. The linking phrases indicate the different scenarios. The values inside the node indicate objects and data members from the class diagram



(c) b. Sets a PIN - Based on the value of Pin, two scenarios are possible, either the PIN is already set or it is not set



(d) c. Finally, the entered Pin can be valid or invalid - setPin() is called to set the correct Pin.



In the above scenario tree, start from the root node and traverse all the way down. Each path corresponds to a scenario.

Scenario 1: User with a valid account has already set a Pin
Scenario 2: User with a valid account has not set a Pin and sets a valid Pin
Scenario 3: User with a valid account has not set a Pin and sets an invalid Pin
Scenario 4: User has an invalid account

Which of the following scenarios are not described in the design diagrams? - Scenario 1, Scenario 3 and Scenario 4

Hence, these are defects which need to be rectified in the diagrams

Fig. 9 Steps outlined in the scenario branching worksheet

Module 2: Scenario branching module

Module 2 incorporates the scenario branching strategy and is facilitated by an instructor. Learners are provided with the requirements and design diagrams (class diagram and three sequence diagrams) of an ATM system in a paper worksheet. The scenario branching worksheet is attached as an additional file (Additional file 1). The worksheet is divided into two parts. There are 4 steps in Part 1 of the worksheet. In the first step, the worksheet describes the relevant sub-goals for the first requirement, as shown in Fig. 9a. In the next step, the worksheet describes how to progressively construct the scenario branching tree. The worksheet describes how to construct the intermediate scenario tree for the first sub-goal (*User with a valid account*). The intermediate scenario tree has two branches as shown in Fig. 9b. For the next sub-goal (*Sets a PIN if a PIN has not been set yet*), two more branches are added to the scenario tree, as shown in Fig. 9c. For the last sub-goal (*PIN should be of length 4 and should contain only numbers*), two more branches are added, as shown in Fig. 9d.

Learners follow these steps and construct the scenario branching tree using the Cmap tool. The worksheet then explains how to identify scenarios from the scenario tree and lists down all the scenarios. It then describes which scenarios have not been described in the design diagrams. The steps outlined in Part 1 are explained by the instructor. In Part 2, learners need to construct the scenario tree for the remaining three requirements and identify scenarios which do not satisfy the requirements.

To facilitate construction of the scenario branching tree, learners use the Cmap[7] concept mapping tool. The Cmap tool enables users to create and share knowledge models represented as concept maps. The tool has affordances for adding nodes, and establishing links between nodes. In the context of scenario branching, each node denotes a set of values of the identified variables. The links denote different possible scenarios for the sub-goals. Using the Cmap tool, learners can add, modify, and delete nodes and links. Hence, the Cmap tool helps learners incrementally construct the scenario tree for a given requirement. It also provides a means for learners to explore the design space by thinking of different possible scenarios in the design based on the requirements.

Features in the TELE facilitating model-based learning

Model progression of activities

The order of challenges in the "Design Tracing Stage" of Module 1 is based on the model progression of activities (Mulder et al. 2011). In model progression, learners are provided models which vary in dimensions such as their perspective, degree of elaboration, and their order (White and Frederiksen 1990). When learners are provided opportunities for successive refinements of their model, it helps them progressively understand and develop models of the given phenomenon. Various strategies support this idea of model progression. These include strategies such as learners exploring a full working model [*prior exploration* (Kopainsky et al. 2015; Kopainsky and Alessi 2015)], receiving support in the form of a partial model which outlined the basic structure of a system

[7] https://cmap.ihmc.us/cmaptools/

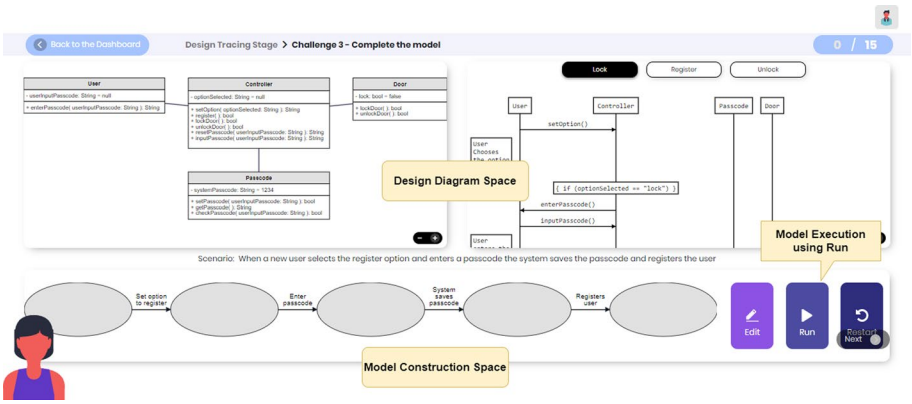


Fig. 10 Various spaces in the Design Tracing Stage challenges

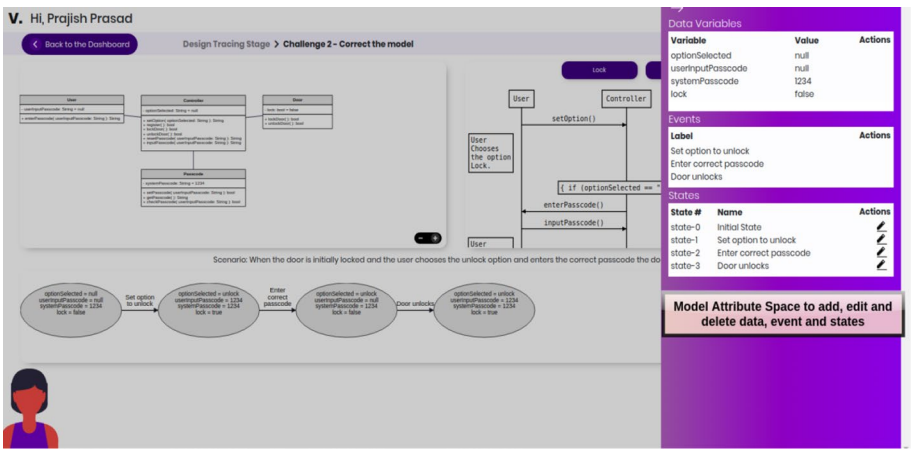


Fig. 11 The model attribute space in the Design Tracing Stage challenges

[*partially worked-out models* (Mulder et al. 2016)], and learners working with a model which contained errors [*learning from erroneous models* (Wijnen et al. 2015)].

We have appropriately adapted these activities to the context of modelling scenarios in a given design. In the first challenge of the "Design Tracing Stage", learners observe the run of the state diagram, i.e. the execution of trace of the scenario. In the second challenge, there is an error in the data flow, which learners have to correct. In the third challenge, the entire data flow has to be traced by learners. In the fourth challenge, control flow and data flow have to be traced. Each of these modelling activities, such as prior exploration, learning from partial and erroneous models have been shown to be beneficial to learners in the context of modelling in science.

