Analysis of Students' Programming Knowledge and Error Development

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von

Ella Albrecht
aus Pawlodar, Kasachstan

Göttingen, August 2021

Betreuungsausschuss

Prof. Dr. Jens Grabowski,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Carsten Damm,
Institut für Informatik, Georg-August-Universität Göttingen

Dr. Henrik Brosenne,
Institut für Informatik, Georg-August-Universität Göttingen

Mitglieder der Prüfungskommission

Referent:	Prof. Dr. Jens Grabowski, Institut für Informatik, Georg-August-Universität Göttingen
Korreferent:	Prof. Dr. Carsten Damm, Institut für Informatik, Georg-August-Universität Göttingen

Weitere Mitglieder der Prüfungskommission

Prof. Dr. Marcus Baum,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Dieter Hogrefe,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Delphine Reinhardt,
Institut für Informatik, Georg-August-Universität Göttingen

Prof. Dr. Stephan Waack,
Institut für Informatik, Georg-August-Universität Göttingen

Tag der mündlichen Prüfung
13. September 2021

Abstract

Learning to program is a hard task since it involves different types of specialized knowledge. You do not only need knowledge about the programming language and its concepts, but also knowledge from the problem domain and general problem solving abilities. Knowing how students develop programming knowledge and where they struggle, may help in the development of suitable teaching strategies. However, the ever increasing number of students makes it more and more difficult for educators to identify students' needs, problems, and deficiencies.

The goal of the thesis is to gain insights into students programming knowledge development based on their solutions to programming exercises. Knowledge is composed of so called *knowledge components* (KCs). In this thesis, we focus on KCs on a syntactic level, which can be derived from abstract systax trees, e.g., loops, comparison, etc., and semantic level, represented by so called *roles of variables*.

Since knowledge is not directly measurable, skill models are an often used for the estimation of knowledge. But, the programming domain has its own characteristics which have to be considered when selecting an appropriate skill model. One of the main characteristics of the programming domain are the dependencies between KCs. Hence, we propose and evaluate a Dynamic Bayesian Network (DBN) for skill modeling which allows to model that dependencies explicitly. Besides the choice of a concrete model, also certain meta-parameters like, e.g., the granularity level of KCs, has to be set when designing a skill model. Therefore, we evaluate how meta-parameterization affects the prediction performance of skill models and which meta-parameters to choose. We use the DBN to create learning curves for each KC and deduce implications for teaching from them.

But not only students knowledge but also their "mal-knowledge" is of importance. Therefore, we manually inspect students' programming errors and determine the error's frequency, duration, and re-occurrence. We distinguish between the error categories *syntactic*, *conceptual*, *strategic*, *sloppiness*, *misinterpretation*, and *domain* and analyze how the errors change over time. Moreover, we use k-means clustering to identify different patterns in the development of programming errors.

The results of our case studies are promising. We show that the correct meta-parameterization has a huge effect on the prediction performance of skill models. In addition, our DBN performs as well as the other skill models while providing better interpretability. The learning curves of KCs and the analysis of programming errors provide valuable information which can be used for course improvement, e.g., that students require more practice opportunities or are struggling with certain concepts.

Zusammenfassung

Programmieren zu lernen ist für viele eine große Herausforderung, da es unterschiedliche Fähigkeiten erfordert. Man muss nicht nur die Programmiersprache und deren Konzepte kennen, sondern es erfordert auch spezifisches Domänenwissen und eine gewisse Problemlösekompetenz. Wissen darüber, wie sich die Programmierkenntnisse Studierender entwickeln und welche Schwierigkeiten sie haben, kann dabei helfen, geeignete Lehrstrategien zu entwickeln. Durch die immer weiter steigenden Studierendenzahlen wird es jedoch zunehmend schwieriger für Lehrkräfte, die Bedürfnisse, Probleme und Schwierigkeiten der Studierenden zu erkennen.

Das Ziel dieser Arbeit ist es, Einblick in die Entwicklung von Programmierkenntnissen der Studierenden anhand ihrer Lösungen zu Programmieraufgaben zu gewinnen. Wissen setzt sich aus sogenannten *Wissenskomponenten* zusammen. In dieser Arbeit fokussieren wir uns auf syntaktische Wissenskomponen, die aus abstrakten Syntaxbäumen abgeleitet werden können, und semantische Wissenskomponenten, die durch sogenannte *Variablenrollen* repräsentiert werden.

Da Wissen an sich nicht direkt messbar ist, werden häufig Skill-Modelle verwendet, um den Kenntnissstand abzuschätzen. Jedoch hat die Programmierdomäne ihre eigenen speziellen Eigenschaften, die bei der Wahl eines geeigneten Skill-Modells berücksichtigt werden müssen. Eine der Haupteigenschaften in der Programmierung ist, dass die Wissenskomponenten nicht unabhängig voneinander sind. Aus diesem Grund schlagen wir ein dynamisches Bayesnetz (DBN) als Skill-Modell vor, da es erlaubt, diese Abhängigkeiten explizit zu modellieren. Neben der Wahl eines passenden Skill-Modells, müssen auch bestimmte Meta-Parameter wie beispielsweise die Granularität der Wissenkomponenten festgelegt werden. Daher evaluieren wir, wie sich die Wahl von Meta-Parameters auf die Vorhersagequalität von Skill-Modellen auswirkt und wie diese Meta-Parameter gewählt werden sollten. Wir nutzen das DBN, um Lernkurven für jede Wissenskomponenten zu ermitteln und daraus Implikationen für die Lehre abzuleiten.

Nicht nur das Wissen von Studierenden, sondern auch deren “Falsch”-Wissen ist von Bedeutung. Deswegen untersuchen wir zunächst manuell sämtliche Programmierfehler der Studierenden und bestimmen deren Häufigkeit, Dauer und Wiederkehrrate. Wir unterscheiden dabei zwischen den Fehlerkategorien syntaktisch, konzeptuell, strategisch, Nachlässigkeit, Fehlinterpretation und Domäne und schauen, wie sich die Fehler über die Zeit entwickeln. Außerdem verwenden wir k-means-Clustering um potentielle Muster in der Fehlerentwicklung zu finden.

Die Ergebnisse unserer Fallstudien sind vielversprechend. Wir können zeigen, dass die

Wahl der Meta-Parameter einen großen Einfluss auf die Vorhersagequalität von Modellen hat. Außerdem ist unser DBN vergleichbar leistungsstark wie andere Skill-Modelle, ist gleichzeitig aber besser zu interpretieren. Die Lernkurven der Wissenskomponenten und die Analyse der Programmierfehler liefern uns wertvolle Erkenntnisse, die der Kursverbesserung helfen können, z.B. dass die Studierenden mehr Übungsaufgaben benötigen oder mit welchen Konzepten sie Schwierigkeiten haben.

Acknowledgements

During the time of my thesis I was supported by several persons who I want to thank. First of all, I want to thank my supervisor Prof. Dr. Jens Grabowski for his guidance, helpful suggestions, and for pushing me forwards to finally finalize my thesis. Working in his research group contributed not only to my professional but also to my personal growth. In addition, I would like to thank my second and third supervisors Prof. Dr. Carsten Damm and Dr. Henrik Brosenne for their valuable feedback during my presentations.

I also want to thank all my colleagues at the Institute for Computer Science for all the fruitful discussions, the fun moments, and for sharing frustrations – A problem shared, is a problem halved. With them, work did not feel like work. I am not sure if I ever will find such wonderful colleagues again. Special thanks go to Patrick Harms and Simone Münz who become some kind of surrogate parents to me and made Göttingen feel like a second home for me.

Furthermore, I especially want to thank Andreas Neubauer, who did not hesitate to offer me his proof-reading services after knowing me only for a few hours, and Stefan Menge for proof-reading this thesis and helping me improving it.

Moreover, I would like to thank Nikolaj Gebert who came into my life in the dark age of PhD thesis writing and lighted it up. Thanks for all the patience, his never-ending motivation, and the chocolate which let me survive. I am thankful for having him in my life.

Finally, my deepest gratitude goes to my parents who always supported me unconditionally – even when their little girl moved far far away to evil Göttingen. They always let me think that I can achieve whatever I want and they will always stay behind me. I love you and thank you more than words can say.

Contents

Acronyms

2-TBN 2-time-slice Bayesian Network.

AFM Additive Factor Model.

AST abstract syntax tree.

AUC area under the curce.

BKT Bayesian Knowledge Tracing.

BN Bayesian Network.

CKM conjunctive knowledge model.

CPT conditional probability table.

DBN Dynamic Bayesian Network.

DKT Deep Knowledge Tracing.

EDM Educational Data Mining.

HMM Hidden Markov Model.

IRT Item Response Theory.

ITS Intelligent Tutoring System.

KAM Knowledge Application Model.

KC knowledge component.

LFA Learning Factor Analysis.

MOOC massive open online course.

MRH most-recent holder.

MWH most-wanted holder.

PFA Performance Factor Analysis.

PP programming primitive.

PPM proportional model.

RMSE root mean square error.

RNN recursive neural network.

SAPP semantically augmented programming primitive.

List of Figures

List of Listings

List of Tables

1. Introduction

Nowadays, computers are indispensable from our daily life. Thanks to the digital revolution, automatization, robotics, and IT become essential parts of almost all branches of industry. Children born in last decades of the 20th century are called *digital natives* since they grow up with technology from the digital age [7]. In this digitized world, programming gains importance not only for software developers but also in other disciplines like, e.g., engineering, pharmacy, finance etc. German chancellor Angela Merkel declares programming as a key competency such as reading or writing [8].

In general, programming can be seen as a toolbox for solving problems, e.g., resource planning, automatization, analysis of large data. However, learning to program is a hard task since it requires different types of specialized knowledge. You do not only need knowledge about the programming language, its concepts and conventions, but also knowledge from the problem domain and general problem solving abilities to be able to develop a strategy for solving a problem [2]. Knowing how students develop programming knowledge and where they struggle, may help in the development of suitable teaching strategies.

Digitization already has a huge impact on education nowadays. Face-to-face teaching is often supported by e-learning systems but also pure distance learning emerges in the last years through massive open online courses (MOOCs), e.g., Coursera [9], edX [10], or Udacity [11]. Especially during the Covid-19 pandemic many schools and universities had to switch to online distance learning. Such systems allow for large data collection which is a “gold mine” [12] of educational data.

In this thesis, we want to use data from an online assessment system for open-ended programming exercises to analyze how a student’s programming knowledge and errors evolve over time by looking at his/her submitted source code. Since knowledge is not measurable directly, we apply techniques from student modeling to estimate a student’s knowledge based on his/her performance. Student models are one of the four basic components in Intelligent Tutoring Systems (ITSs).

A traditional ITS consists of four components [13] (see Figure 1.1):

- *Domain model*: Knowledge can be divided into several *knowledge components (KCs)*. Koedinger et al. [14] define KCs as “an acquired unit of cognitive function or structure that can be inferred from performance on a set of related tasks”. KCs represent units of knowledge, e.g., rules, concepts, principles, or facts, a student is expected to learn in a specific domain. In the domain model, KCs and their dependencies, as well as the corresponding learning materials, are defined. Often, it is considered as the “ideal” student model, but may also contain typical errors and misconceptions.

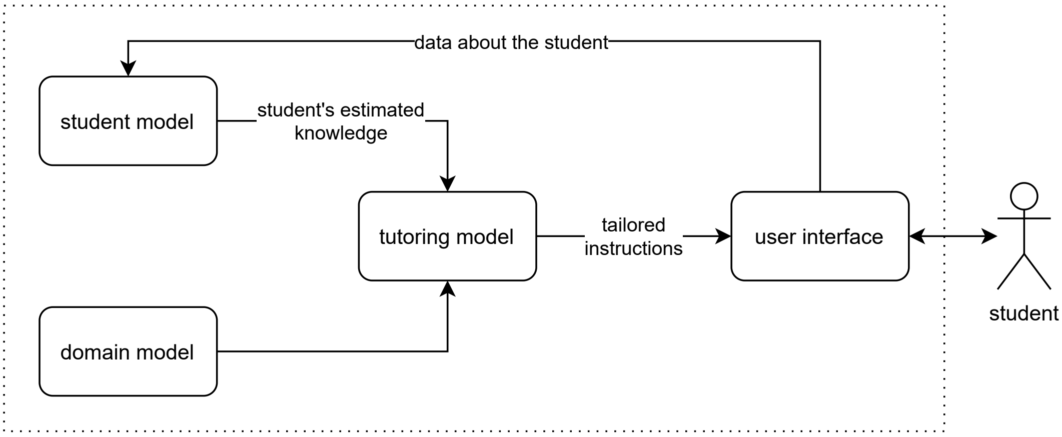


Figure 1.1.: Basic architecture of an ITS (adopted from [1])

- *Student model*: The student model stores information about each individual student. Typically, it contains student's current affective [15, 16, 17, 18, 19, 20], motivational [21, 16, 22] or cognitive [23, 20, 24, 25, 26] state which is inferred from the student's interaction with the ITS.
- *Tutoring model*: The tutoring model uses information about a student's current state to make decisions, e.g., which topic to present next, in which form, e.g., as video or text, or to select appropriate exercises. It allows to reflect each student's individual needs.
- *User interface*: The user interface is responsible for the interaction between the student and the system. It interprets the student's input, e.g., speech, typing, clicking, and produces output, e.g., text, videos, diagrams, questions.

A main task of the student model is the estimation of the student's current knowledge. According to Desmarais and Baker [27], we will refer to student models for the estimation of a student's knowledge as *skill models*. Within this thesis, we use two common skill models, Performance Factor Analysis (PFA) which is based on logistic regression and Dynamic Bayesian Networks (DBNs) which is based on Bayesian theory, to estimate a student's knowledge at each step during a programming beginners course. We evaluate in how well the models fit for the prediction of student's performance and fine-tune the models to meet the special requirements of the programming domain. We can then use the models to track the change of the student's knowledge over time and use the interpretability of the models to get further insights, e.g., about difficulty of certain KCs. Those skill models are implementations of the more general *overlay model* [28]. The overlay model represents the student's knowledge as a subset of the domain level, i.e., it describes which KCs or to what extent each KC has been learned by the student.

However, overlay models are not sufficient to model students cognitive state. Students often derive own incorrect rules or facts because of misconceptions such that the main assumption of overlay models, that students have incomplete but correct knowledge of the domain, is violated [29]. This means that also the student's "mal-knowledge" should be considered when looking at students' knowledge development. Therefore, we analyze the errors students make and how those change over time to identify critical and less severe misconceptions.

Educational Data Mining (EDM) is an emerging research area which uses data from educational settings to get insights into learners[1] and learning [30]. Ihantola et al. [31] categorized the research goals of EDM in programming into three categories: *student*, *environment*, and *programming*. The category *student* refers to information about a student individually, e.g., a student's ability and knowledge [32, 33, 34], behavior [35, 36, 37, 38], or drop-out risk [39, 40, 41], which can be used for personalization in adaptive systems or for timely interventions before students are left behind. *Environment* involves insights about the learning environment itself, e.g., the student's usage of an IDE or hints [42, 43], which can be used to improve the educational systems for a better learning experience. The last category refers to information about the learning process itself, e.g., how do students learn in general (not on the level of an individual student), which errors they make [44, 45, 46], or which patterns can be observed [47, 48]. The work of this thesis falls into the categories *student*, by analyzing properties of the programming domain and integrating them into skill models, and *programming* since we are investigating patterns of knowledge development and programming errors.

1.1. Scope of the Thesis

In this thesis, we want to investigate how students learn to program. We are looking at how their knowledge evolves over time and which errors they make. Programming knowledge is manifold, it contains reading, writing, testing and debugging code. In this thesis, we focus on code writing. We only consider KCs which can be directly derived from a student's source code, i.e., on the syntactic level in form of programming primitives and on the semantic level by looking at roles of variables. We do not include higher level knowledge, e.g., programming plans since they can often be determined by assembling variable roles. We also do not consider a student's problem solving ability since it is hard to derive a student's intention solely by code.

Basically, knowledge consists of two parts, things you learn and errors you make. Since learning is not directly measurable, we use skill models to estimate a student's learning progress. Therefore, the first research question is:

[1] Since we focus on university data in this thesis, we use the terms *learner* and *student* interchangeably throughout the thesis

- **RQ 1:** How can we construct a skill model for the programming domain?

We subdivide the question into several subquestions:

- **RQ 1.1:** How great is the difference between students' code?
- **RQ 1.2:** Which effect does meta-parametrization have on the prediction performance of skill models?
- **RQ 1.3:** How can we modify DBNs to reflect the properties of the programming domain?

The second research question deals with the errors students make in programming:

- **RQ 2:** Which errors do students make during programming?

Also this question leads to more detailed subquestions answered in this thesis, which are:

- **RQ 2.1:** What are the most common errors made by students?
- **RQ 2.2:** Which errors are hard/easy to fix?
- **RQ 2.3:** Which errors re-occur often?

Using the skill model from RQ 1 and the error landscape from RQ 2, we answer the following research question:

- **RQ 3** How does the programming knowledge of students change over time?

This question leads to the following more detailed subquestions, which we also answer in this thesis:

- **RQ 3.1:** Which KCs are hard/easy to learn?
- **RQ 3.2:** How do errors change over time?
- **RQ 3.3:** How do students differ regarding the errors made? Can we identify different error patterns?

Previous studies showed that different patterns in how students write their code can be observed (see Section 3.3). RQ 3.3 is a result from our hypothesis that we can also identify such patterns in programming errors.

1.2. Goals and Contributions

The work on this thesis extends the body of knowledge in the area of student modeling and EDM by the following contributions:

- The identification of meta-parameters for skill models in the programming domain and analysis of their importance in skill modeling.
- A formal extension of two common skill models, Additive Factor Model (AFM) and PFA, by the ability to reflect flexible KC-item mapping.
- A dynamic multi-skill DBN model which allows to model KC granularity as well as pre-requisite relationships between KCs, dynamically assign the required KCs for solving an item, and implicitly models the problem solving ability.
- A new representation of source code as Knowledge Application Model (KAM) and a code similarity measure based on the comparison of two KAMs.
- An analysis of the most common errors made in C programming.
- An analysis of learning curves for KCs based on a DBN.
- An investigation of common patterns in the development of made errors.
- A data set of student's programming solutions with manually validated and classified errors.
- A framework for the automatic assessment of programming exercises in C.

1.3. Impact

The results of this thesis as well as work that has been done to enable this work have been published in peer-reviewed conference proceedings:

- Ella Albrecht, Jens Grabowski,"Sometimes It's Just Sloppiness - Studying Students Programming Errors and Misconceptions", in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*,2020.
  Own contributions
  I am the lead author of this publication. All main contributions, analysis, and evaluations have been done by myself.
- Ella Albrecht, Fabian Gumz, Jens Grabowski, "Experiences in Introducing Blended Learning in an Introductory Programming Course", in *Proceedings of the 3rd European Conference of Software Engineering Education (ECSEE'18)*,2018.
  Own contributions
  I am the lead author of this publication. I did most of the writing, the complete implementation of SmartAPE as well as the analysis of the results. Fabian Gumz was responsible for the content of learning management system ILIAS and the description of his experiences with it.

- Ella Albrecht, “A Framework for the Estimation of Students’ Programming Abilities”, in *Proceedings of the 10th International Conference on Educational Data Mining (EDM’17)*, 2017.
  Own contributions
  This publication is a doctoral symposium paper where the idea of the doctoral project were presented. I am the single author of this paper and established all of the work on my own.
- Ella Albrecht, Jens Grabowski, “Towards a Framework for Mining Students’ Programming Assignments”, in 2016 IEEE Global Engineering Education Conference (EDUCON), 2016.
  Own contributions
  I am the lead author of this publication. All main contributions have been done by myself.

In addition, we have submitted a paper to a scientific journal which is currently under review:

- Ella Albrecht, Jens Grabowski, “How Meta-Parametrization Affects Prediction Performance in the Programming Domain”, *User Modeling and User-Adapted Interaction* (revised with minor revision).
  Own contributions
  I am the lead author of this publication. All main contributions, implentations, and analysis have been done by myself.

1.4. Structure of the Thesis

We assume that the reader is familiar with the basics of probability theory and machine learning techniques. Otherwise, we refer to [49] and [50] as a reminder. The thesis is structured as follows:

In **Chapter 2**, we lay the foundation for our thesis by providing background information about programming knowledge (Section 2.1). In Section 2.2, we introduce roles of variables. Since our code analysis is based on abstract syntax trees (ASTs), these are introduced in Section 2.3. To understand our modeling approach with DBNs, we provide an introduction to Bayesian theory and Bayesian networks in Section 2.4. Furthermore, we give an overview of common skill models in Section 2.5.