Affordances for model refinement

In the "Design Tracing Stage" challenges of Module 1, learners go through four challenges which progressively scaffold them to trace a given scenario by constructing a state diagram. Figure 10 shows the interface for Challenge 3 in the Design Tracing Stage. (The

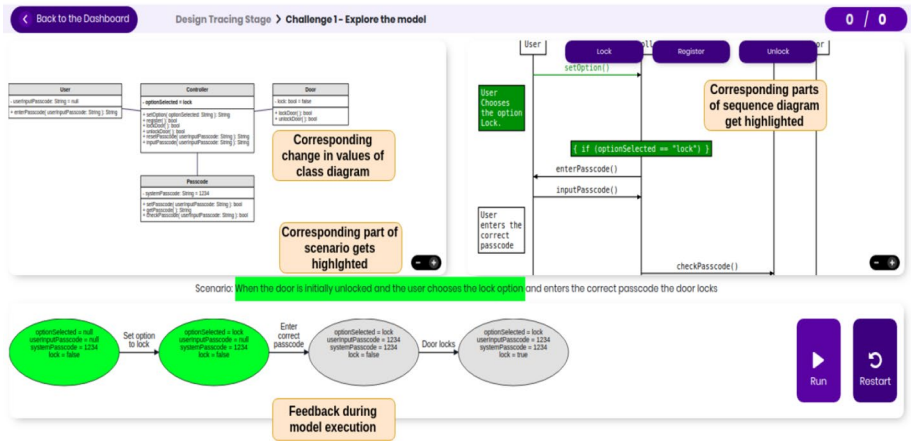


Fig. 12 Affordance of the run feature in the Design Tracing Stage challenges

interface is similar for all four challenges in the Design Tracing Stage.) Learners interact with the model (i.e. the state diagram) in the *model construction space*, as shown in the figure. Learners can construct and revise the model by clicking the "Edit" button on the interface. On clicking the "Edit" button, learners are provided with the *model attribute space*, as shown in Fig. 11. As seen in this figure, learners can add, edit, and delete data variables, events, or states. These actions are reflected in the model state diagram. For example, when a new data variable is added with an initial value, the data variable and value get added in all the states in the state diagram. Learners can then modify the values in each state of the state diagram by editing the respective states.

In Module 2, the affordances of the Cmap tool also support model creation and refinement. Learners *construct a model* of the given requirements by constructing a scenario branching tree. They can edit the nodes and links of the tree, thereby facilitating *model revision*. They perform *model comparison and evaluation* when the identified scenarios are compared and evaluated with the appropriate design diagrams, and missing scenarios in the design are identified. Hence, constructing the scenario branching tree using the Cmap tool assists learners in constructing an accurate mental model of the requirements and enables them to map these requirements to the given design.

Affordances for model execution visualisation

Module 1 of the TELE provides the affordance to visualise the state-wise execution of the model, by clicking on the "Run" button in the interface. Each click of the "run" button results in the system checking the correctness of a particular state in a linear order. When the learner clicks on "Run" after constructing the state diagram, the system matches the first state with the expert model. If correct, the state is highlighted in green and the system proceeds in checking the next state. If a state contains an error, the execution stops, the state is highlighted in red and the agent provides a feedback message indicating an error. Whenever a learner clicks on "Run" the next time, the execution starts from the first state.

Based on each click of the "Run" button, appropriate highlights are made in the class and sequence diagram as shown in Fig. 12. With the help of these highlights, learners are

Table 1 Pedagogical agent message types in Module 1 of the TELE

Message type	Example
Explaining parts of the interface	There are three stages comprising different challenges. Complete each stage to unlock challenges in the next stage. Earn up to 100 points as you attempt these challenges. You also gain skills in design tracing as you successfully complete these challenges
Explaining concepts	In design tracing, we trace the sequence of data and function changes across different diagrams for a given scenario of system execution
Introducing goals of a challenge	In this challenge, your goal is to construct the state diagram. You can use the data, events and state tab to construct the state diagram
Providing feedback	It looks like there is an error in the highlighted state. Trace the state change of variables and try to identify the error

able to visualise the relevant changes in the class and sequence diagrams in a particular state. A transition to a new state triggers a set of events in the sequence diagram. The visualisation shows the corresponding changes in the class diagram (change of values of variables) for each event/message in the sequence diagram. Learners can replay this execution step, review previous execution steps, and move forward. As they observe these visualisations and corresponding changes in the design diagrams, they are able to visualise the state change of different variables at a given state and also understand the relationship between the class and sequence diagram. Thus, the "Run" feature enables learners to validate their model and refine the model based on the feedback provided.

Feedback and diagnosis of learner's errors

Providing appropriate feedback is essential in a technology-enhanced learning environment, so as to enable learners to proceed in their tasks.

In the Design Tracing Stage of Module 1, the objective of the learner is to trace the scenario using the state diagram. The state diagram has to match the expert model for the learner to successfully complete the challenge. Each state in the state diagram is compared with the corresponding state in the expert model, and feedback is provided by the system. As the learner uses the "Run" feature to validate their model, the agent provides appropriate feedback regarding the correctness of each state in the state diagram. The feedback messages do not directly tell learners what to do, but directs their attention to what can be done to resolve the error. A summary of feedback messages provided in Module 1 is shown in Table 1.

In Module 2, learners are introduced to the scenario branching activity worksheet, which is facilitated by the instructor. The instructor takes learners through Part 1 of the worksheet (Module 2: Scenario branching) and scaffolds them to construct a scenario branching tree for the first requirement. After this, learners have to create a scenario branching tree for the remaining three requirements. During this time, the instructor is available to provide appropriate feedback for difficulties that may arise as they construct the scenario branching tree.

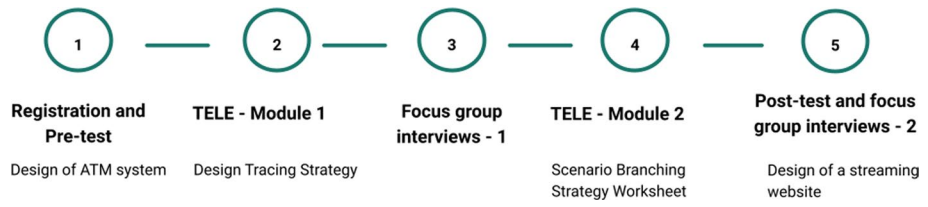


Fig. 13 Summary of the study procedure

Research design

In this section, we describe the study which we conducted to evaluate the effectiveness of the model-based learning TELE. We investigate how the TELE is enabling students to effectively uncover defects in the design, and how features in the TELE are contributing towards student learning.

Research questions

The research questions guiding this study are:

- RQ 1: What are the effects of the MBL TELE in students' ability to uncover defects in the design?
- RQ 2: How are features in the MBL TELE contributing towards student learning?

Study procedure

We conducted the study in an engineering institute located at a metropolitan city in our country. The study was spread over two days. Twenty-two students (m=16, f=6) in the second year of their undergraduate degree in Computer Science and Information Technology were part of the study. Prior to the study, they attended a workshop where they were taught basic UML diagrams like use case, class, and sequence diagrams. Hence, all students were familiar with UML diagrams prior to the study. Participants signed a consent form prior to the study and were free to withdraw from the study at any point in time.