In **Chapter 3**, we put our work in relation to existing work by looking at research on common programming mistakes (Section 3.1), work about skill models which have already been used in programming (Section 3.2), and work focusing on the analysis of students’ knowledge evolution (Section 3.3). We summarize previous work and describe the novelty of our work in Section 3.4.

Chapter 4 deals with our approach for the estimation of programming knowledge. First, we present our approach for the identification of KCs in source code and introduce the KAM as a new representation for source code in form the KCs applied in Section 4.1. In Section 4.2, we define two types similarity metrics, one based on ASTs and one based on KAMs. We identify the special properties of the programming domain and show how we adjusted PFAs and DBNs to be able to cope with those properties in Section 4.3. In Section 4.4 we describe how we construct learning curves from the knowledge estimates of the DBN.

In **Chapter 5**, we present how we analyzed programming errors. First, we explain our manual inspection procedure and how we categorize programming errors in Section 5.1. Then, we define different metrics for errors in Section 5.2. In Section 5.3, we then describe how we use clustering to identify common patterns in error development.

In **Chapter 6**, we present the three case studies we performed. We first describe our data and the data collection process in Section 6.1. Then we describe the setup and provide the results of each case study. The case studies cover the evaluation of our skill models for the programming domain (Section 6.2), the analysis of programming errors (Section 6.3), and the analysis of how the programming knowledge and errors evolve over time (Section 6.4).

We discuss our findings in **Chapter 7** by providing answers to our research question in Section 7.1 and discussing the strengths and limitations of our approach in Section 7.2. Furthermore, we present potential threats to validity in Section 7.3.

Finally, we conclude the thesis in **Chapter 8** and provide an outlook on future work.

2. Foundations

In this chapter, we present the foundations of this thesis. First, we describe what is considered as programming knowledge by looking at mental models and roles of variables in Section 2.1. Since our code analysis is based on ASTs, we will then give a short introduction into ASTs and AST-differencing in Section 2.3. Furthermore, we have a closer look at skill models, because these are the main concepts that we use in this thesis, in Section 2.5. For a better understanding of skill models based on Bayesian theory and our approach using DBNs, we present the foundations of Bayesian Networks (BNs) in Section 2.4.

2.1. Programming Knowledge

Programming can be seen as a design task where programming primitives and instructions are assembled to a complete program which solves a specific problem [2]. Figure 2.1 shows that the programming task can be subdivided into several subtasks.

First, the problem needs to be understood. This requires knowledge in the particular problem domain, e.g., in order to calculate the prime numbers from 1 to 1000, one needs to have basic mathematical knowledge and to know the definition of a prime number. Then the programmer has to design or create a plan for a solution of the problem, i.e., decomposing the problem into subproblems and creating plans as a solution for those problems. This involves general knowledge about design strategies, but can also require algorithmic knowledge or the knowledge of a specific design language, e.g., UML. During coding, the design is transformed into code. Here, knowledge of the used programming language is essential, but also knowledge about coding conventions and of hardware features that may affect the software implementation.

Two fundamental programming activities are *composition* and *comprehension* [2]. Composition relates to *writing* a program. From knowing *what* has to be achieved one derives instructional steps *how* to achieve it. Comprehension works in the other direction: by having given a set of instructions, i.e., the code, we evaluate *what* the program tries to accomplish. Besides composition and comprehension, Shneiderman and Mayer [51] identify two further programming activities: *debugging* and *modification*. Debugging is often involved in composition and describes the process of locating defects in the program. Often the term is used for the combination of detecting (testing), locating (debugging) and removing (bug fixing) errors in the code.

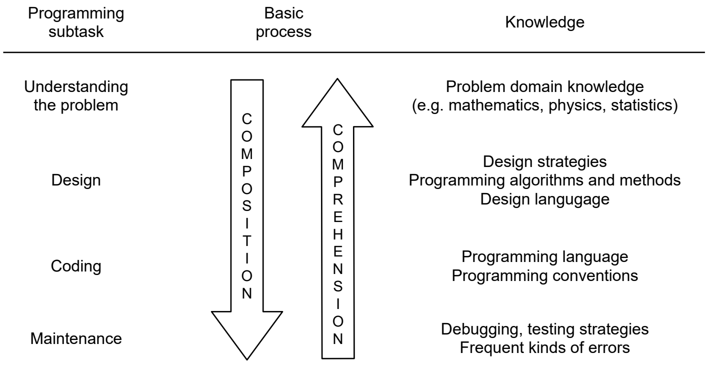


Figure 2.1.: Programming tasks (adopted from [2])

In cognitive sciences, it is basically distinguished between two types of knowledge: *declarative* and *procedural* [52] knowledge. Declarative knowledge describes the knowledge about facts, concepts, or principles. In contrast, procedural knowledge describes how declarative knowledge is used when solving problems. Shneiderman added a second dimension for programming knowledge. He distinguishes between *syntactic* and *semantic* knowledge [53]. Syntactic knowledge refers to the knowledge of the syntax of a particular programming language, e.g., that statements end with a semicolon or the names of built-in functions in C. It can be learned by memorization and has only a low cognitive demand. But, as easy as it can be learned, it is prone to forgetting. Semantic knowledge consists of general programming concepts which are independent of a concrete programming language and more resistant to forgetting. It can be organized hierarchically, ranging from low-level concepts, e.g., how to assign a value to a variable, over mid-level concepts, e.g., counting elements in an array, to high-level concepts, e.g., binary search or specific algorithms. Bayman and Mayer [54] refined the model by diving semantic knowledge into *conceptual* and *strategic* knowledge. Conceptual knowledge describes the knowledge about how certain concepts work, e.g., what happens when assigning a value to a variable or when looping. Strategic knowledge refers to composing syntactic and conceptual knowledge to solve novel problems.

Bertels et al. [3] proposed a cognitive model of programming knowledge which defines the "missing" link between syntactic and semantic knowledge. The model is organized hierarchically (see Figure 2.2):

- **Programming primitives (PPs)**: PPs make up the lowest level of the model. They are the basic building blocks of a program and refer to syntactic knowledge.
- **Semantically augmented programming primitives (SAPPs)**: Simply based on the

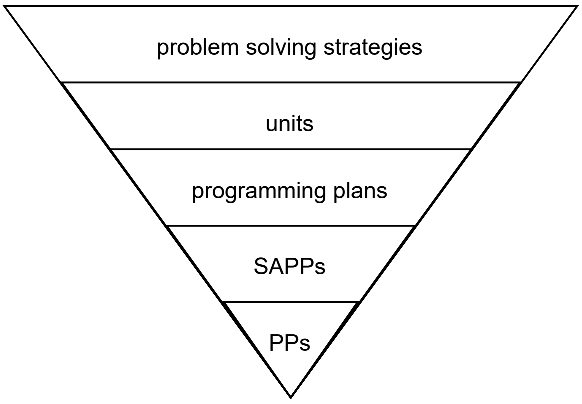


Figure 2.2.: Levels of programming knowledge (adopted from [3])

code level, SAPPs are equal to PPs. However they comprise much more information about their function, control and data flow. For example, instructions can be distinguished regarding their purpose like initialization, read instruction, or count instruction. But also variables can have different roles, e.g., a variable can be used as a counter or to hold the most recent value of a series of inputs. Roles of variables are discussed in more detail in Section 2.2.

- **Programming plans**: Programming plans describe how multiple instructions can be combined to achieve a certain goal [55]. Examples for plans are swapping values of two variables or finding the smallest element in an array.
- **Units**: Programming plans can be organized in so called units [56], e.g., counting the elements of an array or summing the elements of an array can be generalized in a unit under enumeration of an array.
- **Problem solving strategies**: Plans can be combined to solve a novel problem, e.g., a plan for calculating the sum of array values can be combined with counting the elements of an array to calculate the average, even if the knowledge of the average plan is not yet present. Strategies for this composition process [55] represent the highest level of programming knowledge.

2.2. Roles of Variables

Variable roles describe the stereotypical behavior of a variable, i.e., how the variable values change over time. Sajaniemi has shown that the following 11 roles cover 99% of variables used in novices' programs [57]. While the first 8 roles are applicable to all kinds of variables, the last 3 roles refer to more complex data structures like arrays or lists:

Fixed value A fixed value variable does not change its value after initialization, i.e., it is a read-only variable. The fixed value can be a constant value, the value of another variable or a calculation, or be read from an input.

Stepper A stepper is variable whose sequence of successive values is predictable. A stepper does not necessarily have to increment/decrement by 1, but can also be, e.g., a multiplication by a fixed factor. A stepper variable has to be updated in a loop. A stepper is also often used for loop control.

Counter A counter is variable which is incremented under certain conditions and is used for counting. A counter variable has to be updated in a loop and is protected by a guard.

Most-recent holder (MRH) The MRH is a variable storing the last value in a sequence of traversed values. Typically, a MRH is used to store the latest read value from input. A MRH has to be updated in a loop.

Most-wanted holder (MWH) The MWH stores the "best" value in a sequence of values gone through so far. There are no restrictions regarding the definition of "best". A typical example for a MWH is storing the smallest or biggest value. The MWH has to be updated in a loop and requires a guard which checks whether a better value was found.

Gatherer A gatherer accumulates the values gone through so far. A typical example for a gatherer is a variable that stores the sum of a sequence of values. A gatherer has to be updated in a loop and typically is initialized with 0.

Follower A follower always gets assigned the old value of another variable. A follower is typically used for comparison of succeeding values in an array or list.

One-way-flag A one-way flag is a boolean variable that once the value was changed after initialization keeps its value. Typically, the one-way flag is used for loop or branch control, e.g, check for valid input.

Temporary A temporary variable has only a short lifetime. It is initialized in loops or selections. Typically it is used to store intermediate results of calculations or for swapping values.

Organizer An organizer is a data structure where the elements are re-organized during runtime. A typical example would be an array used for sorting. It requires swapping of its elements.

Container A container is a data structure that allows storing multiple elements. Typically containers are arrays that store input values.

```
#include <stdio.h>
#define MAX 100

int main(void){
  int input[MAX], count = 0, int number;

  printf("Please enter up to %d integer values:\n", MAX);
  int check = 1;
  while (count < MAX && (check = scanf("%d", &number)) == 1) {
    input[count] = number;
    count++;
  }

  if (check != 0) {
    int i = 0;
    while (i < count) {
      int min = i, swap, j;
      j = i+1;
      while (j < count) {
        if (input[j] < input[min]) {
          min = j;
        }
        j=j+1;
      }
      swap = input[i];
      input[i] = input[min];
      input[min] = swap;
      i = i+1;
    }
  }
  else printf("Invalid input");
  return 0;
}
```

Listing 2.1: Example program

Walker A walker is used to traverse a data structure. A typical example for a walker is an iterator in Java. A walker is a special type of a stepper. As such it has to be updated in a loop and is often used for loop control.

Example 2.1 *Listing 2.1 shows an implementation of selection sort. Numbers are read from input and then sorted. We can identify 8 variables in the code. The variable* `number` *is a MRH, the variables* `i`*,* `j` *and* `count` *are steppers. The array* `input` *is an organizer since it is used for sorting the values stored in it. The variable* `check` *is a one-way flag. Once the input is invalid,* `check` *is set to 0 and the loop is stopped. The variable* `min` *is a MWH since it stores the minimum number found so far. The variable* `swap` *is a temporary variable since its value is discarded after each repetition of the loop.*

Variables can change their roles during their lifetime. We can distinguish between *proper* and *sporadic* changes [57]. A sporadic change occurs when the value of a variable is not

```
int i = 0;
while(scanf("%d",&i)){
  int sum = 0;
  while (i>0){
    sum += i;
    i––;
  }
  printf("The sum of the numbers from 1 to %d is %d", i, sum);
}
```

Listing 2.2: Example of a proper role change

required anymore and the variable is re-initialized for a different use, i.e., role. Sporadic changes are bad programming practice since they reduce readability of code. In proper role changes, the current value of a variable is used in the new role. For example, the variable `i` in Listing 2.2 is used as a MRH in the outer loop and changes its role to a stepper in the inner loop.

2.3. Abstract Syntax Trees

When compiling a program, the source code is translated into machine code. For this purpose the compiler needs to parse the code. Intermediate representations of the program help the compiler to gather information about the program which can be used for analyzing, translating, and optimizing the code [58].

2.3.1. Intermediate Representations of Code

Programming languages are defined by a context-free grammar.

Definition 2.1 (Context-free grammar [59]) *A* context-free grammar $\mathscr{G} = (S, T, NT, P)$ *is a quadruple where*

- *S is the start symbol;*
- *T is the set of terminal symbols;*
- *NT is the set of non-terminal symbols; and*
- *P is the set of production rules in the form of $NT \rightarrow (NT \cup T)^{+}$.*

A context-free grammar describes how sentences can be constructed. A parser uses the grammar to derive a parse tree from the code by starting at the start symbol S and looking which production rules are applicable. Non-terminal symbols structure the language, while terminal symbols define the basic building blocks of the language.

```
S –> stmt
stmt –> expr ;
| whileStmt ;
| ifStmt ;
| {stmtList}
expr –> ( expr )
| comparison
| assignment
| arithmetic
| identifier
| constant
whileStmt –> while ( expr ) stmt
| do stmt while ( expr );
ifStmt –> ...
stmtList –> stmt
| stmtList stmt
comparison –> expr compOp expr
compOp –> <
| >
| <=
| >=
| !=
| ==
```

Listing 2.3: Example grammar

Example 2.2 *Listing 2.3 shows an excerpt from the grammar for the C programming language. The parse tree for the code snippet* `while (i < 10) {...}` *is depicted in Figure 2.3.*

The parse tree contains each token that is present in the source code and gets quite large. However, much of the information is unnecessary for the analysis of the code. We do not exactly need to know how the syntax of a while loop declaration looks like. To reduce the size of the parse tree, unnecessary syntactical information is removed and nodes are combined. The resulting tree is called an *abstract syntax tree (AST).*

Definition 2.2 (Abstract syntax tree [60]) *An* abstract syntax tree *is an ordered tree $\mathscr{A} = (V, E)$ with*

- *V being the set of nodes $v = (l, \varphi)$ where $l \in \Sigma$ denotes the label in an alphabet Σ, $\varphi \in String \vee \varepsilon$ denotes the value of the node, and ε describes the empty String,*
- *E being the set of edges $e = (v, c, i)$ where c is the i-th child of v,*
- *$parent(v) \in V \cup \emptyset$ denotes the parent of each node $v \in V$,*
- *$root(\mathscr{A}) = v$ with $parent(v) = \emptyset$ denotes the root of the AST, and*
- *$children(v) \subseteq V \times \mathbb{N}$ denotes the ordered set of children (c, i) of node $v \in V$.*

Each node of an AST is associated with a label, which corresponds to the name of the production rule and describes the *node type*, and a value which is a String describing an

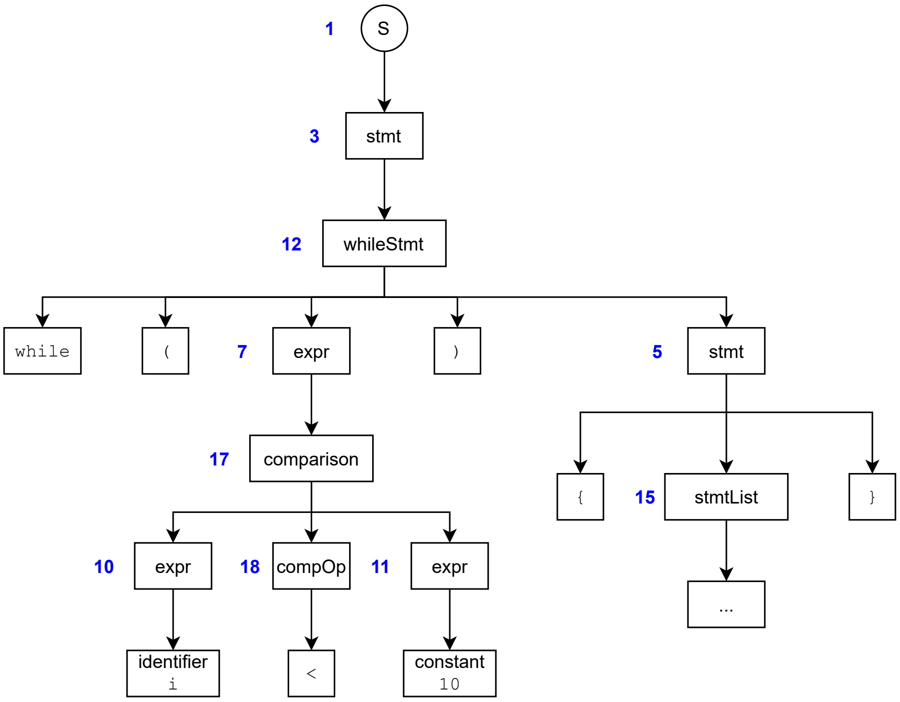


Figure 2.3.: Parse tree for `while(i<10){...}`. The numbers next to non-leaf nodes denote which rules from the grammar in listing 2.3 were applied at each step

operator, a name or any other actual token in the code. A node may also have no value which is then denoted by an empty String.

Example 2.3 *Figure 2.4 shows the AST corresponding to the parse tree from Figure 2.3. The AST is defined by the following set of nodes* $V = \{(S,\varepsilon),(while,\varepsilon),(comparison,<),(stmtList,\varepsilon),(identifier,i),(constant,10),\ldots\}$. *Pure syntactical tokens like braces, brackets, and keywords are removed. Instead of intermediate nodes like `expr` and `stmt` nodes are directly connected to the concrete type of expression, e.g., comparison, or statement, e.g., `whileStmt`. Operators are denoted as label directly in the expression to which they belong instead of an extra node, e.g., the operator `<` is denoted in the node `comparison`.*

Although ASTs contain much less nodes than parse trees, they still contain full information about the underlying source code.

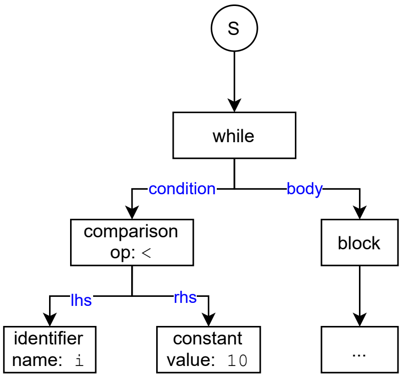


Figure 2.4.: AST for `while(i<10){...}`

2.3.2. AST Differencing

To determine the difference between two pieces of code on a fine-grained syntactical level, AST differencing can be used. AST differencing determines a sequence of actions to transform a source AST A into a target AST B. The sequence of actions is called an *edit script*. Falleri et al. [60] consider four edit actions in their GumTree algorithm:

- $update(v, \varphi_n)$: replacing the value of a node $v = (l, \varphi_{old})$ by a new value φ_n, such that $v = (l, \varphi_n)$.
- $add(v, v_p, i)$: adding a new node $v = (l, \varphi)$ to the AST such that $parent(V) = v_p$, i.e., v_p is the parent of v, and $(v, i) \in children(v_p)$, i.e., v is the i-th child of v_p. In case that $v_p = \emptyset$, v becomes the new root of the AST, $root(\mathscr{A}) = v$ and $children(v) = \{(root_{old}(\mathscr{A}), 0)\}$, i.e., the old root of the AST becomes a child of the newly added node.
- $delete(v)$: removing the node v from the AST, such that $(v, i) \notin children(parent_{old}(v))$ $\forall i \in \mathbb{N}$.
- $move(v, v_p, i)$ moving a subtree with v as root, such that $parent(v) = v_p$, $(v, i) \in children(v_p)$, and $(v, j) \notin children(parent_{old}(v))$ $\forall j \in \mathbb{N}$.

Removing and adding subtrees can be realized by subsequently removing or adding single nodes to the AST and are as such covered by the previous actions.

In general, AST differencing follows a two step process [60]:

1. *Mapping*: A mapping of nodes from AST A to nodes in AST B is generated. A mapping describes which node in the source AST corresponds to which node in the target AST. Each node can only be mapped to exactly one or zero other nodes and both nodes have to have the same label, i.e., being of the same node type. In the

GumTree algorithm [60], isomorphic subtrees are used in a top-down followed by a bottom-up approach to identify matches between nodes.

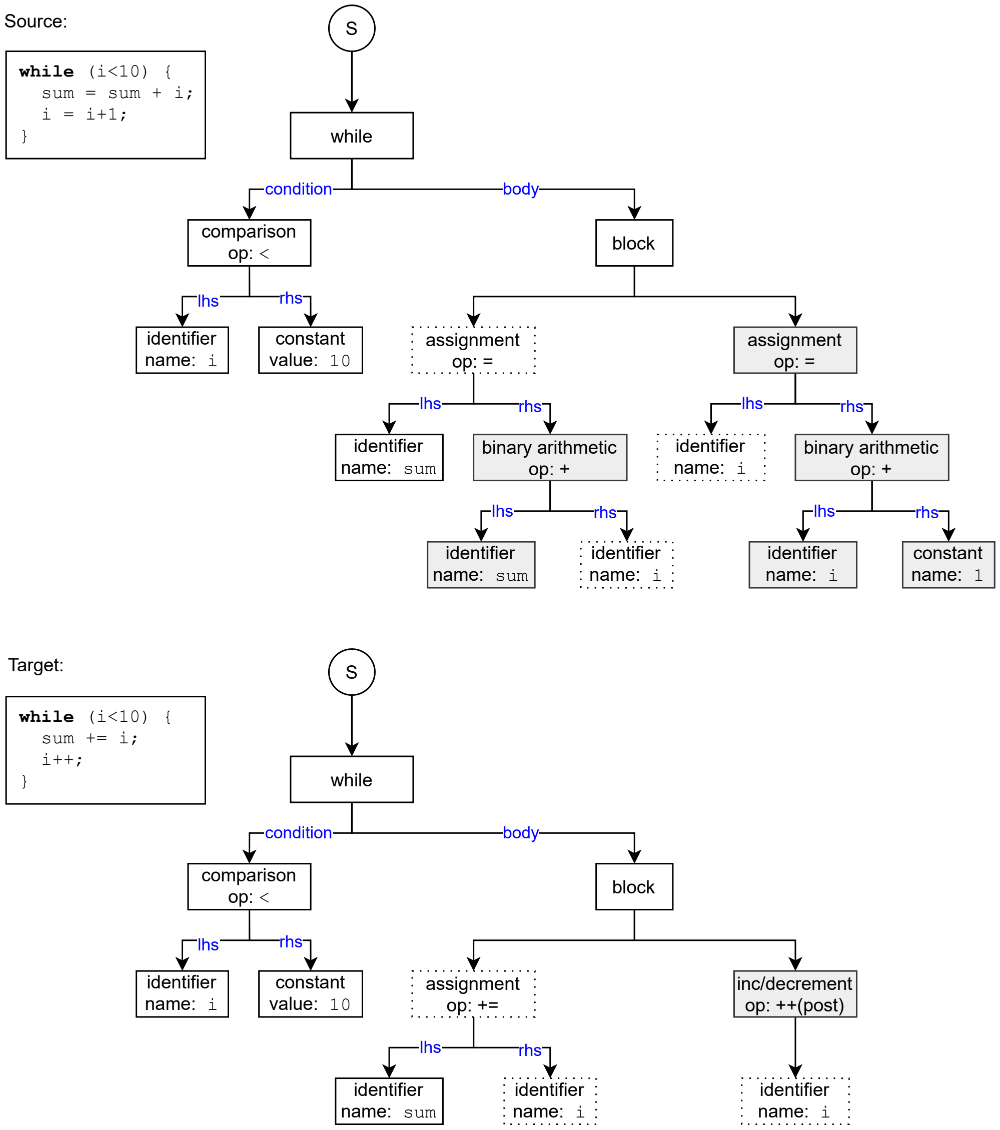
2. *Edit script generation*: The mapping from the first step is used to derive an edit script with as few actions as possible. The algorithm by Chawathe et al. [61] proceeds in 5 phases:
   - *Update phase*: For each pair of nodes in the mapping with distinct values, the value of the node is updated by the target value.
   - *Align phase*: A pair of nodes in the mapping is aligned when the children of these nodes are in the same order. If they are not aligned, a move operation is performed to align them.
   - *Insert phase*: For each unmatched node in the target AST with a matched parent, the unmatched node is added as child to the matched parent node.
   - *Move phase*: For matched nodes x in the source tree and y in the target tree with unmatched parents, the subtree of x is moved as a child to the node matched to $parent(y)$.
   - *Delete phase*: All remaining non-matched nodes are deleted.

Example 2.4 *Figure 2.5 shows the mapping between the nodes of two ASTs. The target program differs from the source program in such that it uses an add-assignment and increment instead of two assignments with an add-expression. The edit script for the ASTs would look as following:*

```
update((assignment,=),+=)
add((inc/decrement,++(post)),(block,eps),1)
move((identifier,i),(assignment,+=),1)
move((identifier,i),(inc/decrement,++(post)),0)
delete((identifier, sum))
delete((binary arithmetic,+))
delete((identifier,i))
delete((constant,1))
delete((binary arithmetic,+))
delete((assignment,=))
```

2.4. Bayesian Networks

In this section, we introduce the formalism of BNs and their generalization the DBNs. Furthermore, we have a look at two noisy models which are used to simplify certain probability distributions in BNs. BNs are probabilistic graphical models used to model uncertainty in knowledge, that is why they are also called *belief networks*.


Source:

```
while (i<10) {
  sum = sum + i;
  i = i+1;
}
```

Target:

```
while (i<10) {
  sum += i;
  i++;
}
```

Figure 2.5.: Comparison of two ASTs. The solid white boxes describe perfectly matched nodes, the dotted boxes correspond to matched nodes and the solid grey boxes correspond to unmatched nodes

Definition 2.3 (Bayesian Network ([62])) *A* Bayesian network *is a directed acyclic graph* $\mathscr{B} = (G, \Theta)$ *with*

- *the underlying graph* $G = (V, E)$ *with vertices* $V = \{X_1, ..., X_n\}$ *and edges* $E : V \times V$
- *the set of parameters* $\Theta = \{\theta_{X_i|pa(X_i)} = P_G(X_i|pa(X_i))|X_i \in V\}$ *where* $pa(X_i)$ *denotes the set of parents of the node* X_i*.*

In a BN each node represents a random variable X_i and an edge denotes a direct dependency between variables. A variable X_i is statistically independent of all its non-descendants in the BN. The parameters of a BN are conditional probability tables (CPTs) describing the probability of each realization of a variable conditioned on its parents. The BN can be used to efficiently encode a joint probability distribution over the random variables in V [62]:

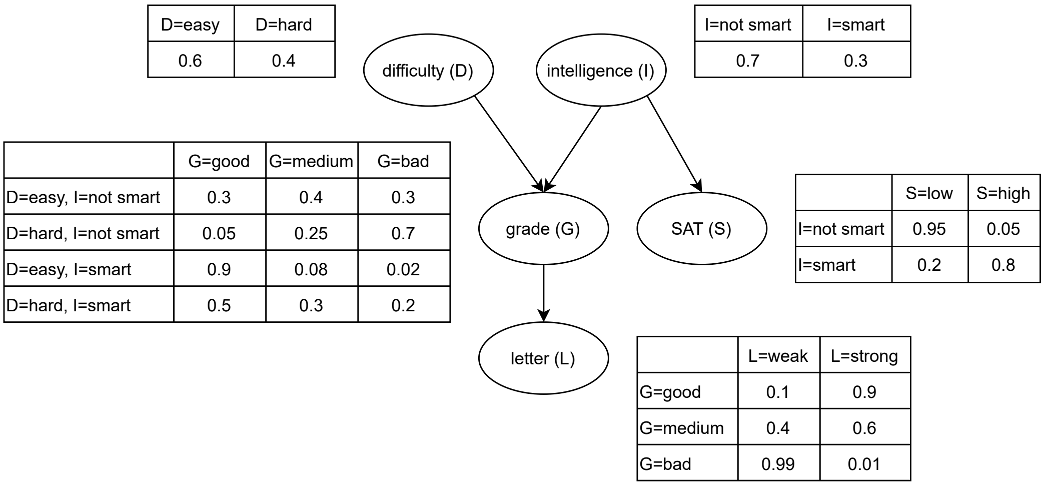
$$P_G(X_1, ..., X_n) = \prod_{i=1}^{n}(X_i|pa(X_i)) = \prod_{i=1}^{n}\theta_{X_i|pa(X_i)} \tag{2.4.1}$$

Example 2.5 *Figure 2.6 shows an example of a BN. It represents the dependencies between the five random variables* Difficulty, Intelligence, Grade, SAT, *and* Letter. *A student's grade depends on the course difficulty and his/her intelligence. The student's SAT score depends on his/her intelligence, and the quality of the recommendation letter depends on the grade. The difficulty of a course and a student's intelligence are independent of each other. A student's grade and his/her SAT score are independent given his/her intelligence.*

Each node is associated with a CPT. For example, the probability that a student gets a good grade in case that the course is easy and he/she is smart is 90% ($P(g^1|i^1, d^0)$*).*

Since the BN encodes a joint probability distribution, it can be used for reasoning and evaluating different inference queries using marginalization. Observing the value of a variable is called *evidence*. After observing one or multiple evidences our belief about other variables can be updated by inference, e.g., using the algorithm by Lauritzen and Spiegelhalter [63].

Example 2.6 *Consider the BN from Example 2.5. We are interested in the probability that a student achieves a high score in the SAT, i.e.,* $P(S = high)$*. Without knowing anything about the student or the course* $P(S = high) = 0.28$ *(see Figure 2.7a). After observing that the student gets a good grade, our belief can be updated with the new evidence* $P(G = good) = 1$ *resulting in the probability distribution presented in Figure 2.7b. The probability that the student is smart increases and as a consequence also the probability of a high SAT score increases to* $P(S = high) = 0.51$*.*



D=easy	D=hard
0.6	0.4

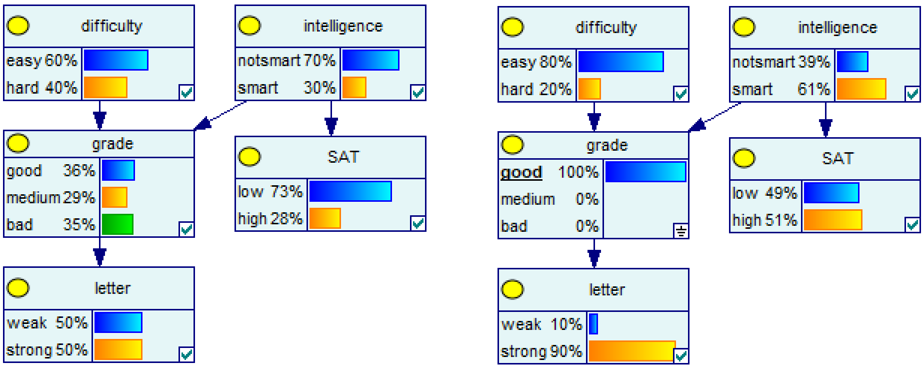
I=not smart	I=smart
0.7	0.3

	G=good	G=medium	G=bad
D=easy, I=not smart	0.3	0.4	0.3
D=hard, I=not smart	0.05	0.25	0.7
D=easy, I=smart	0.9	0.08	0.02
D=hard, I=smart	0.5	0.3	0.2

	S=low	S=high
I=not smart	0.95	0.05
I=smart	0.2	0.8

	L=weak	L=strong
G=good	0.1	0.9
G=medium	0.4	0.6
G=bad	0.99	0.01

Figure 2.6.: An example for a BN with CPTs (adopted from [4])



(a) Without evidence

(b) With evidence G=good

Figure 2.7.: Updating probabilities with evidence

2.4.1. Dynamic Bayesian Networks

DBNs [64] are a generalization of BNs and allow to model changes in the state of variables over time. The BN is replicated for each time step which is called a *time slice*. A node from one time slice can also be connected to nodes in the next time slice.

Definition 2.4 (2-time-slice Bayesian Network) *A* 2-time-slice Bayesian Network (2-TBN) *is a directed acyclic graph* $\mathscr{T} = (V, E_0, E_1, \Theta)$ *with*

- *the set of nodes* V*,*
- *the set of intra-time-slice edges* E_0*,*
- *the set of inter-time-slice edges* E_1*, and*
- *the set of parameters* Θ*.*

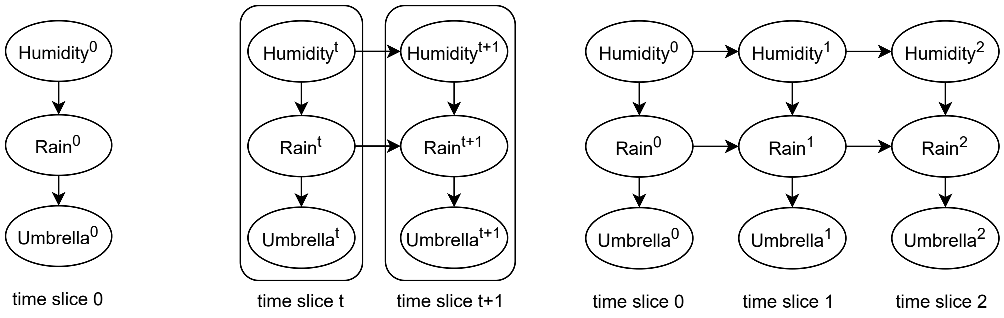
A 2-TBN encodes the time behavior of a BN. The nodes represent a random variable X_i at a time step t, which is denoted by X_i^t. A 2-TBN distinguishes between two types of edges, *intra-time-slice edges* which describe immediate dependencies between variables, i.e. dependencies within the same time slice, and *inter-time-slice edges* describe that a variable's value at time t depends on the value of an variable at time $t-1$. The parameters of 2-TBNs are CPTs as in BNs.

Definition 2.5 (Dynamic Bayesian Network [4]) *A* Dynamic Bayesian Network *is defined as* $\mathscr{D} = (\mathscr{B}_0, \mathscr{B}_\rightarrow)$ *where*

- $\mathscr{B}_0$ *is a BN representing the initial distribution over states and*
- $\mathscr{B}_\rightarrow$ *is a two-time slice BN.*

A DBN consists of a BN describing the network at time step 0 and a 2-TBN describing the network at each following time step and the transitions between time slices. A DBN can be transformed into a BN for a fixed number of steps by *unrolling* it, i.e., modeling each time slice explicitly.

Example 2.7 *Figure 2.8 shows an example of a DBN. It is defined by the BN* $\mathscr{B}_0$ *in Figure 2.8a and the 2-TBN* $\mathscr{B}_\rightarrow$ *in Figure 2.8b with intra-time-slice edges* $Humidity^t \rightarrow Rain^t$ *and* $Rain^t \rightarrow Umbrella^t$ *and inter-time slice edges* $Humidity^t \rightarrow Humidity^{t+1}$ *and* $Rain^t \rightarrow Rain^{t+1}$*. This means that whether it rains tomorrow does not only depend on tomorrow's humidity but also on whether it is raining today. Figure 2.8c shows the corresponding unrolled BN for 3 time steps.*



(a) Initial network $\mathscr{B}_0$ (b) 2-TBN $\mathscr{B}_{\rightarrow}$ (c) Unrolled BN for 3 time steps

Figure 2.8.: An example of a DBN and its composition

2.4.2. Noisy-OR/Noisy-AND Gates

A major problem in applying BNs is the effort for model building. Structure as well as parameters have to be defined either by experts or learned from data [65, 66, 67]. A CPT for a node X requires the specification of $2^{pa(X)}$ parameters, i.e., the number of parameters grows exponentially with the number of parents. But, more parameters mean that more data is required to guarantee good results in parameter learning. In small data sets, many conditioning cases are represented only by few or even no data such that the learned CPTs are partially uninformative.

Consider, we have a binary variable Y with binary parent variables $X_1,...,X_k$. To make $Y = true$ it would be sufficient if at least one of the parent variables is *true*. Such dependencies can be modeled using a *noisy-OR* gate.

Definition 2.6 (Noisy-OR Gate [4]) *Let Y be a binary-valued random variable with binary-valued parents $pa(Y) = \{X_1,...,X_k\}$. The CPT of a* noisy-OR *is defined by:*

$$P(Y=0|X_1,...X_k) = (1-\lambda_0)\prod_{i:X_i=1}(1-\lambda_i)$$

$$P(Y=1|X_1,...X_k) = 1-((1-\lambda_0)\prod_{i:X_i=1}(1-\lambda_i))$$

where λ_i for $i \in \{0,...,k\}$ are the noise parameters.

Instead of 2^k parameters only $k+1$ parameters are required to fully specify a CPT when using a noisy-OR gate. The parameter λ_i describes the probability that $Y = 1$ in case that $X_i = 1$ and all other parent variables being 0. The parameter λ_0 is called *leakage* and describes the probability of Y being *true* although all its parents are *false*. Table 2.1 shows the full CPT for a noisy-OR node Y with two parents X_1 and X_2.

X_1	X_2	$Y=0$	$Y=1$
0	0	$1-\lambda_0$	λ_0
1	0	$(1-\lambda_0)(1-\lambda_1)$	$1-((1-\lambda_0)(1-\lambda_1))$
0	1	$(1-\lambda_0)(1-\lambda_2)$	$1-((1-\lambda_0)(1-\lambda_2)$
1	1	$(1-\lambda_0)(1-\lambda_1)(1-\lambda_2)$	$1-((1-\lambda_0)(1-\lambda_1)(1-\lambda_2))$

(a) With leakage λ_0

X_1	X_2	$Y=0$	$Y=1$
0	0	1	0
1	0	$1-\lambda_1$	λ_1
0	1	$1-\lambda_2$	λ_2
1	1	$(1-\lambda_1)(1-\lambda_2)$	$1-((1-\lambda_1)(1-\lambda_2))$

(b) Without leakage, i.e., $\lambda_0 = 0$

Table 2.1.: CPT of a noisy-OR with two parent nodes

On the other hand, we might have a logical AND-dependency of parents. Using DeMorgan's law $X_1 \wedge X_2 = \overline{\overline{X_1} \vee \overline{X_2}}$, a noisy-OR can be transformed into a *noisy-AND*.

Definition 2.7 (Noisy-AND Gate) *Let Y be a binary-valued random variable with binary-valued parents $pa(Y) = \{X_1, ..., X_k\}$. The CPT of a* noisy-AND *is defined by:*

$$P(Y=0|X_1,...X_k) = 1-((1-\lambda_0)\prod_{i:X_i=0}(1-\lambda_i))$$

$$P(Y=1|X_1,...X_k) = (1-\lambda_0)\prod_{i:X_i=0}(1-\lambda_i)$$

where λ_i for $i \in \{0,...,k\}$ are the noise parameters.

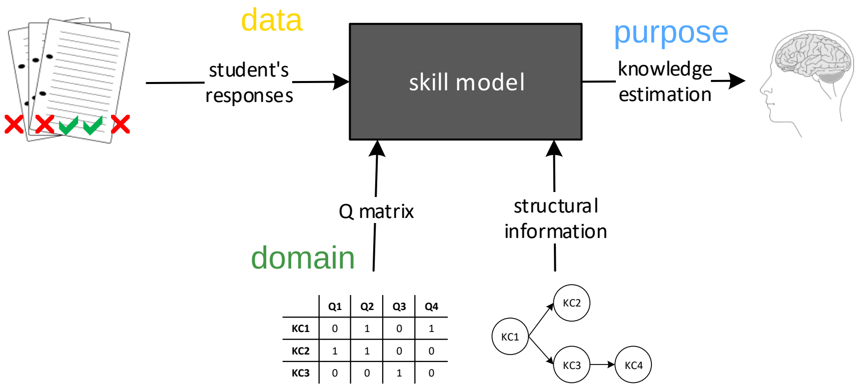
The noise parameter λ_i describes the probability of $Y=0$ in case that $X_i = 0$ and all other parent variables being 1. Table 2.2 shows the full CPT of a noisy-AND node Y with two parents X_1 and X_2.