A summary of the study procedure is shown in Fig. 13. At the start of the study, participants filled a registration form (pre-intervention survey) with their basic information like name, branch, overall percentage in the last semester, and were asked to rate their confidence in understanding of object-oriented design, class and sequence diagrams, and in identifying defects in design diagrams. The registration form also contained the consent form. We initially explained the main goals of the study, and what students will learn as they interact with the TELE. After providing this information, students solved a pre-test. In the pre-test, students were given the requirements and design diagrams of an ATM system and were asked to identify defects in the design diagrams based on the requirements. (The pre-test is attached as an Additional file 2.) After solving the pre-test, students interacted with Module 1 of the TELE. The first author was available to answer any doubts which students had while interacting with

Table 2 Summary of the research questions, their corresponding data source, and analysis method

Research question	Data source	Data analysis method
RQ 1: Ability to identify defects	Responses to "identify defects" question in the pre-test and post-test	Content analysis
RQ 2: Features in the TELE contributing towards student learning	(1) Focus group interviews, (2) Interaction log data	Thematic analysis of log data

Module 1. After a break of one hour, students were introduced to Module 2. The first author facilitated this session by explaining how to construct a scenario branching tree for the first requirement. Students interacted with the mapping tool and constructed scenario trees for the remaining requirements. The next day, students solved a post-test. The format of the post-test was similar to the pre-test. The requirements and design diagrams were for a video streaming website. (The post-test is attached as an Additional file 3.) The complexity of the design problem in the pre-test and post-test was similar, as both problems contained the same number and types of requirements, sequence diagrams, and classes in the class diagram. They then filled a feedback form (post-intervention survey), which contained usability and questions related to their confidence of design diagrams and identifying defects in design diagrams. We conducted focus group interviews with students in between Module 1 and 2, and at the end, in order to elicit their perceptions of the TELE and its features.

Data sources and analysis

We used three data sources to answer the research questions: (1) student written responses in the pre-test and post-test, (2) student interaction log data for Module 1, and (3) student responses in the focus group interviews. A summary of the research questions and corresponding data sources and analysis methods is shown in Table 2.

RQ 1: Students' ability to identify defects

To answer RQ 1, we analysed student responses to the following "identify defects" question in the pre-test and post-test—*"Identify defects (if any) in the following design diagrams based on the requirements. For each defect, provide a logical explanation of why you think it is a defect."* We collated all student answers to the "identify defects" questions and used the content analysis method (Baxter 1991) to analyse their answers. The steps we followed for the analysis is as follows:

1 *Representation of student answers:* The written text which each participant wrote as a response to the question was typed verbatim to a spreadsheet. We considered a written sentence or group of sentences which referred to a particular defect, as the unit of analysis. Each row in the spreadsheet corresponded to a sentence/sentences written by participants. Many students listed multiple defects. Out of the 22 students, 4 students either skipped the pre-test or the post-test and hence were excluded from the analysis. We identified 45 answers in the pre-test, and 49 answers from the post-test, from 18 student response sheets.

2 *Descriptive coding:* We assigned a descriptive code to each sentence. The objective of descriptive coding is to avoid any prejudices or preconceptions and stay close to the data. For example, for the following student response—*"In the Register Pin Sequence diagram, we do not use checkPinLength() before setPin(). By doing so, a user may add length(PIN) != 4 breaking the criteria."*, the descriptive code assigned was *"checkPinLength() function is not used before setPin() in register sequence diagram"*
3 *Generate categories and sub-categories:* We inferred categories and sub-categories based on the patterns, explanations, and relationships between the descriptive codes. The categories reflected the meaning inferred from the descriptive codes and can explain larger segments of data. We created memos and notes which described the category definitions and criteria for assigning descriptive codes to these categories. For example, in the descriptive code provided above, the sub-category inferred is *"checkPinLength() function is not called"*, and the category is *"Identify necessary functions which are not used."*.
4 *Determine number of responses in each category*: We then compared categories of student responses in the pre-test and post-test to examine differences in the types of defects students identified.

To establish reliability of the generated categories, two rounds of coding were done with a rater. In the first round, the first author explained the aims and research questions of the study to the rater. We then took 10 responses, discussed the categories corresponding to each response, and came to an agreement on conflicting entries. In the second round, the first author and rater independently assigned categories to another 17 responses. There was an agreement on 13 out of the 17 responses. The first author then independently assigned categories to the remaining responses.

RQ 2: Role of features in the TELE in students' learning

To answer RQ 2, we used students' interaction data from Module 1, and student answers from the focus group interviews. We explain the analysis methods for each in detail below.

As students interacted with the MBL TELE Module 1, specific user interactions were logged. We specifically asked students for consent to use their interaction logs in the consent form. A total of 12 students gave consent to use their interaction logs.

The interaction logs contained data such as which challenge the user was at a particular time and how they interacted with the state diagram model. Interactions with the state diagram included action sequences while constructing, editing, and executing the state diagram model in different challenges, and what feedback was provided by the agent as a result of these actions.

After conducting the study, we downloaded all user interaction logs from the server. These interactions of all users were stored in a single CSV file on the server, ordered by timestamp. We used the pandas library[8] in Python for the pre-processing and analysis. From the single CSV file, we extracted all interaction logs for each user who provided consent and ordered the data by timestamp. Hence, we had a separate file which

[8] https://pandas.pydata.org/

contained the interaction logs of each user. We then combined all user files and generated a new CSV file. This interaction log data describe the sequence of actions a user performed in the MBL TELE Module 1 and provide indicators of how features in the TELE were helping them perform the required activities (RQ 2).

In the focus group interviews, we asked questions regarding how students went about interacting with the TELE. The goal of these questions was to delve deeper into how students were interacting with the TELE and to gather their perceptions about various features in the TELE. We analysed the focus group interview transcripts using a thematic analysis approach (Braun and Clarke 2006). The purpose of thematic analysis is to identify patterns of meaning across a dataset that provide an answer to the research questions. The process followed in thematic analysis is similar to content analysis. However, the key difference is that while content analysis uses a descriptive approach in its interpretation of quantitative counts of the codes, thematic analysis provides a purely qualitative, detailed, and nuanced account of the data.

We transcribed the focus group interview and divided the transcript into sentence-level segments. We followed a similar process of descriptive and inferential coding and generated themes. The themes emerging from the student interviews were used to answer RQ 2.