2.5. Skill Models

Figure 2.9 depicts the general structure of skill models. A Q matrix and other structural information, e.g., dependencies between KCs, define the *domain model*. A Q matrix defines for each learning item, i.e., question or exercise, which KCs are required to answer the learning item correctly. Usually, a skill model uses the domain model as background knowledge for the inference of a student's current knowledge state. The knowledge of a

X_1	X_2	$Y=0$	$Y=1$
0	0	$1-((1-\lambda_0)(1-\lambda_1)(1-\lambda_2))$	$(1-\lambda_0)(1-\lambda_1)(1-\lambda_2)$
1	0	$1-((1-\lambda_0)(1-\lambda_2)$	$(1-\lambda_0)(1-\lambda_2)$
0	1	$1-((1-\lambda_0)(1-\lambda_1))$	$(1-\lambda_0)(1-\lambda_1)$
1	1	λ_0	$1-\lambda_0$

Table 2.2.: CPT of a noisy-AND with two parent nodes



	Q1	Q2	Q3	Q4
KC1	0	1	0	1
KC2	1	1	0	0
KC3	0	0	1	0

Figure 2.9.: General scheme of a skill model

student is estimated from *data* by looking at the sequence of his/her responses to the learning items. At each *step* it is evaluated which KCs were applied correctly and which were applied incorrectly at that step. A step usually corresponds to a learning item. But in case that the student is allowed to work more than once on an item, a step can also be defined on a submission or even keystroke level. The current knowledge state is often expressed as the probability that the student "knows" a certain KC or the probability that the student will solve a particular learning item correctly and depends on the concrete *purpose* of the skill model.

In this section, we present different skill models based on Bayesian theory (Section 2.5.1) and logistic regression (Section 2.5.2) as well as models which comprise both techniques (Section 2.5.3).

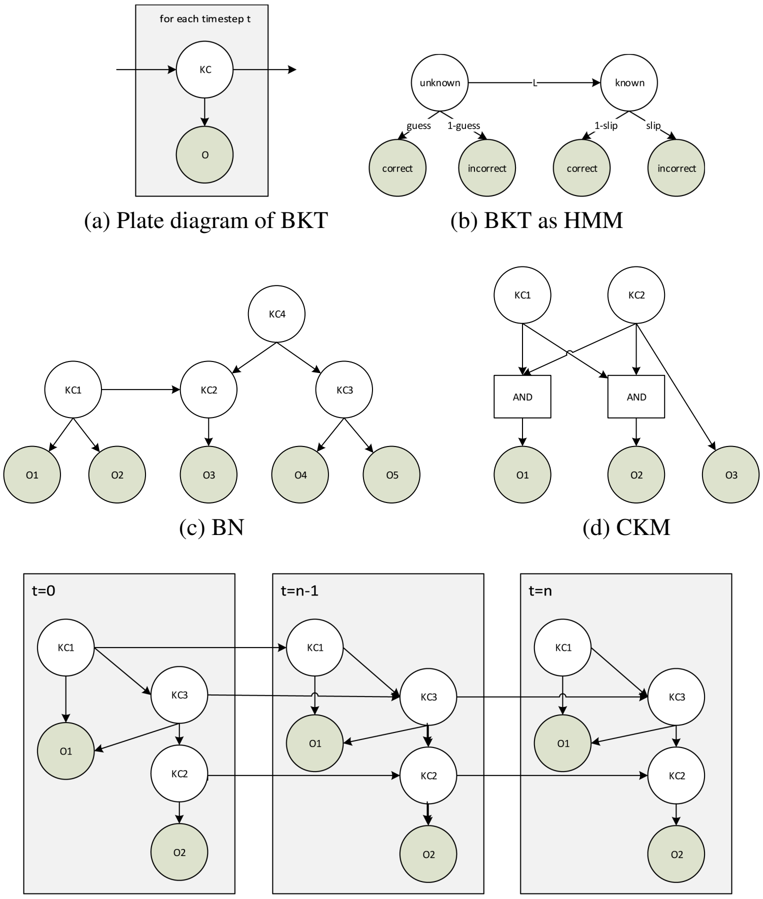
2.5.1. Models Based on Bayesian Theory

One of the most famous skill modeling approaches is Bayesian Knowledge Tracing (BKT) [68]. BKT uses a Hidden Markov Model to model a student's knowledge of a certain KC (see Figure 2.10b). The current knowledge state of a KC is modeled by a binary hidden state, i.e., the KC can be either *known* or *unknown*. The student can switch from the state *unknown* to the state *known* with a transition probability L, often also referred to as learning rate. In classical BKT, a transition from *known* to *unknown* is not possible, i.e., there is no forgetting. The knowledge state is estimated by observing the student's responses to questions. A student's answer to an item may be correct although the student is in the *unknown* state, i.e., the student can *guess* the answer. On the other hand, the student can answer to an item incorrectly although the student is in the *known* state, i.e., the student can *slip*. Besides the learning rate, the guess and slip probabilities, BKT has a fourth parameter L_0 which describes the probability that the student already knows the KC before any practice opportunity. A lot of extensions were done to extend classic BKT, e.g., the integration of item difficulty [69] or the contextualization of estimates for the guessing and slipping parameters [70]. However, BKT observes all KCs separately. It does not consider multiple KCs in one observation or the dependencies between KCs.

BNs [71] are graphical models that encode causal relationships between variables, i.e., objects of interest or observations (see Section 2.4). In skill models, nodes in a BN represent KCs and responses to exercises. The edges between the nodes describe the interdependencies between KCs or which KC leads to which observation (see Figure 2.10c). The nodes can be either in the state *known* or *unknown* for KCs or *correct* or *incorrect* for observations of responses. The disadvantage of BNs is that the number of parameters increases rapidly the more dependencies they contain. Furthermore, they only model a static point in time and not the complete evolution over time.

When using DBNs (see Section 2.4.1) in student modeling, KCs are modeled by nodes which states not only depend on the observation nodes in the current time step but also on the KC's state in the previous time step. As DBNs are special types of BNs, they have the same problem of a high computational complexity.

A special BN which aims at allowing multiple KCs per item in BKT is the conjunctive knowledge model (CKM) [72]. The CKM assumes a conjunctive relationship between KCs, i.e., that all KCs that are required for the solution of an item have to be known in order to answer the item correctly. To model the conjunctive relationships noisy-AND gates (see Section 2.4.2) are used instead of complete CPTs (see Figure 2.10d). Noisy-AND gates are defined by only two parameters *guess* and *slip* which determine the probability to answer a learning item correctly although at least one KC is unknown resp. to answer a learning item incorrectly although all required KCs are known. This drastically reduces complexity of the BN when a lot of KCs are involved.



(a) Plate diagram of BKT

(b) BKT as HMM

(c) BN

(d) CKM

(e) DBN

Figure 2.10.: Bayesian skill models. White nodes represent hidden nodes, i.e., they can not be observed directly.

2.5.2. Logistic Regression Models

Other skill models are not based on Bayesian theory but are regression models. They are slight variations of the Item Response Theory (IRT) [73]. The Learning Factor Analysis (LFA) [74] can be used to calculate the probability p that a student i will solve a particular exercise j correctly. The LFA model is defined as follows:

$$ln\left(\frac{p}{1-p}\right) = \alpha_i + \sum_{k \in KCs(j)} \beta_k + \gamma_k n_{ik} \tag{2.5.1}$$

If we want to estimate the probability p that a student i will solve a particular exercise correctly, we consider the number of his/her previous practice opportunities n_{ik}. The counter is weighted by a parameter γ_k to describe how large the influence of the previous attempts on the current knowledge is. A parameter β_k describes the general difficulty of a KC k. The parameter α_i captures a student's ability. The parameters α_i, β_k, and γ_k are estimated from data by logistic regression.
The AFM [75] extends LFA by an additional parameter δ_j which reflects the difficulty of a learning item:

$$ln\left(\frac{p}{1-p}\right) = \alpha_i + \delta_j + \sum_{k \in KCs(j)} \beta_k + \gamma_k n_{ik} \tag{2.5.2}$$

Performance Factor Analysis (PFA) [76] re-configures the LFA model by focusing on "the strongest indicator of student learning: performance" [76]. In PFA, the student's ability parameter is dropped. Observations of previous attempts are split into success or failure. Instead of only counting how often a KC was applied, it is distinguished between the number of correct applications s_{ik} and the number of incorrect applications f_{ik} of a KC k. Similar to the previous counter n_{ik}, these counters are weighted by parameters γ_k for s_{ik} resp. ρ_k for f_{ik}:

$$ln\left(\frac{p}{1-p}\right) = \sum_{k \in KCs(j)} \beta_k + \gamma_k s_{ik} + \rho_k f_{ik} \tag{2.5.3}$$

While parameters of skill models based on regression can be learned relatively fast even for large numbers of parameters, their main drawback is that they do not incorporate dependencies between KCs.

2.5.3. Other Skill Models

There are also some attempts to combine classic BKT with logistic regression. LR-DBN [77] adds an additional layer of observable nodes to BKT which represent whether a certain sub-skill is required for the solution of a certain learning item or not. Instead of a fixed value for the transition probability from the *unknown* to the *known* state, the transition probability is determined by a logistic regression over all sub-skills. LR-DBN only focuses on sub-skills as features for transition probability. Feature Aware Student knowledge Tracing (FAST) [78] allows for general features for transition as well as emission probabilities. This allows to include, e.g., student or item indicator features.

In Deep Knowledge Tracing (DKT) [79] recurrent neural networks are used to model students learning. The input is a one-hot encoded vector which describes which skill was applied and whether it was applied correctly. The output vector describes for each learning item the probability that the student will solve the item correctly. The main disadvantage of DKT is that the model is not interpretable. The parameters that are fit are weighting matrices without any educational meaning.

Table 2.3 gives an overview of common skill models, their parameters, their capabilities, and their disadvantages.

skill model	multi KCs	KC count	dependencies	parameters	output	disadvantages
BKT	–	binary	–	slip, guess, learning rate, initial knowledge	probability that student is in a certain knowledge state	single-KC model, high computational complexity
BN	√	binary	√	CPTs for each node	probability for each KC being in a certain state	static model
DBN	√	binary	√	CPTs for each node	probability for each KC being in a certain state	high computational complexity
CKM	√	binary	–	slip, guess	probability for each KC being in a certain state	no dependencies considered
LFA	√	multiple	–	student ability, KC difficulty, influence of previous attempts	probability that a student will answer to a certain learning item correctly	no dependencies considered
AFM	√	multiple	–	student ability, learning item difficulty, KC difficulty, influence of previous attempts	probability that a student will answer to a certain learning item correctly	no dependencies considered
PFA	√	multiple	–	KC difficulty, influence of previous successes, influence of previous failures	probability that a student will answer to a certain learning item correctly	no dependencies considered
LR-DBN	sub-skills	binary	sub-skills	slip, guess, initial knowledge, a sub-skill's contribution to learning rate	probability that student is in a certain knowledge state	only sub-skill relationships possible
FAST	sub-skills	binary	sub-skills	emission and transition probabilities	probability that student is in a certain knowledge state	only sub-skill relationships possible
DKT	–	binary	–	input weight matrix, recurrent weight matrix, initial state, readout weight matrix	probability that a student will answer to a learning item correctly	no interpretable parameters

Table 2.3.: Overview of common skill models

3. Related Work

In this thesis, we perform a broad analysis of students' programming knowledge. We have a look at students' errors and use skill models to estimate their knowledge of programming constructs. Thus, we first start by giving an overview of related work dealing with the analysis of students' programming errors. Afterwards, we have a look on prior applications of skill models in the programming domain. Since we also want to analyze the development of programming knowledge over time, we complete our related work with reporting about the state of the art regarding the identification of common behavioral patterns in programming. In the end, we put the identified related work into context with our work and define our research delta.

3.1. Programming Errors

As broad as the variety of programming errors made by students is also the number of different classifications of errors [6]. Table 3.1 lists different categorizations of programming errors which are *complete* in the sense that an error can always be assigned to at least one of the categories. Further categories of programming errors can be found in Appendix B.

Two common categorizations base on on *which programmatical level* and *when* the error occurs. The first one divides errors into *syntax*, *semantic*, and *logic* errors [80]. Syntax errors refer to violations of the syntax resp. grammar of the programming language (see Section 2.3), e.g., a missing semicolon at the end of a statement, or imbalanced parenthesis. Semantic errors are related to an improper use of a programming construct, e.g., a missing initialization of a variable. Logic errors refer to problems during problem-solving, e.g., a wrong calculation. The second categorization distinguishes between *compile-time* and *run-time errors*. Compile-time errors are found during compilation and correspond to syntax and static semantic errors. Run-time errors occur when the program is executed. Run-time errors can be further sub-divided into errors which make a program crash, e.g, a null pointer exception or array index out of bounds, and *failures*, i.e., errors where the program does not behave as intended resp. specified, e.g., an error in a calculation. Failures are typically revealed by testing.

In general, a piece of code can be *missing*, *spurious*, *misplaced*, or *malformed* [81]. The concrete manifestation of the error can be seen in a context, e.g., input/output, initialization, declaration. For example, a missing output refers to a mismatch between implementation and specification which expects an output. A spurious output means, that an output is imple-

	Error	**Description**
	Compile-time errors	Errors arising during compilation
	Run-time errors	Errors only occurring at run-time
Hristova et al. (2003) [80]	Syntax Errors	Incorrect syntax of a programming language
	Semantic Errors	Improper use of programming constructs
	Logic Errors	Errors due to not respecting specification
Johnson et al. (1983) [81]	Missing	Omitting required program element
	Spurious	Including unnecessary program element
	Misplaced	Putting necessary program element in wrong place
	Malformed	Putting incorrect program element in right place
Spohrer and Soloway (1986) [82]	Construct-based problem	Problems related to semantics of language constructs
	Plan composition problem	Problems with putting parts of program together
Zehetmeier et al. (2015) [83]	Mental typo	Sloppiness
	Knowledge gap	Not knowing definitions or terms
	Misconception	Faulty understanding of a construct/concept
	Wrong choice	Inappropriate selection of solution process in a given setting
	Structural blindness	Inability to structure components in a given setting
	Quality gap	Inability to stick to quality standards
	Lack of innovation	Inability to construct a new solution from previous knowledge in a new context

Table 3.1.: Categories of programming errors

mented when it was not expected to. A misplaced output error could be, e.g., implemented inside a loop instead of outside. A malformed output would be, e.g., ignoring a specified output format.

Spohrer and Soloway [82] analyzed students' syntactically correct programs and distinguish between two basic error categories: *construct-based* and *plan composition* problems. Construct-based problems refer to difficulties students have with certain programming constructs and their semantics, e.g., confusing *and* and *or*. Plan composition problems describe difficulties students have when putting parts of a program together to achieve a certain goal, e.g., not considering uncommon, unlikely, or boundary cases.

The revised Bloom's taxonomy [84] describes cognitive competencies on six different skill levels which are also called the *cognitive process dimension*. Zehetmeier et al. [83] define error categories related to the cognitive process dimensions (see Table 3.2). The first level *remember* refers to retrieving knowledge from long-term memory. So the inability to remember, how to access a specific element in an array, can be seen as a *knowledge gap*. The second level *understand* means not only remembering the syntax of a certain construct but also understanding its semantics. A lack in understanding relates to a *misconception*. For example, a student may know how to write a do-while loop, but is not aware that the loop is always executed at least once in contrast to a while loop. *Apply* on the third level means, that the student is able to apply his/her knowledge in the correct context, i.e., he/she is able to select the correct construct for his/her intended purpose, e.g., selecting the data structure

Level	**Cognitive process** [84]	**Error class** [83]
0	–	Mental typo
1	Remember	Knowledge gap
2	Understand	Misconception
3	Apply	Wrong choice
4	Analyze	Structural blindness
5	Evaluate	Quality gap
6	Create	Lack of innovation

Table 3.2.: Relationship between revised Bloom's taxonomy and the error categories by Zehetmeier et al.

list instead of set when the order of elements matters and duplicates of elements shall be allowed. An error on this level falls into the category of *wrong choice*. Level 4 *analyze* deals with a student's ability to analyze a problem, break it down into smaller parts and analyze their relationships. The corresponding error category is called *structural blindness* and is comparable to Spohrer and Soloway's [82] plan composition problem. While the student is able to solve the individual sub-problems, he/she has problems with putting them together such that the overall problem is solved. Level 5 deals with the *evaluation*, i.e., a student shall be able to critically analyze and test his/her program. Errors on this level correspond to a *quality gap* and are related to good programming practices and testing, e.g., errors as a result of "spaghetti" code or the usage of `goto` statements. The highest level in the revised Bloom's taxonomy is *apply* where the student shall be able to apply his/her previous knowledge to new, unseen problems. For example, if a student was able to calculate the sum of the numeric elements in the array by using the array subscript, he/she should also be able to achieve the same goal by using pointer arithmetic after learning about the relationship between pointers and arrays. The lack of that ability refers to the corresponding error class *lack of innovation*. Additional to the given six level of the Bloom's taxonomy, Zehetmeier et al. defined a level 0 *mental typo* which refers to errors due to a lack in concentration, e.g., typos, missing semicolons etc.

In most studies regarding the most common errors made by students, compiler messages were analyzed to identify common programming errors since they can be easily evaluated automatically [85, 46, 86, 87]. However McCall and Kölling [88] showed that diagnostic messages and actual compiler errors often do not fit together and depend on the context by inspecting source code manually. Hence, only considering diagnostic messages to get an insight into programming errors is not sufficient.

Brown and Altadmri [85] developed a post-lexing and extension for compilation for the detection of common errors based on a survey conducted by Hristova et al. among computer science professors and students [80].

Spohrer and Soloway [82] showed that most errors students make are not not due to misconceptions about language constructs but a result of plan composition problems by inspecting 152 solutions for three exercises in total and comparing in how far the plans of the students deviate from the expected plans.

Ettles et al. [89] classified logic error into three categories: algorithmic, misconception, and misinterpretation. They used the assumption that same logic errors result in same failed test cases to group solutions inspecting them manually. They restricted their studies to errors that occurred at least in 5% of the time or more than 50 times. However, their exercises basically focused on variables, control structures, and arrays.

3.2. Skill Models in the Programming Domain

Although there exists a great variety of student modeling approaches, only few of them are applied to the programming domain, especially to open-ended programming exercises. Table 3.3 gives an overview of previous work of skill models in the programming domain and which meta-parameters (see Section 4.3.1) were used by the authors.

Corbett and Anderson [68] use BKT for the estimation of programming knowledge in the ACT Programming Tutor for LISP. The KCs in this approach are rules of the form "to achieve goal X, do Y". The order in which the rules should be applied for each exercise is deterministic, thus applying a rule in the wrong order means that the KC corresponding to that rule was not applied successfully. The estimated knowledge state describes the probability that a student knows a certain rule. However, in open-ended programming exercises it is unnatural to pre-define a concrete solution path. Students should be relatively free in the selection of KCs and the order of statements they want to use to achieve a goal.

Mayo and Mitrovic [90] use a constrained-based model (CBM) [91] in their SQL-Tutor for student modeling. Constraints define conditions that have to be satisfied by every correct solution. A constraint consists of a pair of conditions (C_r, Cs) where C_r is the relevance condition and C_s is the satisfaction condition. If the relevance condition is true in a student's solution then the satisfaction condition also has to be true, otherwise the student's solution is incorrect. To estimate a student's mastery of a KC, i.e., constraint, they use a simple BN for each KC which consists of four nodes: $RelevantIS_{c,p}$ describes whether a KC c is relevant in the ideal solution to learning item p, $RelevantSS_{c,p}$ describes whether the KC is relevant in the student's solution and depends on the relevance in the ideal solution, $Mastered_c$ describes whether the student has mastered the KC, and $Performance_{c,p}$ describes whether the constrained was satisfied and is conditionally dependent on $RelevantSS_{c,p}$ and $Mastered_c$. While $RelevantIS_c, p$, $RelevantSS_{c,p}$, and $Mastered_c$ are binary variables, $Performance_p$ can take one of three values: *satisfied*, *violated*, *not-relevant*.

Kasurinen and Nikula [33] applied BKT to python exercises. Their KCs were particular programming structures, e.g., files or loops. They defined a set of rules which describe guidelines for a preferred solution. Each guideline was associated with a certain programming structure. For example, the rule that each open file should be closed relates to the KC *file*. They checked if the guidelines were followed by means of structural analysis. If they were followed, the KC was assumed as applied correctly. Overall, they provided a very limited set of guidelines and KCs.

Gonzales-Brenes and Huang [78] evaluate how features like, e.g., multiple sub-skills, item indicators, or practice opportunities affect prediction performance. They used data from JavaGuide [92] to fit BKT, PFA, LR-DBN, and FAST models. In JavaGuide, students have to answer which output a piece of code produces or which value a variable will have. KCs are Java concepts and JavaParser [93], an AST-based indexing tool, determines which KCs are required to solve a particular exercise. If the student answers correctly, all required KCs are assumed as applied correctly, otherwise they are assumed as applied incorrectly. For training and testing each attempt of a student was considered as step. However, these types of exercises only evaluate whether students have the ability to understand code but not if they can actually write code.

Huang et al. [72] also used the data from JavaGuide for fitting a skill model. They extended CKM by further layers of hidden nodes that represent combinations of KCs resp. integration skills [94] and their mastery. Since CKMs make it difficult to model multiple attempts on the same learning item, they used the average success rate over attempts. This allows them to consider the complete history of attempts with only one step for each item.

Berges and Hubwieser [32] used IRT to assess coding abilities from source code. They identified 21 KCs covering the concepts of object-oriented programming, e.g., array, assignment, or overloading. For each KC one or more observable items were assigned, e.g., the items “Are there arrays with pre-initialization declared in the code” or “Is there any access of the elements of an array in the code” are related to the KC *array*. Each item can be answered by yes or no to evaluate whether the related KC was applied correctly or not. Since they only had one large assignment, they only modeled a student’s knowledge at a point in time but no learning.

Similar to our approach, Yudelson et al. [34] used programming language elements which were obtained by means of an AST parser as KCs for Java. They interpreted each item-test case-combination as a step and considered KCs as applied correctly if the test passed. They used different configurations of AFM to estimate students’ knowledge of the KCs. They used the KC actually used by the students in their solution as Q matrix and slight variations of it by distinguishing between additions and deletions of KCs in subsequent solutions. Furthermore they used a PC algorithm for systematic conditional independence to reduce the set of required KCs. They showed that models based on automatically extracted KC can be practically useful and that models using the reduced set of required KCs perform better than the ones using all of the actually used KCs as Q matrix. However, they only checked for the presence or absence of KCs in a student’s solution without considering how often or in which order the KCs were applied.

Huang et al. [95] also tried to reduce the set of required KCs. Again they used data from JavaGuide to fit different BKT and PFA models. To define a Q matrix, they used three reduction methods. First, they selected those KCs as required which occur most often in the model solution. The second approach is to use those subset of KCs which are most difficult as required KCs. The difficulty of KCs was determined by the KC difficulty coefficient of the PFA model. As third reduction method they used a labeling by experts. They showed that reducing the set of required KCs can lead to a better prediction performance. However, their reduction methods are based on a fixed existing code (as students have only to predict output/a variable value and not to write code by themselves) and, thus, are not applicable on open-ended programming exercises except for the expert-based labeling.

Rivers et al. [96] used AFM and learning curve analysis to identify KCs students are struggling with. Similar to Yudelson et al. [34] and our approach the KCs were elements which they derived automatically from ASTs. But instead of assuming that all KCs in a solution were applied incorrectly if the solution is incorrect, they used the ITAP algorithm [97] to find a corrected version of the solution. The KCs which are different in the original solution and the corrected one are then assumed as being applied incorrectly.

Only Yudelson et al. [34] and Rivers et al. [96] have considered different meta-configurations. While Rivers et al. [96] focused mainly on how to define a *step* and how to account for incorrectly applied KCs, Yudelson et al. [34] focused more on the definition of a Q matrix. Most of the approaches used either a fixed Q matrix which was defined by an expert (either human or based on a model solution) or used the KCs which were actually applied by the students in their solution. All of the approaches considered every submission/attempt as a step.

3.3. Patterns in Programming Behavior

There is a lot of research using snapshots of students' code to determine how students progress through an assignment. Metrics referring to changes in code, number of error messages, and students' compilation or testing behavior are used to determine and visualize a student's trajectory. However, these trajectories are often analyzed manually to identify general programming strategies of students [87, 98, 99, 100, 101]. Blikstein [98] identified seven canonical strategies in code creation. He could identify three different groups of students: *copy and pasters* who start with an existing program and modify it often by browsing for other useful sample programs and pasting code from that programs into their current program; *self-sufficients* who try to completely write the program by themselves; and *mixed-mode* students who are a mixture between the previous both types, i.e., write most pieces of their code by themselves but also look for solutions of difficult parts in sample programs. Hosseini et al. [99] used JavaParser [93] to extract concepts used in the solutions and compared the change of concept number to the change in the correctness level. They identified 4 types of students: *builders* gradually build their programs, with increasing number of concepts also the correctness increases; *massagers* usually have long streaks with only small changes in the number of concepts used without changing the level of correctness of their program; *reducers* start with large amount of code and then gradually

	skill model	KCs	observations	KC count	step definition	Q matrix
Corbett and Anderson [68]	BKT	production rules	rule applied correctly/incorrectly	binary	each coding step	expert
Mayo and Mitrovic [90]	BN	constraints	relevant/satisfied/violated constraints	binary	every	used
Kasurinen and Nikula [33]	BKT	programming structures, e.g., while, for, file	correct application of guidelines related to KCs	binary	every	used
Gonzales-Brenes and Huang [78]	PFA, LR-DBN, FAST	Java concepts obtained by JavaParser [93]	output/variable value answered correctly/incorrectly	binary	every	expert
Huang et al. [94]	CKM-HSC	Java concepts obtained by JavaParser and combinations of them (integration skills) [93]	output/variable value answered correctly/incorrectly	binary	average success rate over attempts	expert
Berges and Hubwieser [32]	IRT	concepts of object-oriented programming, e.g., array, assignment, overloading	observable items related to KCs	binary	only one assignment	all
Yudelson et al.[34]	AFM	Java concepts obtained by JavaParser [93]	passed/failed test cases	binary	every, diff	used, PC
Huang et al. [95]	KT, PFA	Java concepts obtained by JavaParser	output/variable value answered correctly/incorrectly	binary	every	used, most frequent, most difficult, expert
Rivers et al.[96]	IRT/AFM	language elements obtained by an AST parser	deviation of applied KCs from KCs in model solution determined by ITAP algorithm (incorrect KCs diff+all)	binary	first, every	expert

Table 3.3.: Skill models for programming and their meta-parameters

reducing the number of concepts by optimizing their code moving towards an increasing correctness; *strugglers* are students who can not make their program work for a long time, no matter how often they change their code. Meier et al. [101] analyzed students compile times and error frequencies in an introductory Python course. They identified 3 different groups of students: *bricklayers* who gradually build and run their programs and are comparable to builders by Hosseini et al. [99]; *stonecutters* who run their code only after writing a large portion of the code and then need several attempts to fix their errors bit by bit; and *masters* who also run their code only after their code is almost complete but only have few errors which they fix quickly.

However, manual analysis of students' trajectories is tedious. Thus, other studies use machine learning to analyze students' progress within an assignment. Piech et al. [48] trained Hidden Markov Models (HMMs) on data from exercises with Karel the Robot [102] using a programming language similar to Java. The states of the HMM correspond to states of the program and achieved subgoals, so called *milestones*. They use k-medioids clustering with a distance metric defined by a combination of the difference in API calls and AST difference to compute the milestones. Then, an EM algorithm was used to train the HMM. Finally, k-means clustering is used to determine different patterns in the paths students took through the HMM. They found out that students who make smaller steps always in the right direction of the solution, comparable to the builders by Hosseini et al. [99], usually achieve a higher midterm score than students who get stuck in sink states before directly creating a completely correct solution.

Wang et al. [100] use recursive neural networks (RNNs) which are fed with AST representations of the student's code to trace a student's knowledge development within an assignment and predict whether he/she will be able to solve the exercise correctly. Using k-means clustering they found 5 trajectory clusters in which they identified 3 different groups of students by qualitative analysis. The first group corresponds to builders by Hosseini et al. [99], who make consistent progress. The second group makes inconsistent progress and follows more a trial-and-error approach, comparable to tinkerers by Perkins et al. [103]. Students in the third group use more hard-coded simple elements instead of more complex elements like if-statement or loops.

Yee-King et al. [104] used k-means clustering to identify different programming behavior. They identified 4 groups of programming patterns: students in the first group tend to change their code a lot, most often by using new vocabulary, i.e., function calls the student has not used or seen previously; students in the second group do not change much in their code, but the changes were done quickly; students in the third group also had a lot of changes but they often reused code from demonstrator code and are, thus, comparable to the copy and pasters by Blikstein [98]; students in the last group also tend to reuse code but in contrast to the third group, the changes were only small.

Only few studies considered changes throughout a course instead of a per-assignment basis [33, 105, 106]. Blikstein et al. [105] analyzed code update patterns in a Java programming course and how this patterns changed throughout the course. Code update patterns are defined by the size of the code changes, i.e., adding, removing, or modifying a line, and the frequency of code changes. Based on the level of change in their update patterns they divide the students into five groups. They found out that students who change their update pattern to a great amount tend to achieve better results in the final exam. Ahadi et al. [106] analyzed how students' overall performance changed over time by looking at the average correctness of students' submissions. They find out that the performance which is explainable by the incremental nature of programming and that new knowledge has to be integrated continuously.

Most related to our work, are the work by Rivers et al. [96] and Smith and Rixner [107]. Rivers et al. [96] used the tokens from an AST as KCs to train IRT models which describe how each KC is learned over time. They found out that while some KCs show expected learning curves with decreasing error rate, others have been already known at the beginning or were not learned at all. Although, they already thought about some properties of the programming domain like KCs, the definition of a step, and step correctness, an elaborative study about the effect of these meta-parameters is missing. Furthermore, they only restrict to the syntactical level of knowledge.

Smith and Rixner [107] focused on the analysis of run-time errors over time which they call the *error landscape*. They do not only consider error frequencies, but also how many submissions were required to solve the error, and how often an error re-occurred within an assignment. Unfortunately, they only analyze the change of errors within an assignment or between two subsequent assignments but not throughout the complete course. Furthermore, they focused on run-time errors which result in exceptions, syntactical errors and failures however are not further sub-divided.

3.4. Summary and Research Delta

While most studies using code snapshots provide insights into how students write programs, they do not investigate how students learn to program. In this thesis, we want to fill the gap by analyzing how programming beginners' knowledge and "anti-knowledge", i.e., errors, evolves over time. We extend the properties of the programming domain identified by Rivers et al. [96] and evaluate how they can be incorporated into skill models and which effect they have on the performance of the skill models. Instead of using IRT, we want to use DBNs for creating learning curves. Furthermore, we do not only consider the lowest level of knowledge in programming, i.e., syntax, but also a more semantic level by taking roles of variables as SAPPs into account.

Similar to Smith and Rixner [107], we analyze the error landscape of programming beginners. But instead of looking at the errors on a per-assignment basis, we track the evolution of errors throughout the complete course. We prefer to use the categorization of programming errors by Bayman and Mayer [54] in our work which distinguished between three types of knowledge: *syntactic*, *conceptual*, and *strategic* (see Section 2.1). In most studies, only errors in the categories of syntactic and conceptual knowledge are analyzed. Studies regarding strategic errors are only limited to a small set of data or focus only on a small set of language concepts. In our work, we consider all types of errors that lead to a solution being considered as incorrect by manually analyzing student's submissions in a C programming beginners course. Since we hypothesize that students may have different error patterns, we use the results from the error landscape to cluster students by their error patterns. The identification of common patterns in knowledge as well as error evolution can lead to new insights of certain issues during learning and lay a foundation for personalization in ITS.

4. Estimation of Programming Knowledge

In this section, we present our methodology for the estimation of a student's programming knowledge. First, we describe how we extract the KCs from student's code in Section 4.1. For the similarity analysis of the code, we define three different similarity metrics in Section 4.2. In Section 4.3, we adjust PFAs and AFMs to cope with a dynamic Q matrix and propose a DBN topology for the programming domain. Afterwards, we describe how the DBN can be used to derive learning curves in Section 4.4.

4.1. Identification of Knowledge Components in Students' Code

As stated in Section 2.1, programming consists of several skills, including writing, reading, or debugging code. Therefore, KCs for programming can be manifold, e.g., rules for compliance to particular coding conventions, the ability to understand certain compiler errors, or the ability to assemble pieces of code such that a certain goal is achieved. In this thesis, we focus on the two lower levels of programming knowledge as defined by Bertels et al. [3]: programming primitives and semantically augmented programming primitives (see Section 2.1).

4.1.1. Programming Primitives

We use the grammar of the C programming language as defined by the ISO/IEC 9899:2011 (C11) standard [108] to extract PPs. We summarize similar PPs to derive KCs and their hierarchical structure. The KCs on the lowest level, 0, refer to the basic building blocks of C, i.e., preprocessor directives, declarations, expressions, and statements. Level 1 KCs describe the logical manifestation of these basic blocks for the preprocessor directives (include, define), statements (iteration, selection, jump, label, and block), and expressions (e.g., comparison, member access, increment/decrement, assignment, logical), or in case of the declaration the single components of which it is made (type specifier, type qualifier, storage class, and declarator). Level 2 contains the concrete instantiation of the abstract elements on level 1, e.g., we can distinguish between `while`, `for`, and `do-while` loops. Level 3 finally contains all remaining types of AST nodes. The higher the level, the more detailed the sub-division of the elements. Table 4.1 shows how many KCs are assigned to each level. Table 4.2 shows an excerpt of our KC classification related to the language element *statement*. *Statement* is the most general element and, thus, a KC on level 0. On

level	0	1	2	3
number of KCs	4	30	103	147

Table 4.1.: Number of identified KCs on each level

0	1	2	3

statement	block
	iteration
	jump
	label
	selection
		break
		continue
		do while
		for
		goto
		if
		return
		switch
		while
			case
			default
			else
			then

Table 4.2.: Classification of KCs for statements on different levels

level 1 it is sub-divided into the different types of statements in the C programming language, i.e., *block*, *iteration*, *jump*, *label*, and *selection*. These statement types can be further sub-divided, e.g., *selection* can be sub-divided into *switch* and *if*. On level 3, we regard the single parts of these elements, e.g., a switch statement can contain *case* and *default* labels. The complete classification of KCs can be found in Appendix A.

The KCs used by a student in his/her solution can be derived from an AST (see Section 2.3). We use the CDT framework [109] of the Eclipse Foundation as a basis to create our own AST. In each node of the AST, we store which KCs are associated with the node on each level, such that we can easily access the applied KCs when analyzing the program.

4.1.2. Semantically Augmented Programming Primitives

In Section 2.2, we showed that variables can have different semantic roles within a program. In this thesis, we are using these roles as KCs on the SAPP level. The process of variable analysis consists of several steps which are depicted in Figure 4.1 and will be described in the following.

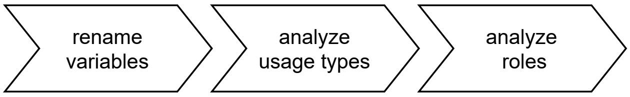


Figure 4.1.: Process of variable role analysis

Rename variables

In many programming languages identifier names can be defined at different levels, each having a scope, i.e., region in the program where the name can be used. Thus, a variable with the same name can be declared multiple times. Each use of an variable is associated with a *binding* which describes to which declaration the current variable belongs. Two variables with the same name but different bindings can be treated as completely different variables. To make this independence explicit, we rename all variables by suffixing an integer number.

Analyze usage types

Variables can be used in different contexts. The *variable usage* describes how the variable is used in a certain occurrence and is defined by four elements:

- *Scope:* The scope is a list of elements of the types *loop*, *if*, *switch*, and *function* and describes the nesting of the variable.
- *Access type:* The access type describes whether the variable is written or read in that usage.
- *Usage type:* The usage type describes the variable's immediate task within an expression. Since an expression can be composed of multiple expressions, a variable can have more than one usage type. So also the variable usage is defined by a list describing the nesting of expressions. For the analysis of variable roles, we have identified the following usage types: declaration, initialization, assignment, function call, branch condition, loop condition, return value, increment/decrement, input, output, array index, comparison, and parameter.
- *Superior statement:* The superior statement is the next higher statement in which the variable is nested.

Scope, usage type and superior statement are determined by going up the AST. For the scope it is looked for loops, if-statements and function declarations. The superior statement is the first statement reached when going up the AST from the variable node. The usage types follow from the expression nesting until the superior statement is reached. The access type is determined by looking at the usage of the variable. A variable has a write access if is used at the left-hand side of an assignment, in an increment/decrement, or has the usage type input.

Example 4.1 *Consider the example program in Listing 4.1. The variable `i` has six occurrences or* variable usages *within the code. The definition of the variable usage of `i` in line 5 would look as following:*

- Scope*: loop - loop - function*

```
int main() {
  int i = 0;
  while(scanf("%d",&i)) {
    int sum = 0;
    while (i>0){
      sum += i;
      i—;
    }
    printf("The sum of the numbers from 1 to %d is %d", i, sum);
  }
}
```

Listing 4.1: Example program

- Access type: *read*
- Usage type: *comparison - while condition*
- Superior statement:

```
while (i>0){
  sum += i;
  i--;
}
```

The scope results from the nesting in two `while` loops which are located in the `main` function. The expression `i>0` is a comparison and is used as a condition of the `while` loop. Since the `while` loop is the first statement the variable occurrence is nested in, the complete `while` statement is the superior statement. None of the above rules for write access applies to that occurrence of `i`, as such the access type is read.

Analyze roles

As already stated in Section 2.2 variables may have different roles during their lifetime. To cope with sporadic changes, we split the lifetime of a variable into several *lifelines*. A new lifeline of a variable starts if the variable is re-initialized. We consider a variable as re-initialized if it is assigned a fixed value. A fixed value can be a constant, a fixed variable, or an expression consisting only of fixed values. A lifeline of a variable consists of all usages of that variable until the re-initialization. The roles *fixed*, *temporary*, and *organizer* are analyzed on the complete lifetime of a variable, while the remaining roles are analyzed on a per-lifeline basis.

Rist [110] investigated how students created plan schemas. He determined that each plan has a *focal line* which immediately implements the goal. For instance, if the plan is to sum elements, the focal line is the sum statement. It encodes the core task of the plan. Byckling et al. argue that variable roles also have a focal line [111]. To identify the variable roles in the code, we define constraints on the focal line, the scope of the focal line and the allowed

Focal line	(1) *var* = *var op fixedvalue* (2) *var* = *fixedvalue* op_{com} *var* (3) *var op*= *fixedvalue* (4) *var*++ (5) *var*−− (6) ++*var* (7) −−*var*
Focal scope	loop - *
Usage	• initialization outside of scope • inside scope read-only

Table 4.3.: Constraints for the variable role *stepper*

value changes during the lifetime resp. lifeline of the variable. The rule descriptions for all variable roles can be found in Appendix C.

Example 4.2 *Table 4.3 shows the constraints for the variable role* stepper. *A focal line can be expressed by an assignment (1)-(2), an equals assignment (3), or by increment/decrement (4)-(7) where var is a placeholder for the treated variable, fixedvalue corresponds to a fixed expression,* `op` *can be a binary operator, and* op_{com} *is a commutative binary operator (*`+` *or* `*`*). The focal line has to be located directly inside a loop. The* $*$ *denotes that the outer nesting of the loop does not matter. For the usages of the variable not in the focal line applies that the initialization has to be located outside the loop and that the stepper variable is only changed in the focal line inside the loop, i.e., further usages of the variable inside the loop are restricted to read-only access.*
Applying these rules to the program in Listing 4.1, we can identify the focal line (5) for a stepper in line 7. The scope of that usage is loop - loop - function *and as such fits to the scope of a stepper. The initialization (line 1) is located outside of the focal scope, the usages inside the scope (lines 5,6, excl. focal line) are read-only. As a result, we can conclude that the variable* `i` *is a stepper.*

A special construct of C programming are *pointers*. The value of a pointer variable is the address of a memory location in contrast to a normal variable which contains the value that is stored at that address. Consider the example in Listing 4.2. The value of variable `a` is changed in line 4 by changing the value of the address the pointer `p` is pointing to. Without considering the indirect manipulation via pointers, we would assume that `a` is a *fixed* variable. However a complete static tracking of pointer behavior is not feasible if pointer arithmetic comes into play. Therefore, we only account for simple dependencies in the form of *pointer* `=` `&`*variable* when checking whether the value of a variable changes. When analyzing the role of a pointer variable, we analyze the role behavior of the pointer, i.e., the change of the addresses, not thus of the values it points to. For example, the pointer `p` in Listing 4.2 is *fixed* although the value at that address is changed.

```
int a = 5;
int b = 2;
int* p = &a;
*p = 3;
printf(a*b); // output: 6
```

Listing 4.2: Pointer example

4.1.3. Knowledge Application Model

There exists a variety of representations of source code, e.g., ASTs which describe the syntactic structure of a program, control and data flow graphs illustrating the control resp. data flow of programs, or program dependency graphs [112], that comprise both data as well as control dependencies. We introduce a model representing a program in terms of KCs that are used in a certain solution for a programming exercise, the *Knowledge Application Model (KAM)*.