Findings

RQ 1 Findings: Ability to identify defects

We identified the following categories which students provided when asked to identify defects in the design diagrams based on the requirements:

- *Identify scenarios which do not satisfy the requirements:* All responses in which students explicitly mentioned scenarios where the design is not satisfying the requirements were placed into this category (e.g. *"The sequence diagram for Register does not asks the user for new registration or login"*).
- *Change existing functionalities and requirements:* In their responses, students suggested improvements to existing functionalities as well as change of requirements. All responses where students change the requirement or add a sub-function to an existing functionality have been assigned to this category (e.g. *"a premium user wants to pay quarterly during registration—this case is also not being looked upon in the design"*).
- *Add new functionalities in the design:* Students also introduced new functionalities to the design, rather than identifying defects based on the requirements. We categorised all responses into this category if students are adding a completely new functionality, which is unrelated to any of the requirements provided (e.g. *"If a user forgets his/her PIN, there is no option available in the ATM to recovery"*).
- *Change data types, functions, and structure of the class diagram*: Another category of student responses was based on changes in the class diagram and change in data type of variables. For example, students incorrectly stated that the data type of card number, account number, balance, and PIN should be "int" rather than "string". Stu-

Table 3 Differences between pre-test and post-test in student responses to identifying defects

Categories	Total no of responses in pre-test	Total no of responses in post-test	Wilcoxon signed rank test			Significant difference ($p < 0.05$)
			Z	p	r	
Identifying scenarios which do not satisfy the requirements	12	37	−1.99	0.02	0.33	Yes
Change in data types, function and structure of class diagram	6	0	−1.83	0.03	0.30	Yes
Add new functionality	11	1	−1.83	0.03	0.30	Yes
Change in existing functionalities and requirements	14	8	−0.92	0.18	0.15	No
No defects	2	3	−0.53452	0.29649	0.089087	No

dents stated that some variables are missing in the class diagram and also suggested structural changes in the class diagram.

- *Blank responses and no defects:* Some students either left the answer blank or explicitly mentioned that there were no defects.

In the evaluation question, students were required to identify relevant scenarios which do not satisfy the requirement and not add new functionalities or change existing functionalities. However, adding and modifying existing functionalities in itself is not an undesirable behaviour. When experts are asked to create designs from requirements, they expand both the problem and the solution space. Experts expand the problem space by simulating alternate scenarios and hence derive new requirements and constraints which are not stated in the problem (Adelson and Soloway 1985). However, the task presented to students was different from a design creation task. For this task, the objective was to identify defects based on the requirement, and hence adding or changing functionalities was not the goal. In previous studies with novices, we observed that students added and modified existing functionalities instead of evaluating the given design. We did not explicitly tell them not to add new functionalities, as we wanted to observe how the TELE has facilitating this change.

For the evaluation question, many students identified multiple defects corresponding to the identified categories. Differences in the category responses of students in the pre-test and post-test are shown in Table 3. Using the Shapiro–Wilk test, we observed a significant departure from normality for all five categories. Hence, we used the Wilcoxon signed rank test to compare the pre-test and post-test responses in each category. The Wilcoxon signed rank test indicated that students identified more scenarios which do not satisfy the requirements in the post-test compared to the pre-test ($Z = -1.99, p = 0.02, r = 0.33$). Students also identified lesser "Change in data type" ($Z = -1.83, p = 0.03, r = 0.3$) and "Add new functionality" defects in the post-test compared to the pre-test ($Z = -1.83, p = 0.03, r = 0.3$). However, we did not see significant

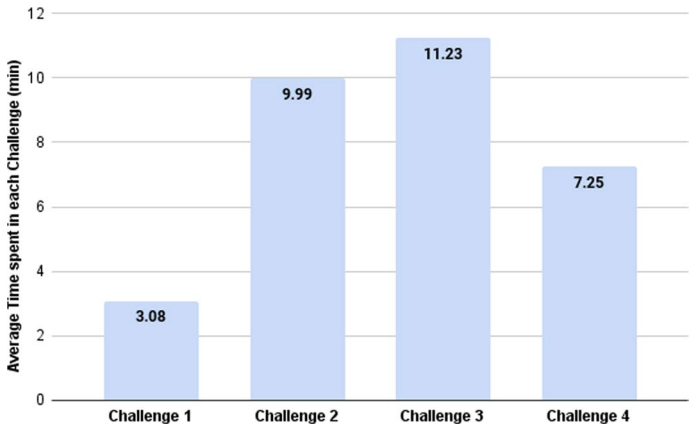


Fig. 14 Average time spent by 12 students in each challenge of the Design Tracing Stage

differences in the "Change existing functionalities and requirements" and "No defects" categories in the pre-test and post-test.

We can thus infer that students were able to identify more defects by identifying scenarios which do not satisfy the requirement in the post-test. Students adding new functionalities and referring to them as defects have reduced in the post-test. These results indicate that students have improved in evaluating the design against the given requirements.

RQ 2: Role of features in the TELE in students' learning

Based on the analysis of the interaction data and thematic analysis of the focus group interview transcripts, we identified three broad categories of how features in the TELE are contributing towards student learning: (1) the model-progression activities, (2) the model execution visualisation features, and (3) the scenario branching strategy. We explain each in detail below.

The "Design Tracing Stage" activities help students progressively trace scenarios in the design

What are certain indicators that students found the progression of activities in the "Design Tracing Stage" beneficial? We examined the following attributes from the interaction log data and interviews to answer this question.

- *The order students followed:* Using the interaction logs, we determined the order in which students interacted with different activities. We looked at what are possible paths learners took as they interacted with the TELE.
- *Time spent in each challenge*: The interaction logs captured the timestamp of the start and end of each activity. Thus, the time spent in each challenge can serve as a proxy for the difficulty for that challenge.
- *Students perceptions*: During the focus group interviews, we asked students their perception of the order of activities. These responses complemented the results from the log data and helped us gauge the effectiveness of the order of activities.

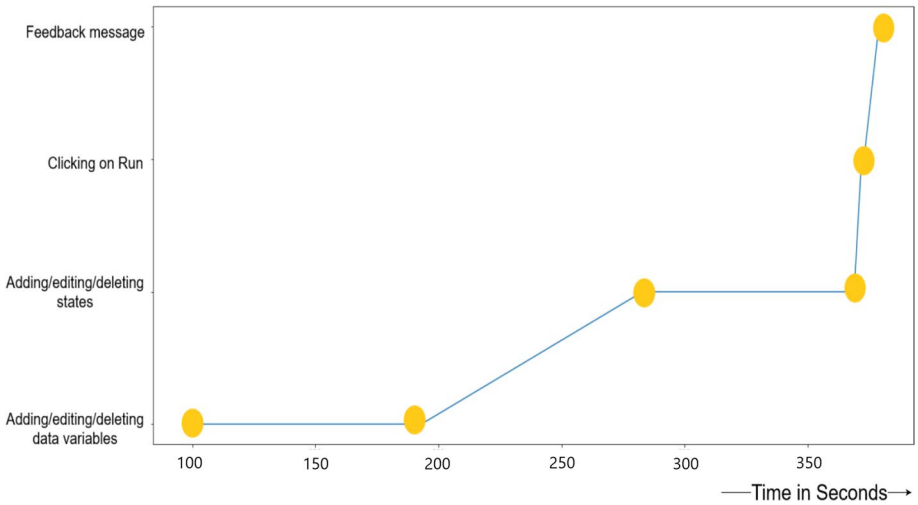


Fig. 15 Example graph of the single run strategy

Findings from interaction logs

From the interaction logs, we observed that all 12 students followed the prescribed order of challenges. We also looked at the average time spent by students in each task. We chose the metric of average time spent, as the time spent on a task can be considered as a reasonable estimate of the difficulty of that task. Figure 14 shows the average time spent in each challenge by students. We observe that students spent the least time on Challenge 1. This is obvious, as learners had to only explore the state diagram and observe its execution in Challenge 1.