Definition 4.1 (Knowledge Application Model) *A Knowledge Application Model (KAM)* $\mathcal{K} = (V, E)$ *is a hierarchical directed path graph with*

- *the set of nodes* $V = V_s \cup V_c$*,*
- *the set of simple nodes* V_s *with* $v_s \subseteq KCs$*,*
- *the set of complex nodes* V_c *with* $v_c = (\tau, \mathcal{K}_{emb})$ *where* τ *is the type and* $\mathcal{K}_{emb}$ *is the embedded KAM of the complex node, and*
- *a function* $\tau : V_c \rightarrow \{FUNCTION, IF, SWITCH, WHILE, DO-WHILE, FOR\}$ *which assigns a* type *to each complex node,*
- *a function* $emb : V_c \rightarrow \mathcal{K}^P$ *which denotes the* embedded KAM *for each complex node, and*
- *the set of edges* $E \subseteq V \times V$*.*

In KAMs, we distinguish between two types of nodes, *simple* nodes and *complex* nodes. Simple nodes represent simple sequences of statements while complex nodes represent nesting in the program structure. Nesting is caused by functions, selection statements, i.e., if and switch statements, and loops. Thus, each complex node v_c is annotated with its *type* τ. For level 0 and 1 the potential types are reduced to loop, selection, and function. Additionally, complex nodes contain an *embedded KAM* $emb(v_c)$ which represents the nested code within the function/statement.

To create the KAM for a solution, the code is sliced into basic blocks. Basic blocks are sequences of statements without branching. For each basic block, a simple node v_s is created which contains a set of all KCs used in this block $v_s \subseteq KCs$. Thus, the order of instructions within a basic block does not matter for the KAM. The number of KCs within a simple node $v_s \in V_s$ is denoted by $|v_s|$. When nesting occurs, we create a complex node for the nesting element with the corresponding type and construct the embedded KAM for the nested code.

Figure 4.2 shows how the embedded KAMs are created for particular branching elements. The embedded KAM of a function starts with a simple node containing the KCs from the function header. This node is followed by one or more nodes representing the body of the function. These nodes can be simple as well as complex nodes. An embedded KAM of complex nodes with type WHILE or DO-WHILE starts with a simple node representing the looping condition followed by one or more nodes representing the body of the loop. Similarly, the embedded KAM of a `for` loop is constructed. In this case, the first node does not only contain the KCs of the looping condition, but also the KCs of the initialization and the iteration expression contained in the `for`-header. The embedded KAM of an if statement starts with a simple node containing the KCs of the condition, followed by one or more nodes representing the then branch and - if existent - by one or more nodes representing the else branch. A switch statement is expressed by an embedded KAM starting with a simple node representing the switch expression. This is followed by nodes for each case.

Definition 4.2 (Path length in KAMs) *The* path length *of a KAM $\mathscr{K} = (V,E)$ is defined as:*

$$pathlength(\mathscr{K}) = |V| + \sum_{v_c \in V_c} pathlength(emb(v_c))$$

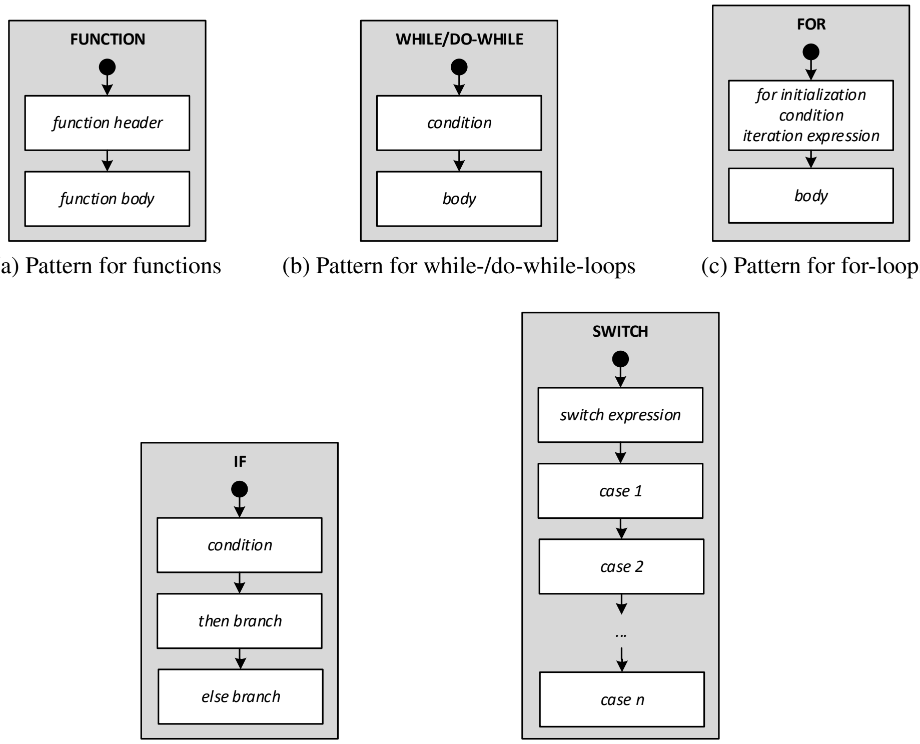
The path length corresponds to the number of nodes in an KAM by also adding the numbers of nodes in the embedded KAMs.

Example 4.3 *Figure 4.3 shows an example of a KAM. The example program consists of the preprocessor directive `include` and a `main`-function. The preprocessor directive represents the first basic block and is thus transferred into a simple node. The `main` function is transferred into a complex node of the type FUNCTION. The function can be sliced into three basic blocks: the variable declaration at the beginning of the function, the if statement, and the return statement at the end of the function. While the declaration and the return statement can be transferred into simple nodes, the if statement has to be represented by a complex node of type IF. The embedded KAM of the IF-node is split into a simple node for the condition of the if statement, a simple node for the then branch, and a simple node for the else branch. The path length of the KAM is 9.*

A KAM can show the separation of KCs in a program and contains limited control flow information, but it does not consider the order of statements within basic blocks, variable naming, or concrete values of primary expressions.

4.2. Code Similarity

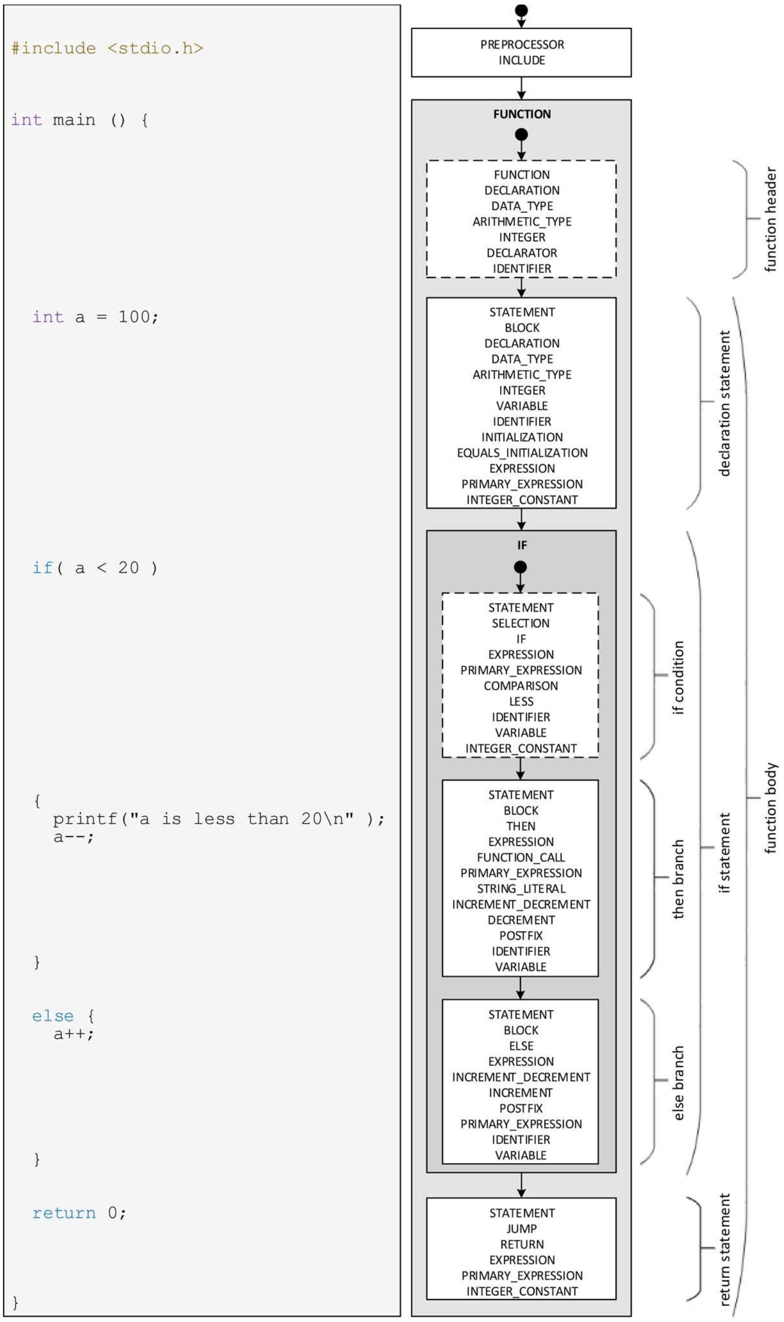
In software engineering, the assessment of code similarity has various applications like clone detection [113], code plagiarism detection [114], finding similar bug fixes [115], and



(a) Pattern for functions (b) Pattern for while-/do-while-loops (c) Pattern for for-loops

(d) Pattern for if-statements (e) Pattern for switch-statements

Figure 4.2.: Construction patterns for complex nodes



```
#include <stdio.h>

int main () {

  int a = 100;

  if( a < 20 )

  {
    printf("a is less than 20\n" );
    a--;

  }

  else {
    a++;

  }

  return 0;

}
```

Figure 4.3.: Example of a KAM (KC level 2)

program comprehension [116]. Similarity algorithms can be divided into several classes. Metric-based similarity detection [117, 118, 119] uses software metrics like, e.g., Halstead complexity metrics, to determine the similarity. Text-based [120, 121] approaches use string comparison of the source code as a measure for similarity. For token-based approaches [122, 123, 124], the code is transformed into tokens and then the token streams are compared. To abstract from formatting and other lexical issues, one can use similarity metrics based on the comparison of trees [125, 126], e.g., ASTs. To capture similarity on a more semantic level, graph-based algorithms can be used by comparing, e.g., control-flow graphs [127] or program dependency graphs [128, 129, 130].

In this section, we present three different similarity metrics that we use to assess the similarity between two solutions of students. The *set similarity* is a very simple metric based on the comparison of the sets of used KCs in the solutions (Section 4.2.1). The set similarity neither accounts for structural differences, nor does it consider when and how often a KC is applied in a solution. The second similarity metric we define here is AST similarity. The AST similarity is based on the costs of transforming the AST of one solution into the AST of the other solution (Section 4.2.2). The last similarity metric we present here is the KAM similarity. It is based on the comparison of two KAMs (Section 4.2.3). In addition, we introduce a metric which we call *diversity* in Section 4.2.4. It describes the proportion of different solutions. While the diversity tells us whether the solutions are different, the similarity describes to what extent they differ.

4.2.1. Set Similarity

For the set similarity of two solutions we are just regarding the overall sets of KCs of the solutions independently of the order in which they were applied. A typical measure for the similarity of two sets is the Jaccard index [131].

Definition 4.3 (Set Similarity) *Let $KCs(i)$ and $KC(j)$ be the sets of KCs used in solutions i and j. The* set similarity *is defined by:*

$$sim_{set}(i,j) = \frac{|KCs(i)| \cap |KCs(j)|}{|KCs(i)| \cup |KCs(j)|}$$

Example 4.4 *Consider the code snippets from Figure 2.5. Table 4.4 shows the KCs on each level for the both solutions as well as the resulting set similarity. Each higher level also contains the KCs from the lower level to mitigate the differences on higher levels. We can see that set similarity decreases with increasing KC level.*

4.2.2. AST Similarity

One possible approach to determine the similarity between two ASTs is to use the *tree edit distance*, i.e., look at which actions are necessary to transform one AST into another. In

level	KCs solution i	KCs solution j	$\lvert\mathbf{KCs(i)} \cap \mathbf{KCs(j)}\rvert$	$\lvert\mathbf{KCs(i)} \cup \mathbf{KCs(j)}\rvert$	$\mathbf{sim_{set}}$
0	statement expression	statement expression	2	2	1
1	iteration comparison primary block assignment arithmetic	iteration comparison primary block assignment inc/decrement	7	9	0.78
2	while identifier less integer constant basic assignment binary arithmetic	while identifier less integer constant add assignment postfix increment	11	18	0.61
3	addition	post increment	11	19	0.58

Table 4.4.: Set similarity analysis on different KC levels

doing so, each action is associated with certain *costs*. We use the GumTree algorithm [60] (see Section 2.3.2) to determine an edit script between an ASTs $\mathscr{A}_1 = (V_1, E_1)$ and an AST $\mathscr{A}_2 = (V_2, E_2)$. We define the costs for each action with node $u \in V_1$ and $v \in V_2$ as following:

$$costs_{add(v)} = costs_{delete(u)} = costs_{move(v,v_p,i)} = 1 \tag{4.2.1}$$

Deleting or adding a node has a cost of 1 since this action only affects a single node. A move action may affect multiple nodes, since we can move a complete subtree. However, since the code fragment itself – and hence the nodes in the substree – is correct but simply misplaced, we set the costs for a move action also to 1.

$$costs_{update(v,\varphi_{new})} = \begin{cases} 0, & \text{if } l_v = variable \\ \dfrac{sim(\varphi_{old}, \varphi_{new})}{2}, & \text{if } l_v = constant \\ 0.5, & else \end{cases} \tag{4.2.2}$$

Updating a node means that its value has to be changed. The value can represent an operator, a name, a simple type like `int`, or a constant value. Since the label is correct, the node also is partially correct. To account for partial correctness, we define that the maximum costs are 0.5 instead of 1 for an update action. However, it is quite common that variables are named differently in different implementations. To avoid overestimation of costs because of variable naming, we set the costs for an update of the variable naming to 0. We do not apply that rule to function names because they are prescribed for the exercises in our data

operation	costs
`update((assignment,=),+=)`	0.5
`add((inc/decrement,++(post)),(block,eps),1)`	1
`move((identifier,i),(assignment,+=),1)`	1
`move((identifier,i),(inc/decrement,++(post)),0)`	1
`delete((identifier, sum))`	1
`delete((binary arithmetic,+))`	1
`delete((identifier,i))`	1
`delete((constant,1))`	1
`delete((binary arithmetic,+))`	1
`delete((assignment,=))`	1

Table 4.5.: Edit script and costs per operation

in most cases. For constant values used in expressions like, e.g., String literals, we use the similarity between the old and the new value based on the Levenshtein distance [132] to calculate the costs in a range between 0 (at full similarity) and 0.5 (completely different). The rationale is that especially String literals which are used for output usually differ only slightly because of typos or incorrect formatting. For the names of functions, operators, types, includes, etc. a difference in values corresponds to a great logical difference and is, thus, associated with maximum costs.

Definition 4.4 (AST Similarity) *Let $\Delta = \langle o_1, ..., o_n \rangle$ be the edit script to transform AST $\mathscr{A}_1 = (V_1, E_1)$ into AST $\mathscr{A}_2 = (V_2, E_2)$. The AST similarity is defined by:*

$$sim_{AST}(\mathscr{A}_1, \mathscr{A}_2) = 1 - \frac{\sum_{o_i \in \Delta} costs(o_i)}{|V_1| + |V_2|}$$

The numerator in the equation describes the tree edit distance, i.e., the total costs for the execution of an edit script. The denominator normalizes the edit distance according to the number of nodes in both ASTs. In case that the ASTs have no matching nodes, i.e. maximum dissimilarity, the edit distance would need $|V_1|$ deletions and $|V_2|$ additions, i.e., have an edit distance of $|V_1| + |V_2|$, to transform $\mathscr{A}_1$ into $\mathscr{A}_2$.

Example 4.5 *Consider the AST comparison from Example 2.4 on page 18. Table 4.5 shows the costs for each operation in the edit script. The tree edit distance is 9.5. The similarity is:*

$$sim_{AST} = 1 - \frac{9.5}{15 + 10} = 0.62$$

4.2.3. KAM Similarity

For the KAM similarity, we also consider the order of the nodes in a KAM. Since a KAM is a path graph, we can regard a KAM as a sequence of nodes. We can use sequence alignment to determine the difference between two solutions. One common metric for sequence comparison is the Levenshtein distance [132] of two sequences. The Levenshtein distance describes how many changes are required to transform one sequence into an other. There are three types of changes possible: *add*, *delete*, and *replace* and each change is associated with particular costs.

For two KAMs $\mathscr{K}_1 = \langle u_1, u_2, ..., u_m \rangle$ and $\mathscr{K}_2 = \langle v_1, v_2, ..., v_n \rangle$ with paths $\langle u_1, u_2, ..., u_m \rangle$ (resp. $\langle v_1, v_2, ..., v_n \rangle$) we can use the Wagner-Fisher algorithm [133] to calculate the edit distance *ED* as follows:

$$\begin{aligned}
&D_{0,0} = 0\\
&D_{i,0} = i, 1 \leq i \leq m\\
&D_{0,j} = j, 1 \leq j \leq n\\
&D_{i,j} = min \begin{cases} D_{i-1,j-1}, \text{if } u_i = v_j \\ D_{i,j-1} + costs_{insert(v_j)} \\ D_{i-1,j} + costs_{delete(u_i)} \\ D_{i-1,j-1} + costs_{replace(u_i,v_j)} \end{cases}
\end{aligned} \tag{4.2.3}$$

If the nodes are identical, there are no costs. For adding, deleting, and replacing nodes, we have to distinguish between simple and complex nodes:

$$costs_{add(v)} = \begin{cases} 1, \text{if } v \in V_{\mathscr{K}_2,s} \\ 1 + pathlength(emb(v)), \text{if } v \in V_{\mathscr{K}_2,c} \end{cases} \tag{4.2.4}$$

The costs for adding a simple node are 1, while costs for adding a complex node are 1 plus the number of nodes in the embedded KAM of the complex node. The same also applies for deleting a node:

$$costs_{delete(u)} = \begin{cases} 1, \text{if } u \in V_{\mathscr{K}_1,s} \\ 1 + pathlength(emb(u)), \text{if } u \in V_{\mathscr{K}_1,c} \end{cases} \tag{4.2.5}$$

The costs for replacing a node u by a node v are:

$$costs_{replace(u,v)} = \begin{cases} 1 - sim_{set}(u,v), & \text{if } u \in V_{\mathscr{K}_1,s}, v \in V_{\mathscr{K}_2,s} \\ 1 + ED(\langle u \rangle, emb(v)), & \text{if } u \in V_{\mathscr{K}_1,s}, v \in V_{\mathscr{K}_2,c} \\ 1 + ED(emb(u), \langle v \rangle), & \text{if } u \in V_{\mathscr{K}_1,c}, v \in V_{\mathscr{K}_2,s} \\ ED(emb(u), emb(v)), & \text{if } u \in V_{\mathscr{K}_1,c}, v \in V_{\mathscr{K}_2,c}, \tau(u) = \tau(v) \\ 1 + ED(emb(u), emb(v)), & \text{if } u \in V_{\mathscr{K}_1,c}, v \in V_{\mathscr{K}_2,c}, \tau(u) \neq \tau(v) \end{cases} \tag{4.2.6}$$

Replacing a simple node by another simple node means replacing the KCs which differ. Thus, the costs are the proportion of KC which are not similar. Costs for replacing a simple node by a complex node are the costs for transforming a KAM which only consists of the simple node into the embedded KAM of the complex node plus an additional cost of 1 for adding the branching. The costs for replacing a complex node by a simple node are calculated similarly. The costs for replacing a complex by another complex node are the costs for transforming the embedded KAM of the first one into the embedded KAM of the second one. If the types of the complex nodes differ, an additional cost of 1 is added for changing the complex type.