However, we observed that students spent the most time in Challenge 3. Based on the model progression of activities, Challenge 4 was the most difficult one among the challenges, since students had to construct the entire state diagram from scratch. Hence, it is reasonable to assume that students will spend the most time in Challenge 4. However, we observed that students spent more time in Challenge 3, followed by Challenge 2. Students spending lesser time in Challenge 4 than Challenges 2 and 3 gives an indication that the model progression helped students construct the model of a scenario. By the time students reached Challenge 4, they had already observed a correct trace, corrected an incorrect trace, and completed an incomplete scenario trace. Hence, they required lesser time to construct the model of a scenario from scratch in Challenge 4.

Findings from focus group interviews

In the "Design Tracing Stage" of Module 1 of the TELE, the challenges were ordered in increasing order of complexity, where students first explored the model, then corrected an incorrect model, completed an incomplete model, and finally constructed the model from scratch. Most of the students in the focus group interview mentioned that they followed the challenges in order. The key perception by students was that they found the ***challenges in increasing order of difficulty***. Each challenge ***added***

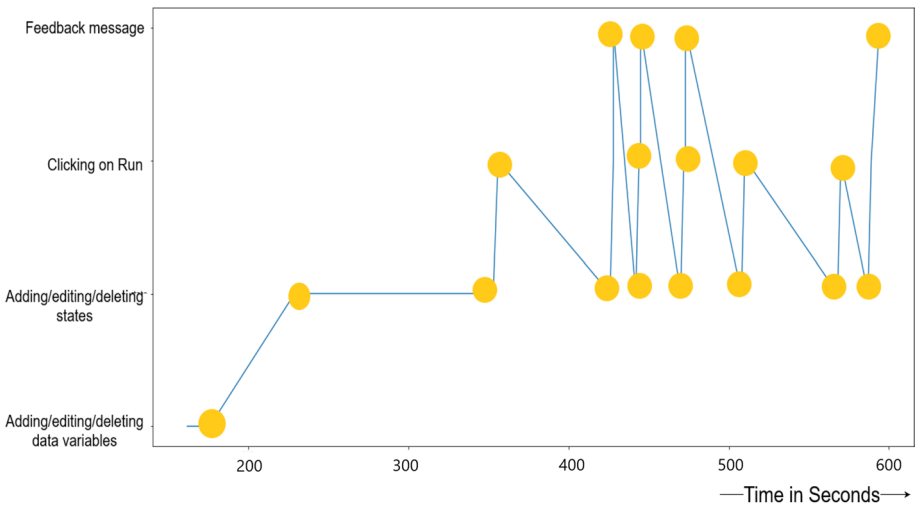


Fig. 16 Example graph of the multiple runs strategy

some more complexity than the previous challenge (*"Firstly, the whole diagram was given, then correct the mistake...We were going step by step"*) This ***enhanced their understanding*** of the design diagrams (*"We were going step by step—understood how the system really work"*) as well as ***enabled them to construct the state diagram at the end.*** (*Like we solved it sequentially. So...And when nothing was given, it was the last question. So I basically had the idea how to go about solving the last challenge*). These perceptions give us indicators that the model progression of the challenges enabled students to model the scenarios in the design. A more open-ended approach (introducing Challenge 3 or Challenge 4 at the start) may not have achieved the desired effect (e.g.*"1. Rather than just pushing us to create state diagrams, it helped us move step by step. 2. It (solving Challenge 4) became easier for us by exploring the model in the first state"*).

These findings from interaction logs and focus group interviews give us indicators that the model progression activities help students progressively trace scenarios in the design.

The model execution visualisation features in Module 1 help students visualise execution of scenarios in the design

We examined the interaction logs and interviews to explore the usefulness of the model execution visualisation.

- *Patterns of using "run" in each challenge:* In Module 1 of the TELE, actions such as adding/editing/deleting data, events, and states are logged by the system in the interaction logs. We analysed patterns of such actions for each user. We excluded Challenge 1, since students did not modify the state diagram in this challenge. For each challenge, a user's actions were plotted onto a graph, with the x-axis corresponding to the time, and the y-axis corresponding to the actions in that challenge (examples in Figs. 15, 16). For the 12 students who gave consent for use of their

Table 4 Number of students using a strategy in each Design Tracing Stage challenge based on their interaction logs

	Challenge 2	Challenge 3	Challenge 4
Single run strategy	2	2	8
Few runs strategy	2	4	2
Multiple runs strategy	8	6	2

interaction logs, we manually examined the 36 graphs (12 × 3) in order to identify different strategies which students used.

- *Student perceptions:* From the focus group interviews, we examined what were students' perceptions about the model execution visualisation and how it helped them in the challenges.

Patterns of using the run visualisation in challenges

Manual examination of the graphs led to identification of three distinct strategies across the three challenges.

Single run strategy: All modifications followed by single run at the end: In this strategy, students worked on the state diagram by modifying (adding/editing/deleting) data variables, and modifying (adding/editing/deleting) states, and clicked on "Run" only at the end to verify their correctly constructed state diagram (an incorrect state diagram would have led to further edits and runs). An example graph of this strategy is shown in Fig. 15. We saw 12 instances (out of 36) across the three challenges of students using the single run strategy.

Few runs strategy: Between one to three cycles of modification and run: In this strategy, students modified data variables and states, clicked on "Run", and modified the data/states again. Students who followed up to three cycles of this behaviour were classified to this strategy. We saw 8 instances (out of 36) across the three challenges of students using this strategy.

Multiple runs strategy: Multiple cycles of modification and run: In this strategy, students performed multiple cycles of modifying data variables and states, and clicking on "Run", as shown in Fig. 16. We saw 16 instances (out of 36) across the three challenges of students using this strategy.

We analysed the predominant strategies used in each challenge, and how transitions of strategies are happening in different challenges. The transitions corresponding to the interaction logs for the 12 students are shown in Table 4. We see that 8 out of 12 students attempted Challenge 2 by employing multiple cycles of modification and run (multiple runs strategy). But by the time they reached Challenge 4, 8 out of 12 students shifted to using the "Run" only at the end (single run strategy), and only 2 out of 12 students still used the multiple runs strategy.

What can this shift from "Multiple Runs" to "Single Run" indicate? Students execute the state diagram to get feedback on their model. Hence, at the start (Challenge 2), they

used the "Run" scaffold to frequently validate their model. Based on this feedback, they refined their model. By the time they reached Challenge 4, the previous challenges had helped them visualise the execution of the control flow and data flow of a given scenario. Thus, in Challenge 4, they did not explicitly use the "Run" scaffold, but could internalise the control flow and data flow for the given scenario. They constructed their model, and verified it at the end.