We can obtain the edit distance from Equation 4.2.3 by calculating $D_{m,n}$:

$$ED(\mathscr{K}_1 = \langle u_1, ..., u_m \rangle, \mathscr{K}_2 = \langle v_1, ..., v_n \rangle) = D_{m,n} \tag{4.2.7}$$

An edit distance on its own has only little expressive power. For example, intuitively, one would assume that two KAMs, each consisting of a single simple node, with an edit distance of 1 are less similar than two KAMs consisting of 100 nodes and an edit distance of 1. Longer paths offer more opportunities for differences. Thus, we define the KAM similarity as the edit distance in relation two the length of the longer path.

Definition 4.5 (KAM Similarity) *The* KAM similarity *between a KAM $\mathscr{K}_1$ and a KAM $\mathscr{K}_2$ is defined as:*

$$sim_{KAM} = 1 - \frac{ED(\mathscr{K}_1, \mathscr{K}_2)}{max(pathlength(\mathscr{K}_1), pathlength(\mathscr{K}_2))}$$

If the edit distance is greater than the number of nodes in the bigger KAM, the similarity is 0. For the example above, that means that the KAMs with one single node would have a similarity of 0 while the KAM with 100 nodes would have a similarity of 0.99.

Example 4.6 *Consider the KAMs in Figure 4.4 which belong to the code snippets from Figure 2.5 on page 19. The colors denote the KC level: orange=0, red=1, green=2, and blue=3. Since the type of the complex nodes is the same, the edit distance between the KAMs corresponds to the edit distance of the embedded KAMs. Also the first simple node in both embedded KAMs is equal, thus has an edit distance of 0. The costs for replacing the second simple node u_2 of KAM 1 by the second simple node v_2 of KAM 2 is calculated looking at the set similarity between the KCs in that nodes. The overall similarity of the KAMs is determined by:*

$$1 - \frac{sim_{set}(KCs(u_2), KCs(v_2))}{3}$$

The resulting similarities are 1 on level 0, 0.89 on level 1, 0.81 on level 2, and 0.79 on level 3.

The KAM similarity rewards structural similarity while differences within basic blocks are mitigated.

4.2.4. Diversity

To analyze how many different solutions students produce and, thus, determine the degree of diversity, we look at the ratio between the number of different KAMs for an exercise and the number of total solutions for that exercise:

$$diversity = \frac{\text{number of different solutions w.r.t. KAM} - 1}{\text{number of total solutions} - 1} \tag{4.2.8}$$

If all solutions are different, the number of paths is the same as the number of solutions and we get a diversity of 1. If all solutions are equal, i.e., in the sense of equal KAMs, there exists only one solution, and we get a diversity of 0.

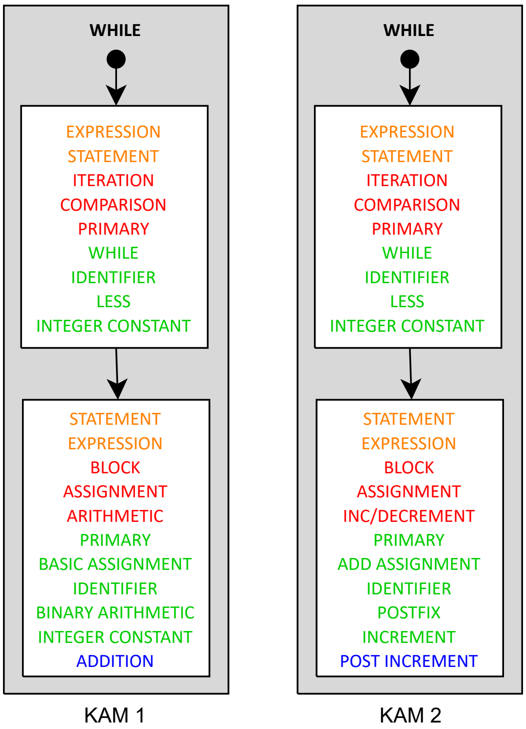


Figure 4.4.: Two KAMs

4.3. Adjustment of Skill Models for the Programming Domain

In this section, we will discuss certain properties of the programming domain which make skill modeling challenging (Section 4.3.1). Afterwards, we show how we adapt PFA (Section 4.3.2) and define a DBN topology (Section 4.3.3) to cope with these properties.

4.3.1. Properties of the Programming Domain

To effectively learn how to program, students need to freely write small programs with as little limitations as possible. This leads to a theoretically unlimited number of possible correct solutions. Having that many different possible solutions, also means that an exercise can be solved by many different combinations of KCs. Thus, the required KCs for solving an exercise are not fixed but dynamic depending on the actual solution path, e.g., one can solve an exercise iteratively or recursively. Furthermore a KC can be used multiple times in a solution, e.g., when multiple loops are required to solve the exercise, and a programming exercise usually requires more than one KC to solve it. Additionally, the KCs are often not independent of each other, e.g., knowing how a while-loop works increases the probability of knowing how a for-loop works because it means that the student has at least understood the basic concept of a loop. To sum up, programming exercises have the following properties:

- multiple KCs required;
- KCs are not independent of each other;
- a KC can be used multiple times in one solution; and
- there are multiple ways of solving the exercise, i.e., the Q matrix is dynamic.

When selecting an appropriate skill model for programming, the characteristics of programming exercises have to be considered. But not only the selection of an appropriate skill model is crucial, but also the selection of meta-parameters. The meta-parameters determine how data has to be collected and how features are selected and processed for training and usage of the skill model. We have identified six meta-parameters for skill models in the programming domain, but they are often also applicable to other domains:

KC level

Domain knowledge, and thus KCs, can be defined on different granularity levels. Higher KC levels allow us to do a more fine-grained estimation of knowledge. However, a higher level also means more parameters that have to be learned in the model, and thus, a higher complexity and execution time. Additionally, more fine-grained KCs may lead to a very sparse distribution of observations making it hard to make predictions.

Step definition

A step can be defined on different levels in the programming domain. It can be based on an exercise level, a submission level, every time a student saves his/her solution, or even on a keystroke level. Which step definitions are applicable often depends on the data available. We collect data on a submission level. On this level, we can distinguish between four different definitions of step:

- *first:* Only the first submission for an item is considered. The rational behind this is that the first attempt represents a student's starting point of knowledge and may be an indicator on which KCs he/she has to work on.
- *last:* Only the last submission for an item is considered. The rational behind this is that the last submission of a student to an exercise represents his/her knowledge state at the end of an exercise. KCs which are still incorrectly applied in the last submission may be a hint on which KCs the student is really struggling with.
- *every:* Considering every submission as a step allows to track a student's knowledge evolution within exercises.
- *diff:* The difference between two subsequent steps is considered as a step. When looking at two subsequent submissions, often only small changes are made in code. To not overestimate a student's knowledge of KCs which are kept unchanged, one could only look at the modified parts of code.

As stated by Huang et al. [94] the first submission of a student is often incorrect while the last one is often correct. Thus, considering only the first or the last attempt could lead to an under- resp. overestimation of a student's knowledge.

Minimum steps

At the beginning of an observation sequence there is often some noise. Usually, you cannot tell after one or two exercises if the student just had luck or actually understood a certain concept. The meta-parameter *minimum steps* defines after how many steps data is assumed to be relatively noise-free and, thus, can be considered for fitting a skill model.

Incorrect KCs

For most skill models it is important to know which KCs were applied correctly and which KCs were applied incorrectly in a solution. While this can be implemented quite easily for multiple choice questions by mapping the options to KCs, it is a more complicated task for open-ended exercises. Since there is not only "the one" correct answer, one has to identify which KCs were applied correctly, which were applied incorrectly, and which KCs are missing although they are required for the solution. Since automatic error correction is still an ongoing research field, we only have a look at three very basic approaches in this thesis:

- *all:* If a solution is incorrect, all KCs that were used in the solution are assumed as being applied incorrectly. This is a pessimistic estimation of KC knowledge since KCs are only considered as applied correctly when the complete solution is correct.
- *qmatrix:* If a solution is incorrect, all KCs which are required but missing in the student's solution or not required but used by the student are assumed as being applied incorrectly. This is an optimistic estimation of KC knowledge since it assumes that a KC is applied correctly if it was required in the solution w.r.t. the Q matrix.
- *diff:* We compare the solution to the most similar solution. The KCs which are different in both solutions are assumed as being applied incorrectly. We do not make any assumptions about the KCs which are same in both solutions as they still could be applied incorrectly.

KC count

In programming exercises a KC can be used more than once in a solution. So we have two possibilities how to count the KCs:

- *binary:* We ignore the concrete number of occurrences and just use a binary value which tells whether the KC was used in the solution or not. The rational behind this is that the student's knowledge of a particular KC is the same for each application of the KC during one solution as it corresponds to the same point in time.
- *multiple:* We use the concrete number of occurrences. The rational behind this is that KCs can be applied in different contexts and, thus, each application is an own practice opportunity.

Q matrix

In open-ended programming exercises, there exist several ways of solving the exercise. Additionally, it is not known in advance which KC the student actually will use in his/her solution. We can use the set of correct solutions to derive different definitions of a Q matrix for an exercise:

- *all:* All KCs from the domain model are required.
- *shared:* The set of KCs that occur in every solution path, i.e., the intersection of the KCs used in every correct solution.
- *union:* The set of KCs that occur in any solution path, i.e., the union of the KCs used in every correct solution.
- *common:* The set of KCs that is used in the most common solution path, i.e., the set of KCs of the KAM which represents the majority of the solutions.
- *used:* The set of KCs the student actually used in his/her solution.

- *set:* The set of KCs that is used on the solution path which is most similar w.r.t. set similarity (see Section 4.2.1) to the actual solution.
- *kam:* The set of KC that is used on the solution path which is most similar w.r.t. KAM similarity (see Section 4.2.3) to the actual solution.

While the actually used KCs in a solution are only applicable for training a skill model but not in practice when predicting a student's performance because we do not know which KC a student will use in his/her solution, the Q matrix definitions *all*, *shared*, *union*, and *common* can always be used because they are independent of a student's solution. The Q matrix definitions based on set and KAM similarity can be used in practice after the first submission as one gets an insight on a student's actual solution path.

4.3.2. Adjustment of PFA/AFM

Logistic regression models like AFM and PFA are able to deal with multiple KCs and with the multiple application of one KC in a solution. We extend traditional PFA/AFM models by a parameter r_{ijk} which describes if student i requires KC k to solve the exercise j. This allows us to define a dynamic, student-dependent Q matrix.

The AFM model is defined as follows:

$$ln\left(\frac{p}{1-p}\right) = \alpha_i + \delta_j + \sum_{k \in KCs} r_{ijk}\beta_k + r_{ijk}\gamma_k n_{ik} \tag{4.3.1}$$

The PFA model is respectively defined as follows:

$$ln\left(\frac{p}{1-p}\right) = \sum_{k \in KCs} r_{ijk}\beta_k + r_{ijk}\gamma_k s_{ik} + r_{ijk}\rho_k f_{ik} \tag{4.3.2}$$

However, the property of dependent KCs is not reflected by skill models based on logistic regression, since logistic regression assumes that features, i.e., KCs, are independent of each other.

4.3.3. A DBN Topology for Skill Models

Since PFA is not able to deal with dependencies between KCs, we use DBNs (see Section 2.4.1) to model those dependencies. A DBN is defined by its structure, also called topology, and a set of parameters. In this thesis, we learn the parameters from data while we define the topology manually. The topology we propose for skill modeling consists of three parts: the domain model, which defines the relationship between KCs, a KC-item mapping, which represents the Q matrix, and the result composition, which is used to distinguish between KC knowledge and the problem solving ability.

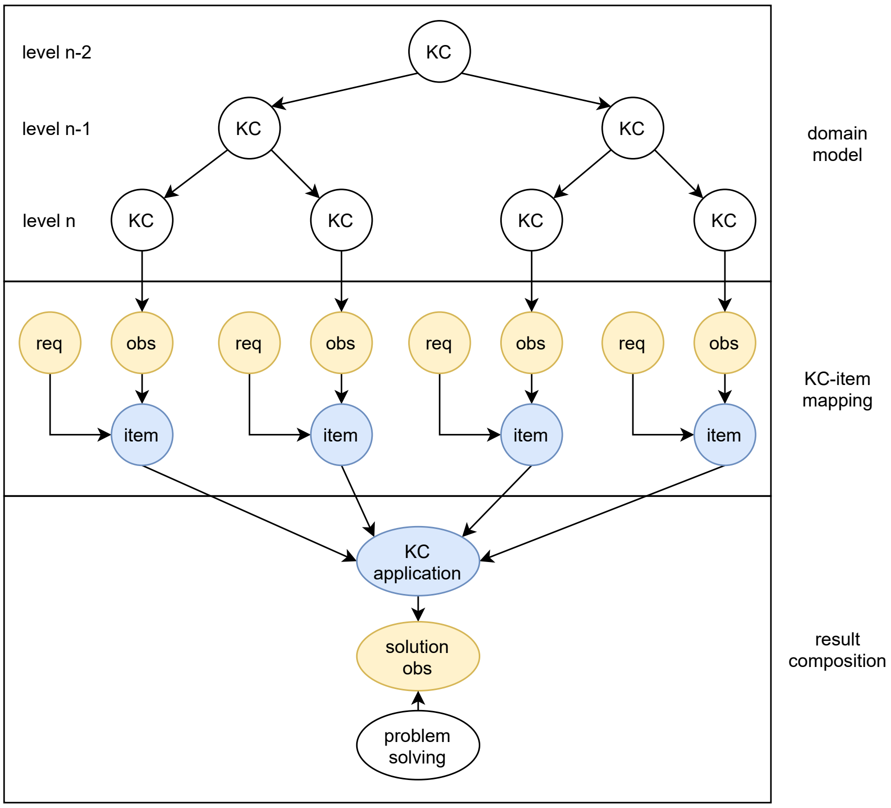


Figure 4.5.: DBN topology for skill modeling. White nodes represent hidden variables, yellow nodes represent evidences and blue nodes represent noisy AND gates with fixed CPTs.

Domain model

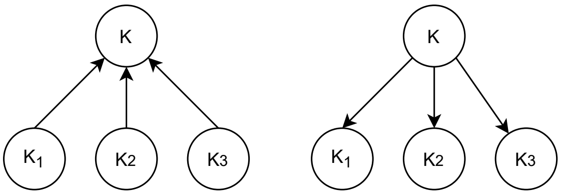
In the domain model, each KC is represented by a binary node which describes whether the KC is known or unknown. We distinguish between two types of dependencies between KCs: prerequisite and granularity relationships.

A *prerequisite relationship* defines that KC K_1 has to be learned before a KC K_2 can be learned. In the DBN, this kind of relationships is represented by an edge from K_1 to K_2 [134]. For our model, we derive the prerequisite relationships from the grammar of the programming language. Typically, one PP is composed of several different components. Each of these components is a prerequisite for the KC represented by the PP if the component is obligatory. For example, an iteration statement consists of an expression representing the looping condition and a block statement. As such, the KCs *expression* and *block* are prerequisites for the KC *iteration*. Since the number of parameters of a node increases exponentially with the number of its parents, we do not consider relationships of KCs which are often used with another KC, e.g., *increment/decrement* with *for*, to keep complexity manageable.

A *granularity relationship* describes a hierarchical structure between KCs. According to Millán and Pérez-de-la-Cruz [5], there are two alternatives to describe a granularity relationship in a DBN (see Figure 4.6). Let K be a KC which can be refined by KCs K_1, K_2, and K_3. Alternative 1 describes that K is composed of three independent subtopics which have to be known in order to know K. An evidence about the state of a sub-KC does not change the belief about the knowledge of the other sub-KCs. Alternative 2 describes that KCs K_1, K_2, and K_3 are all specializations of KC K. Evidence about the knowledge of one sub-KC also increases the belief that the other sub-KCs are known. We have defined different levels of refinement of KCs (see Section 4.1.1). For example, the KC *iteration* is refined by KCs *while*, *do-while*, and *for*. We would assume that a student who is able to apply a while loop also has a high probability of applying a do-while loop correctly since he/she showed that he/she basically understood the concept of an iteration. For this reason, alternative 2 is more appropriate in our use case to model granularity dependencies.

To make the definition of parameters easier, all prerequisites are summarized by a noisy-AND gate (see Section 2.4.2). Then, the CPT for each KC can be defined conditioned on the knowledge of the prerequisites and on the more coarse-grained KC [134].

Example 4.7 *Figure 4.7 shows an example for a domain model. We have two level 0 KCs* statement *and* expression*. Statement is refined by two level 1 KCs* iteration *and* block*. Expression is refined by a level 1* KC binary*. Iteration is refined by three level 2 KCs* while*,* do-while*, and* for*. The KCs* block *and* expression *are prerequisites for the KC* iteration *and are summarized in the auxiliary node P.*



(a) Alternative 1 (b) Alternative 2

Figure 4.6.: Two alternatives for representing granularity relationships [5]

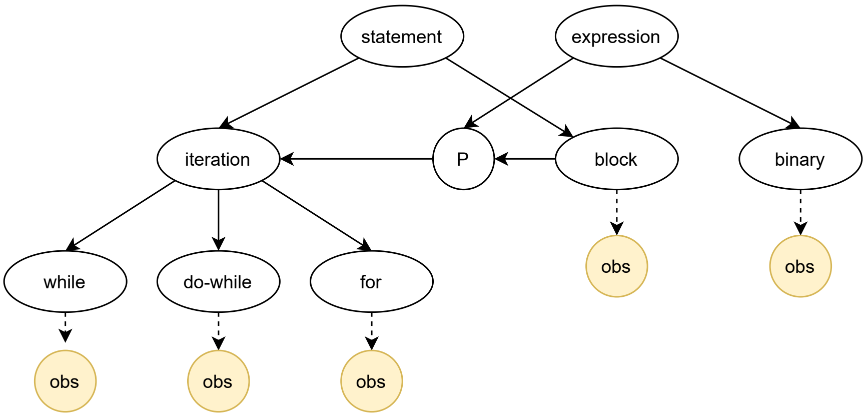


Figure 4.7.: Example for a domain model

	$req = 0$	$req = 1$
$item = correct$	1	$P(obs = correct)$
$item = incorrect$	0	$P(obs = incorrect)$

Table 4.6.: CPT of the *item* node

KC-item mapping

To be able to cope with a dynamic Q matrix, we introduce binary evidence nodes *req* which describe whether a KC is required in a step according to the Q matrix or not. For each higher level KC, a binary virtual evidence node *obs* is defined. This node describes whether the KC was applied correctly or incorrectly. We use a virtual evidence node to account for multiple occurrences of a KC. A virtual evidence describes evidences which are not completely certain, i.e., may have other values than 0 or 1. For example, if a KC was applied 2 times correctly and 1 time incorrectly in a solution, we set the evidence to 0.66. We use noisy-AND gates without leakage to connect the *req* and *obs* nodes. We call them *item* since they reflect the contribution of a KC to the overall result of the solution. If the KC is not required, the item is assumed to be correct. If the KC is required, the probability that the item is correct corresponds to the probability that the student applies the required KC correctly (see Table 4.6).

Result composition

All item nodes are combined by a noisy-AND gate which represents the probability that all required KCs were applied correctly. An evidence node *result* describes whether the exercise was solved by the student correctly or not. The probability that a student solves an exercise correctly depends on the probability of applying all required KC correctly and by composing them correctly which we explicitly model by a node *problem solving ability*. The problem solving ability is a latent trait, i.e., not directly observable.

Inter-time-slice edges

So far, we only defined intra-time-slice edges, i.e., the DBN structure within a time slice. As discussed by Millán and Pérez-de-la-Cruz [5], we use inter-time-slice edges (K^t, K^{t+1}) for each of the highest level KCs K and for the problem solving node. This corresponds to the learning progress of a student like it is modeled in BKT (see Section 2.5.1). There are no inter-time slice edges for lower level KCs since their value is already updated by propagation from the higher level KCs.

4.4. Learning Curves from DBNs

A learning curve describes how the performance of a student (or person in general) changes with the number of practice opportunities [135]. It can be used to assess a student's ability to acquire a new skill or for improving a course by identifying problematic learning curves. To create a learning curve, the knowledge or ability of a student has to be estimated. AFM is a widely used skill model which also defines different types of learning curves [136]. However, AFM is a model based on logistic regression. Logistic regression assumes all parameters and observations to be independent of each other. Nevertheless, in programming KCs often are dependent. Since DBNs are able to deal with dependencies, we propose a learning curve based on estimates of a DBN.