Findings from focus group interviews

Students perceived that the model execution visualisation feature in Module 1 of the TELE was useful. First, *the state diagram execution helped students map a particular state to the corresponding part of the scenario* (*"if we pressed the run button, it showed what part of the scenario covers which state...Basically, it linked the scenario with the state."*). Students also found that the highlighting of the scenario as well as the sequence diagram was useful in *understanding the relationship between the scenario and different diagrams* (*e.g. "It was mapping the step-by-step scenario with the sequence diagram"*).

Visual feedback helped learners identify which parts had errors. (*"If there was something wrong, then it would be in a different colour"*). Students compared the "run" feature to debugging in programming, and this helped them build the model and rectify it in case of errors. *Positive feedback also served as an indicator that a sub-part of the model was correctly modelled,* and students could move on to the next part (*"...it (clicking on run) indicates that the first step is now completed. We can move onto the second step"*). Students found the step-wise run more beneficial as compared to a "run" at the end, because it helped them rectify errors in their model. (*e.g. "Because at the first step, we need to know where we are wrong. After doing whole execution and getting to know and going back on that same point is more difficult and tedious and correcting it at that point."*). Thus, the step-wise execution and evaluation of the model helped students break down the task of modelling the scenario into smaller tasks, thereby making the task easier.

The scenario branching strategy helps in identifying scenarios in the design

In Module 2 of the TELE, students interacted with the mapping tool to construct a scenario branching tree for each requirement. We analysed their perceptions in order to understand the role of the mapping tool in identifying scenarios.

Students found the scenario branching strategy useful in identifying scenarios. They commented on how it was useful in *structuring the design problem* (*"We didn't know which way to go to solve the problem. Now we know a structured way to follow up problems"*). The strategy helped in *breaking down the problem* into scenarios (*"problem can be broken down into scenarios and can be represented in a single diagram"*). The visual representation of the scenario tree enabled them to view all possible scenarios for each requirement (*"Cmap I think, like in a very broad manner, is a connection of state diagrams. Like it entirely shows the whole picture"*) and hence helped them get a *macro-view of the design problem* (*"It gives a bigger picture about the task that we are trying to solve."*). The simplicity of the representation aided them to *identify scenarios which were missing in the design diagrams* (*"In Cmap, we can include all the scenarios. But in sequence*

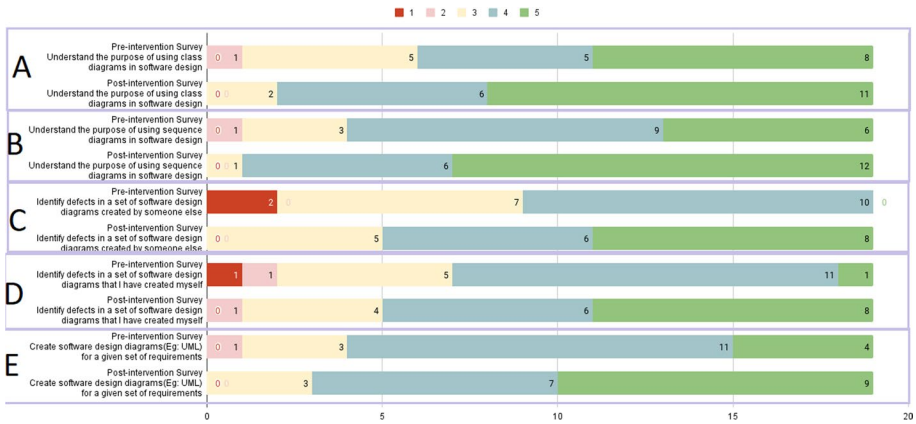


Fig. 17 Comparison of responses given in the pre-intervention and post-intervention surveys. In the surveys, 1 indicates "Not confident at all", and 5 indicates "Absolutely confident"

diagrams and all, we sometimes, fail to do that, sometimes some scenarios get left out, because the diagram is very complex. But in Cmap, it becomes easy to find out the scenario is remaining, let's add it there.")

Hence, the explicit focus on scenarios helped students *evaluate the given design*, which they found difficult during the pre-test (*"I was able to think about more scenarios. In the ATM problem, in that problem I was not able to think what can be done new, but after learning this new session, I was able to elaborate more about what those scenarios can be."*). It also helped them *identify missing functions in the sequence diagrams* (*"If one function is missing, by noting down the scenarios we can add them...I identified one function like, when the user types the username and password, then there was no function to check whether the username is already there or not."*)

Discussion

The MBL TELE enables learners to effectively evaluate design diagrams against the given requirements

Findings from RQ 1 show that there is a change in how students evaluate a given design against the requirements. Based on the differences in the pre-test and post-test, we see that students' understanding of evaluation evolved from merely adding new functionalities and requirements, to a process which involved identifying alternate scenarios in the design which violate the given requirements ("RQ 1 Findings: Ability to identify defects" section). The emphasis of the MBL pedagogy on identifying and modelling scenarios enabled them to perform the design evaluation task.

We see instances of this change in their thinking from the focus group interviews as well (e.g. *"I was able to think about more scenarios. In the ATM problem, in that problem I was not able to think what can be done new, but after learning this new session, I was able to elaborate more about what those scenarios can be."*).

We also analysed the pre-intervention and post-intervention survey responses to examine differences in their confidence in identifying defects in design diagrams. We see that there is an increase in confidence in identifying defects in design diagrams created by others as well as for identifying defects in their own design diagrams. From

Fig. 17 Part C, we can see that in the pre-intervention survey, 10/19 (53%) students rated their confidence as 4 and above, in identifying defects in design diagrams created by others, with no one having rated their confidence as 5. In the post-intervention survey, 14/19 (74%) students rated their confidence as 4 and above, with 8 students rating their confidence as 5 (In the survey, 1 indicates "Not confident at all", and 5 indicates "Absolutely confident"). We see a similar trend for students' rating their confidence in identifying defects in their own design diagrams (Fig. 17 Part D).

Features in the MBL TELE enable learners to perform effective software design evaluation

Findings from RQ 2 show that the features in the MBLE TELE were beneficial for learners. By analysing student interaction logs and their responses in the focus group interview, we found that the activities in the design tracing stage enabled students to progressively trace scenarios in the design. This also validates findings from our previous study, which showed that there is an improvement in students' ability to trace scenarios in the design (Prasad and Iyer 2020b). These findings give evidence for the appropriateness of the model progression activities in enabling learners to construct a models of various scenarios in the design.

Second, the model execution visualisation features in the Design Tracing Stage helped learners simulate the control and data flow of various scenarios in the design, by helping students map a particular state to the corresponding part of the scenario. These features scaffolded learners to model various scenarios in the design. Third, the scenario branching strategy enabled students to structure the design problem and break down the problem into scenarios, thereby enabling them to identify scenarios which were missing in the design diagrams.

The MBL TELE brought about a change in learners' approach towards software design

Students also perceived that interactions with the MBL TELE brought about a change in their approach towards the software design process. In the focus group interviews, students were asked if there were any differences in their approach in evaluating designs in the pre-test and post-test. Students mentioned that the TELE enabled them to think of a *structured way to solve the problem* (e.g. *"We didn't know which way to go to solve the problem. Now we know a structured way to follow up problems, now I could directly go for another problem."*).

Findings from the pre-intervention and post-intervention survey responses show that there is an increase in confidence in students' understanding of the purpose of class and sequence diagrams (Fig. 17 Parts A and B). We see evidence of this in the focus group interviews as well. In these interviews, students reflected on how interactions with the MBL TELE brought about a change in their perception about the *importance of creating a software design before jumping into writing code* (e.g. *"For me I thought firstly before this workshop, I thought like, programming was the main thing behind all the project. But right now, I've learnt that how a blueprint is made for a project, and what software designing is. That was quite useful, it was something new."*). It broadened their perspective of thinking of the system as a whole, before jumping into specifics.

Findings from the survey and focus group interviews show that the VeriSIM pedagogy can be used to help students in the design creation process as well. We see an increase

in students' confidence in creating design diagrams from a given set of requirements (Fig. 17 Part E). From the focus group interviews, we see that students are also able to reflect on how they will use the strategies learnt when they encounter a new design problem. Consider the following excerpt from an interview:

"In my case, I will use first make the use case of the system, and the user who will be the actor. Then I will create all the classes related to the entities present. And after that I will create the scenario what the user will be doing...So the listing out of scenario will be done by me. And after that, looking at the links of the scenario, I will be making the sequence part, and sequence part will be implemented by the state diagrams. So that the proper execution of the program can be done." From this excerpt, we see that students are able to appropriate how they will augment the strategies learnt to create a software design for a given problem.

Limitations

There are certain limitations in this study. First, we have restricted our focus to semantic deficiencies and emphasised on helping students identify and model relevant scenarios. Evaluation involves (in addition to semantics) other aspects such as syntactic and pragmatic deficiencies, as well as non-functional requirements such as modularity and extensibility. These perspectives of evaluation go beyond identifying relevant scenarios and mapping requirements to design diagrams and have not been investigated in this paper.

Second, the designs provided to learners were from familiar domains and were fairly simplistic. However, we believe that the key goal of the VeriSIM pedagogy, of identifying and modelling scenarios in the design, will aid them as they evaluate complex designs as well. This can also be investigated in future studies, where the TELE is modified to include fairly complex designs as well.

Third, the study was not a quasi-experimental study and did not have a control group. The goal of this paper was to show that the VeriSIM pedagogy is a possible solution for teaching–learning of software design evaluation. Future studies with larger samples can explore the effect of other instructional strategies in students' evaluation skills and how it compares with the VeriSIM pedagogy.

Conclusion

In this paper, we described the design and evaluation of a technology-enhanced learning environment, based on the model-based learning pedagogy, aimed at helping students effectively evaluate a software design against the given requirements. Findings from our study conducted with 22 undergraduate students show that there is an improvement in how students evaluate a given design. Students perceived that pedagogical features of the TELE were useful in helping them identify and model various scenarios in the design.

The MBL TELE is available online for anyone to use at the following link - https://verisim.tech/. The TELE can be directly used by instructors as well as students to be trained in evaluating design diagrams against the requirements. The TELE incorporates various

design features such as the ability to construct, modify, and visualise the execution of a given scenario. This extends current work in program visualisation literature, which has primarily looked at visualisations of program execution. Learning environment designers can adapt and extend these features for other software design diagrams and contexts, as well as for the teaching and learning of programming.

Findings from the study give evidence for the model-based learning paradigm as an appropriate pedagogy for software design. Although model-based learning has been extensively studied and applied in science education research, it has not been explored in software design, or even computing education in general. We believe this study opens the space for researchers to investigate the model-based learning paradigm in other aspects of software design. Variations of this paradigm can be applied to create pedagogies and designs which investigate learning and its effectiveness in other contexts—such as for unfamiliar problem domains and designs of different complexities.

Abbreviations
UML: Unified modelling language; MBL: Model-based learning; TELE: Technology-enhanced learning environment.

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s41039-022-00192-0.

Additional file 1. Scenario branching worksheet.

Additional file 2. Pre-test.

Additional file 3. Post-test.

Acknowledgements
The authors would like to acknowledge Kinnari Gatare for the user-interface design of the TELE, Bhupender Singh Maan for the development of the TELE, and T. G. Lakshmi and Herold P. C. for the initial design and planning of activities in the TELE.

Author contributions
PP carried out the development and research of the MBL TELE under the guidance of SI. He also drafted the manuscript, which was checked by SI. Both authors read and approved the final manuscript.

Funding
Not applicable.

Availability of data and materials
Relevant data excerpts supporting the findings are included in the article.

Declarations

Ethics approval and consent to participate
Ethics approval was taken from the author's department. A consent form was signed by all participants at the start of the study.

Competing interests
The authors declare that they have no competing interests.

Received: 27 April 2021 Accepted: 31 March 2022
Published online: 24 May 2022

References

Adelson, B., & Soloway, E. (1985). The role of domain expenence in software design. *IEEE Transactions on Software Engineering, 11*, 1351–1360.

Adelson, B., & Soloway, E. (1986). A model of software design. *International Journal of Intelligent Systems, 1*(3), 195–213.

Azevedo, R., Cromley, J. G., Moos, D. C., Greene, J. A., & Winters, F. I. (2011). Adaptive content and process scaffolding: A key to facilitating students' self-regulated learning with hypermedia. *Psychological Test and Assessment Modeling, 53*(1), 106.

Baker, A., Navarro, E. O., & Van Der Hoek, A. (2005). An experimental card game for teaching software engineering processes. *Journal of Systems and Software, 75*(1–2), 3–16.
Baxter, L. A. (1991). Content analysis. *Studying Interpersonal Interaction, 239,* 254.
Begel, A., & Simon, B. (2008). Novice software developers, all over again. In *Proceedings of the fourth international workshop on computing education research* (pp. 3–14). ACM.
Begel, A., & Simon, B. (2008). Struggles of new college graduates in their first software development job. In *ACM SIGCSE Bulletin* (Vol. 40, pp. 226–230). ACM.
Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology, 3*(2), 77–101. https://doi.org/10.1191/1478088706qp063oa.
Brechner, E. (2003). Things they would not teach me of in college: What microsoft developers learn later. In *Companion of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications* (pp. 134–136). ACM.
Buckley, B. C., Gobert, J. D., Horwitz, P., & O'Dwyer, L. M. (2010). Looking inside the black box: Assessing model-based learning and inquiry in biologica™. *International Journal of Learning Technology, 5*(2), 166–190.
Burgueño, L., Vallecillo, A., & Gogolla, M. (2018). Teaching UML and OCL models and their validation to software engineering students: An experience report. *Computer Science Education, 28*(1), 23–41.
Chase, J. D., Uppuluri, P., Lewis, T., Barland, I., & Pittges, J. (2015). Integrating live projects into computing curriculum. In *Proceedings of the 46th ACM technical symposium on computer science education* (pp. 82–83). ACM.
Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., Paige, R. F., Pierantonio, A., Rensink, A., Salay, R., Taentzer, G., et al. (2018). How do we teach modelling and model-driven engineering? A survey. In *Proceedings of the 21st ACM/IEEE international conference on model driven engineering languages and systems: Companion proceedings* (pp. 122–129).
Clement, J. (2000). Model based learning as a key research area for science education. *International Journal of Science Education, 22*(9), 1041–1053.
Dagenais, B., Ossher, H., Bellamy, R. K., Robillard, M. P., & De Vries, J. P. (2010). Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE international conference on software engineering* (Vol. 1, pp. 275–284). ACM.
Dym, C. L., Agogino, A. M., Eris, O., Frey, D. D., & Leifer, L. J. (2005). Engineering design thinking, teaching, and learning. *Journal of Engineering Education, 94*(1), 103–120.
Fitzgerald, S., Simon, B., & Thomas, L. (2005). Strategies that students use to trace code: An analysis based in grounded theory. In *Proceedings of the first international workshop on computing education research* (pp. 69–80). ACM.
Fretz, E. B., Wu, H.-K., Zhang, B., Davis, E. A., Krajcik, J. S., & Soloway, E. (2002). An investigation of software scaffolds supporting modeling practices. *Research in Science Education, 32*(4), 567–589.
Ge, X., & Land, S. (2004). A conceptual framework for scaffolding ill-structured problem-solving processes using question prompts and peer interactions. *Educational Research Technology and Development, 52*(2), 1042–1629.
Guindon, R. (1990). Knowledge exploited by experts during software system design. *International Journal of Man–Machine Studies, 33*(3), 279–304.
Hestenes, D. (2010). Modeling theory for math and science education. In *Modeling students' mathematical modeling competencies* (pp. 13–41). Springer.
Hestenes, D. (1987). Toward a modeling theory of physics instruction. *American Journal of Physics, 55*(5), 440–454.
Hestenes, D. (1992). Modeling games in the Newtonian world. *American Journal of Physics, 60*(8), 732–748.
Hohmann, L., Guzdial, M., & Soloway, E. (1992). SODA: A computer-aided design environment for the doing and learning of software design. In *International conference on computer assisted learning* (pp. 307–319). Springer.
Hu, C. (2013). The nature of software design and its teaching: An exposition. *ACM Inroads, 4*(2), 62–72.
Hungerford, B. C., Hevner, A. R., & Collins, R. W. (2004). Reviewing software diagrams: A cognitive study. *IEEE Transactions on Software Engineering, 30*(2), 82–96.
Jaramillo, C. M. Z. (2014). Teaching software development by means of a classroom game: The software development game. *Developments in Business Simulation and Experiential Learning, 36*.
Kim, J., Hahn, J., & Hahn, H. (2000). How do we understand a system with (so) many diagrams? Cognitive integration processes in diagrammatic reasoning. *Information Systems Research, 11*(3), 284–303.
Kopainsky, B., & Alessi, S. M. (2015). Effects of structural transparency in system dynamics simulators on performance and understanding. *Systems, 3*(4), 152–176.
Kopainsky, B., Alessi, S. M., Pedercini, M., & Davidsen, P. I. (2015). Effect of prior exploration as an instructional strategy for system dynamics. *Simulation & Gaming, 46*(3–4), 293–321.
Lindland, O. I., Sindre, G., & Solvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE Software, 11*(2), 42–49.
Mc Neill, T., Gero, J. S., & Warren, J. (1998). Understanding conceptual electronic design using protocol analysis. *Research in Engineering Design, 10*(3), 129–140.
Montazemi, A. R., & Conrath, D. W. (1986). The use of cognitive mapping for information requirements analysis. *MIS Quarterly, 10,* 45–46.
Mulder, Y. G., Bollen, L., de Jong, T., & Lazonder, A. W. (2016). Scaffolding learning by modelling: The effects of partially worked-out models. *Journal of Research in Science Teaching, 53*(3), 502–523.
Mulder, Y. G., Lazonder, A. W., & de Jong, T. (2011). Comparing two types of model progression in an inquiry learning environment with modelling facilities. *Learning and Instruction, 21*(5), 614–624.
Nelson, M., & Piattini, M. (2012). A systematic literature review on the quality of UML models. In *Innovations in database design, web applications, and information systems management* (pp. 310–334).
Papaevripidou, M., & Zacharia, Z. C. (2015). Examining how students' knowledge of the subject domain affects their process of modeling in a computer programming environment. *Journal of Computers in Education, 2*(3), 251–282.
Prasad, P., & Iyer, S. (2020a). How do graduating students evaluate software design diagrams? In *Proceedings of the 2020 ACM conference on international computing education research* (pp. 282–290).
Prasad, P., & Iyer, S. (2020b). Verisim: A learning environment for comprehending class and sequence diagrams using design tracing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (pp. 23–33). IEEE.

Quintana, H., & Grados, B. (2020). Applying pair programming practice in the improvement of software design skills, in an undergraduate course. In *Proceedings of the 2020 ACM conference on innovation and technology in computer science education* (pp. 543–544).
Reddy, D., Alse, K., TG, L., Prasad, P., & Iyer, S. (2021). Learning environments for fostering disciplinary practices in CS undergraduates. In *Proceedings of the 52nd ACM technical symposium on computer science education* (pp. 1287–1287).
Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified modeling language reference manual*. The Pearson Higher Education.
Schumacher, R. M., & Czerwinski, M. P. (1992). Mental models and the acquisition of expert knowledge. In *The psychology of expertise* (pp. 61–79). Springer.
Sien, V. Y. (2011). An investigation of difficulties experienced by students developing unified modelling language (UML) class and sequence diagrams. *Computer Science Education, 21*(4), 317–342.
Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE), 13*(2), 8.
Stikkolorum, D. R., & Chaudron, M. R. (2016). A workshop for integrating UML modelling and agile development in the classroom. In *Proceedings of the computer science education research conference* (pp. 4–11). ACM.
Tang, A., Aleti, A., Burge, J., & van Vliet, H. (2010). What makes software design effective? *Design Studies, 31*(6), 614–640.
Travassos, G., Shull, F., Fredericks, M., & Basili, V. R. (1999). Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *ACM sigplan notices* (Vol. 34, pp. 47–56). ACM.
Westphal, B. (2019). Teaching software modelling in an undergraduate introduction to software engineering. In *ACM/IEEE 22nd International conference on model driven engineering languages and systems companion (MODELS-C)* (pp. 690–69). IEEE.
White, B. Y., & Frederiksen, J. R. (1990). Causal model progressions as a foundation for intelligent learning environments. *Artificial Intelligence, 42*(1), 99–157.
Whittle, J., Bull, C., Lee, J., & Kotonya, G. (2014). Teaching in a software design studio: Implications for modeling education.
Wijnen, F. M., Mulder, Y. G., Alessi, S. M., & Bollen, L. (2015). The potential of learning from erroneous models: Comparing three types of model instruction. *System Dynamics Review, 31*(4), 250–270.
Winn, W. (1994). Contributions of perceptual and cognitive processes to the comprehension of graphics. In *Advances in Psychology* (Vol. 108, pp. 3–27). Elsevier.
Zannier, C., Chiasson, M., & Maurer, F. (2007). A model of design decision making based on empirical results of interviews with software designers. *Information and Software Technology, 49*(6), 637–653.

Publisher's Note