We can use the DBN we defined in Section 4.3.3 after training to track a student's knowledge evolution. To achieve this, the evidental nodes like the *req* and *obs* nodes are set accordingly to a student's data record. Then inference is used to update the state of each node. The KC nodes tell us the student's knowledge state of a KC, i.e., the probability that the student knows the KC, at a certain step. The trajectory of these probabilities describes the student's learning curve of that KC.

To get a generalized learning curve for the complete course, we average over all students. However, students may have a different number of steps if we consider all of their submissions as a single step. There are four possibilities how to deal with the different sequence lengths:

1. *Considering only the first attempt:* As we already discussed in Section 4.3.1, the first submission of a student is often an incorrect solution and may underestimate a student's knowledge.

2. *Considering only the last attempt:* In contrast to the first attempt, the last attempt is often a correct solution and may overestimate a student's knowledge.

3. *Considering all attempts:* The advantage of DBNs is that we can even get an estimation for time steps that we do not have observed yet. The problem of that approach is that the sequences are not synchronous anymore. For example, student 1 submits in total 30 solutions to the first 5 exercises. Those exercises are mainly dealing with output, constants, and branching. His/her estimated knowledge of other KCs will be quite low since he/she has not practiced on them yet. Student 2 also submits 30 solutions, but on all 30 exercises which involve all KCs. So the knowledge of the students at each step is not comparable.

4. *Considering all attempts, averaging over exercises:* To avoid the asynchronicity, we consider each exercise as a step and average over all knowledge estimates for that

exercise. In this case, we steer a middle course between underestimation of the first and overestimation of the last attempt.

5. Analysis of Programming Errors

In this thesis, we distinguish between *error types* and *error categories*. Error types describe concrete instances of errors, e.g., a missing semicolon, off-by-one-error, etc. Error categories are more abstract and are defined by a certain viewpoint on the errors (see Section 3.1). For example, from the viewpoint of *when* the error occurs, we can distinguish between the error categories run-time and compile-time errors. In this section, we describe how we analyze the students' programming errors. We start with a description of our manual inspection of students' code and how we identify error types in Section 5.1. We present the error categories we used in this thesis and how we assigned them to error types. Since we are not only interested in the types of errors and their frequencies, i.e., how often they occur, we introduce further metrics for error measurement in Section 5.2. In the end, we describe how we use the collected metrics to identify patterns in the error development with the help of clustering in Section 5.3.

5.1. Definition of Error Types and Error Categories

McCall and Kölling show that diagnostic messages of compilers cannot be mapped one-to-one on actual errors. They can not only be imprecise but also inaccurate [88]. Furthermore, errors in compilable solutions cannot be located automatically in general since this would require an automatic debugging. Therefore, we have to inspect all solutions manually.

In non-compilable solutions, we use the diagnostic message of the compiler as a hint where to look for the error. For compilable solutions, we use an automatic AST-based comparison between the erroneous solution and the most similar correct solution to highlight tentative error locations. Similarly to McCall and Kölling [88], the manual analysis is performed in two steps. First, we go through all solutions and identify the erroneous part of the code. We determine which code parts belong to one error and create the error types on-the-fly. If we have not defined a fitting type for an error yet, we add a new one to our error type list. In the second step, we revise the error types. For each defined error type, we have a second look on the assigned solutions. We change types of errors, if another type fits better, group similar error types and refine others. In this step, we also define which error categories fit the error type.

For the categorization, we use the categories by Bayman and Mayer [54] and extend them by three additional categories [137], since we came across errors which did not fit into the others:

- *Syntactic:* Syntactic knowledge refers to the syntax or rules of a programming language. Typical syntactic errors are, e.g., using commas instead of semicolons in a for-loop header or undeclared variables.
- *Conceptual:* Conceptual knowledge refers to knowledge about how a programming construct works. This category contains semantic errors as well as logic errors due to a misconceptions of a programming construct. A typical conceptual error is, e.g., an uninitialized variable because students think the variable is implicitly initialized with 0.
- *Strategic:* Strategic knowledge refers to the way to assemble programming constructs to achieve a certain goal. Strategic knowledge is often also referred to as *problem solving ability*. A typical strategic error is, e.g., an omitted boundary case.
- *Sloppiness:* Sloppiness refers to errors made unintentionally. Typical errors are, e.g., forgetting a semicolon or typos.
- *Misinterpretation:* Misinterpretation means that a student has another interpretation of the exercise than intended. For example, the task is to continuously read-in strings and immediately output the number of occurrences for each character. Instead, the student writes a program that first reads-in a limited number of strings in an array and just afterwards outputs the character occurrences for each string.
- *Domain:* Domain errors refer to errors made due to the lack of background knowledge required for solving the exercises. For example, if the exercise is to calculate the average of entered numbers, but the student thinks the average is always a division by 2.

Usually we can not tell the actual cause of an error, such that an error can belong to more than one category. For example, an off-by-one error can be conceptual if the student has a misconception about the stop of a loop, strategical if the student has an incorrect solving strategy for a problem, or sloppiness if a student unintentionally typed < instead of <= [137].

5.2. Error Landscape

The term *error landscape* was first introduced by Smith and Rixner [107] and refers to a set of metrics used in the analysis of programming errors. In our work, we consider the frequency, duration, severity, and re-occurrence rate of errors.

Similar to McCall and Kölling [138], we associate each error with a certain state during the manual analysis. We distinguish between three states:

- *New:* The state *new* refers to errors which are newly introduced, i.e., the first occurrence of an error while solving a particular exercise. Multiple occurrences of the same error type but in different contexts are counted as separate errors.
- *Unresolved:* When an error is still present in subsequent submissions, it has the state *unresolved*.

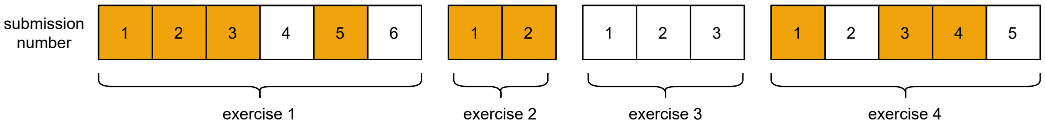


Figure 5.1.: An example of a submission sequence for a certain error. Orange means that the error is present in the submission.

- *Resolved:* When an error is not present in a submission and all subsequent submissions for that exercise, the state of the error is *resolved*.

When an error is not present in one submission anymore, but re-occurs in a subsequent submission, the state of the error remains unchanged because it is simply considered as reverted but not really resolved.

Example 5.1 *Consider the example of a submission sequence of a student for a certain error in Figure 5.1. For exercise 1, the student submitted six solutions. The error was present in submission 1-3, and 5. The state of the error was* new *in submission 1,* unresolved *in submission 2,3, and 5, and* resolved *in submission 6.*

5.2.1. Error Frequency

The *error frequency* f refers to the total number of occurrences of a certain error type t. We can distinguish between submissions and solutions. A *solution* is a student's attempt to solve a particular exercise and consists of a sequence of *submissions*. An error can occur multiple times in a solution at different places in the code. Each occurrence in a single submission is counted separately for the frequency. Furthermore, the error may occur in subsequent submissions if it was not resolved. The re-occurrence of an error in subsequent submission is only counted as one instance. This means, that the error frequency $f_{s,t}$ of a certain error type for a certain student s is defined by the number of errors with error type t and state *new*.

Definition 5.1 (Error frequency) *Let S be the set of students, T be the set of error types, E be the set of exercises, $Sub_{e,s}$ be the set of submissions a student s makes to exercise e, $state(err, sub_i)$ describe the state of an error err in the i-th submission by student s to a particular exercise e, and $type(err)$ denote the type of an error err.*

Then the error frequency *is defined as:*

$$f_{s,t} = \sum_{e \in E} \sum_{sub_i \in Sub_{e,s}} |\{err \mid err \in sub_i, state(err, sub_i) = new\}, type(err) = t\}|$$

We can aggregate the frequencies to get a student-based frequency f_s *which describes how many errors a particular student* $s \in S$ *made*

$$f_s = \sum_{t \in T} f_{s,t}$$

or to get a type-based frequency f_t *which describes how often an error of a particular type* $t \in T$ *was made*

$$f_t = \sum_{s \in S} f_{s,t}$$

Example 5.2 *Consider the submission sequence for an error in Figure 5.1. The frequency of the error is 3 since it occurs in exercise 1, 2, and 4.*

5.2.2. Error Duration

The *error duration* (or also called *time-to-fix*) $d_{s,t}$ describes how long a student s needs on average to fix an error of a certain error type t. We define the duration d_{err} of an error err in terms of the number of submissions between the first occurrence of an error and its resolution:

$$d_{err} = j - i \tag{5.2.1}$$

where i and j define the submission numbers with $state(err, sub_i) = new$ and $state(err, sub_j) = resolved$.

However, not all errors are resolved. We define the default duration $d_{default,e}$ of an unresolved error as the sum of the arithmetic mean μ_e and the standard deviation σ_e of the number of submissions for an exercise e:

$$d_{default,ex} = \mu_e + \sigma_e \tag{5.2.2}$$

But if a student s already has more submissions than the calculated value, the duration is set to number of submissions, i.e., we assume that the student would have resolved the error in the next submission:

$$d_{err} = max\{d_{default,e}, |Sub_{s,e}|\} \tag{5.2.3}$$

where $Sub_{s,e}$ denotes the set of submissions of student s to exercise e.

Definition 5.2 (Error duration) *Let* S *be the set of students,* T *be the set of error types,* $Errors_s$ *be the set of errors made by student* s*,* $type(err)$ *denote the type of an error* err*, and* d_{err} *denote the duration of the error* err*.*

Then the error duration *is defined as:*

$$d_{s,t} = \sum_{err \in Errors_s} \{d_{err} \mid type(err) = t\}$$

We can aggregate the durations to get a student-based duration d_s *which describes the average error duration of student $s \in S$*

$$\frac{\sum_{t \in T} d_{s,t}}{f_s}$$

or to get a type-based duration d_t *which describes the average duration an error of a particular type $t \in T$*

$$\frac{\sum_{s \in S} d_{s,t}}{f_t}$$

Example 5.3 *Consider the submission sequence for an error of type t in Figure 5.1. The duration of the error is $d_{err,1} = 6-1 = 5$ for exercise 1 and $d_{err,3} = 5-1 = 4$ for exercise 4. If we assume that $d_{default,2} < 3$, then $d_{err,2} = 3$ for exercise 2, otherwise it is set to $d_{default,2}$ because it means that the student just "gave up" solving the error. The average duration of an error of type t for student s would be*

$$d_{s,t} = \frac{5+4+3}{3} = 4$$

5.2.3. Error Severity

The *error severity* $s_{s,t}$ describes the mean effort that a student s requires for removing all errors of type t. It is calculated as the product of the error frequency $f_{s,t}$ and the duration $d_{s,t}$ of an error.

Definition 5.3 (Error severity) *Let S bet the set of students and T be the set of error types. Then the* error severity *is defined as*

$$s_{s,t} = f_{s,t} * d_{s,t}$$

Accordingly, we get the student-based severity

$$s_s = f_s * d_s$$

and the type-based severity

$$s_t = f_t * d_t.$$

Example 5.4 *Consider the submission sequence for an error of type t in Figure 5.1. From the frequency $f_{s,t}$ that we calculated in Example 5.2 and the duration $d_{s,t}$ that we calculated in Example 5.3 we obtain the severity $s_{s,t} = 3 * 4 = 12$.*

5.2.4. Error Re-occurrence

The error re-occurrence describes how likely it is that a certain error will be done again, once it has been made by a student.

Definition 5.4 (Error re-occurrence) *Let S be the set of students and T be set of error types. We distinguish between* student-based error re-occurrence

$$r_s = \frac{\{t \in T \mid f_{s,t} > 1\}}{\{t \in T \mid f_{s,t} > 0\}}$$

and type-based re-occurrence

$$r_t = \frac{\{s \in S \mid f_{s,t} > 1\}}{\{s \in S \mid f_{s,t} > 0\}}$$

The student-based re-occurrence is the mean error repetition rate of a student independent of the concrete error type. The type-based re-occurrence rate however defines how likely it is that an error of a certain type is repeated without regarding the concrete student.

5.2.5. Category-based Error Landscape

Summed up, we can distinguish between a student-based error landscape and a type-based error landscape. Since we might also be interested how the error landscapes looks like for certain error categories, we can obtain a *category-based* landscape by aggregating over the different categories.

Definition 5.5 (Category-based landscape) *Let $C_i \subseteq T$ be a category defined by a set of error types $t_i \in T$. Then the* category-based frequency *f_{C_i} is defined as:*

$$f_{C_i} = \sum_{t \in C_i} f_t$$

Accordingly, the category-based duration f_{C_i} and re-occurrence rate r_{C_i} are defined:

$$d_{C_i} = \frac{\sum_{t \in C_i} d_t}{|C_i|}$$

$$r_{C_i} = \frac{\sum_{t \in C_i} r_t}{|C_i|}$$

An error type can fall into several categories. We assume that these categories are uniformly distributed, i.e., if an error type t can belong to n_t categories, the probability that the error is a result of a certain category is $p_t = \frac{1}{n_t}$. When we want to determine the probability that a certain error category is responsible for a solution to fail, we use the intersection rule by multiplying the probabilities of all errors in that solution.

Example 5.5 *A solution has 3 errors e_1, e_2, and e_3. Errors e_1 and e_2 are of type t_1 and error e_3 is of type t_2. We want determine the probability that the solution is incorrect just because of sloppiness. Error type t_1 can be due to sloppiness or misinterpretation, thus $p_{t_1} = \frac{1}{2}$. Error type t_2 is assigned to the categories sloppiness, strategic, and misinterpretation, thus $p_{t_2} = \frac{1}{3}$. So the resulting probability for sloppiness as the only cause is*

$$\frac{1}{2} * \frac{1}{2} * \frac{1}{3} = \frac{1}{12}$$

5.3. Clustering Error Curves

When analyzing the error landscape, we get for each student the frequency of errors he/she makes and the duration of these errors. Plotting the students error frequency and duration over the time gives us an error curve, describing how the student's errors change over time. This is not applicable to the re-occurrence rate, since it is already a metric considering an error's occurrence over time.

To find similar error patterns between students, we need to cluster the error curves of the students. We use k-means clustering [139] for that purpose, because the centroid gives us a representative error curve for each cluster.

We use the category-based frequency and duration for each exercise as feature set, since we are interested whether students differ in the categories of errors and when they do them. Since many students have not completed all 30 exercises of the course, we add an additional feature which describes the number of missing exercises. Since the duration often is much higher than the frequency, we scale the data before clustering.

6. Case Studies

We use data from an introductory C programming course to perform three case studies aiming at answering our research question. Each case study is targeting one major research question. First, we present how we collected data with SmartAPE in Section 6.1. Afterwards, we present the setup and results for each of our case studies.

6.1. Data Collection with SmartAPE

As a basis for our case studies, we use data that we collected in SmartAPE (Smart Assessment of Programming Exercises) [140]. SmartAPE is an online assessment tool where students' programming solutions are automatically evaluated. Figure 6.1 shows the general structure of SmartAPE. The following description of SmartAPE is taken from our previous publication [141].

The student can submit his/her solution to a programming exercise via a web interface. The solution is then compiled using gcc [142] as well as clang [143]. We decided two integrate both compilers since we observed that they return different results regarding semantic checks and warnings. Furthermore, students can get two different descriptions of an error and can decide which one fits them better. In addition, we integrated several static analysis tools. Splint [144] is a tool for checking C programs for security vulnerabilities and coding mistakes. The drawback of splint is its high false positive rate. Therefore, we added cppcheck [145] and clang-analyzer [146] which do not detect as many errors as splint, but have a lower false positive rate.

Since it is not only important to write correct but also readable code in programming, we integrated vera++ [147] to check whether code conventions are complied. If the code is compilable, we run dynamic tests to check whether the code does what is intended to do. We use cunit [148] for testing single functions and dejagnu [149] for testing a complete program. Dejagnu is a framework for testing programs independently of the programming language by comparing a program's output to an expected output via regular expressions. If the execution of a test case takes longer than 10 seconds, a timeout is reported. Since the expected programs are very simple and should have a small execution time, a timeout indicates an infinite loop in most cases. Due to security reasons, the evaluation of the students' code runs on a different server such that is separated from the web frontend and the data base. In addition, the dynamic tests of each solution run in a separated linux

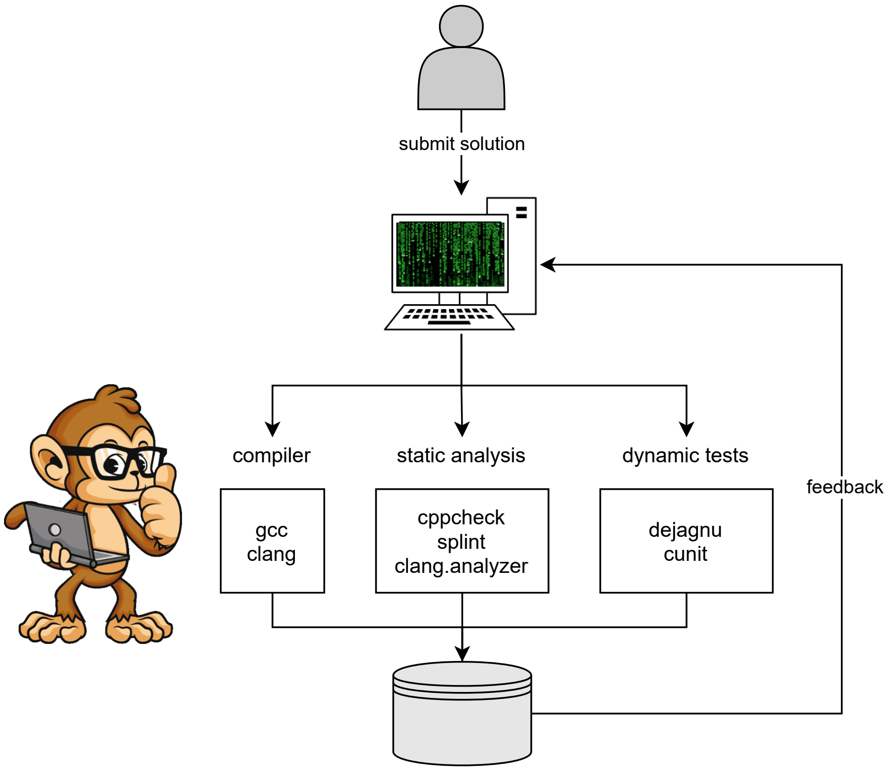


Figure 6.1.: Structure of SmartAPE

container [150]. Compiler messages, static analysis reports and the results of the dynamic tests are parsed and stored in a data base.

The data we use is from our introductory C programming course from the winter term 2016/2017. The students in this course are from Computer Science as well as from other disciplines like, e.g., Physics or Biology. In total, we have 21546 submitted solutions by 281 students to 30 exercises of which 18982 are compilable and 5681 labeled as correct.

6.2. Case Study 1: Skill Models for the Programming Domain

The first case study deals with skill models in the programming domain. We investigate how different students' solutions actually are, how meta-parameters affect skill models, and compare several skill models including DBNs as introduced in Section 4.3.3.

6.2.1. Setup

The main goal of this case study is to evaluate different skill models for the programming domain. For skill modeling the identification of a Q matrix, and thus KCs, is essential. We use KAMs (see Section 4.1.3), which describe a student's program in terms of KCs. Since, we are only able to construct KAMs for compilable code, we remove non-compilable solutions from the data set for this case study. Figure 6.2 shows how we set up the first case study.

Preparation of required data

Using the submitted code of students from SmartAPE, we construct an AST for each solution. We transform each AST into KAMs of different KC levels. We determine for all solutions pair-wisely their similarity as described in Section 4.2. We require the similarity for answering RQ 1.1 and certain definitions of the Q matrix. From the KAM, we can extract which KCs on syntax level the student used and how often it was used in the solution. In addition, we use the AST for the determination of variable roles as additional KCs (see Section 4.1.2). A record of the resulting raw performance data set consists of a student id, an exercise number, the step number, for each KC how often it was applied, and a binary value indicating whether the solution was solved correctly (see Table 6.1).

Creation of meta-parameterized data sets

In the next step, we have pre-processed the raw performance data by removing columns, i.e., features, which were always 0 and correspond to KCs which were not used in any solution, e.g., ATOMIC, REGISTER, INITIALIZATION_LIST, etc. Furthermore, logistic regression assumes parameters to be independent of each other. However, in our data

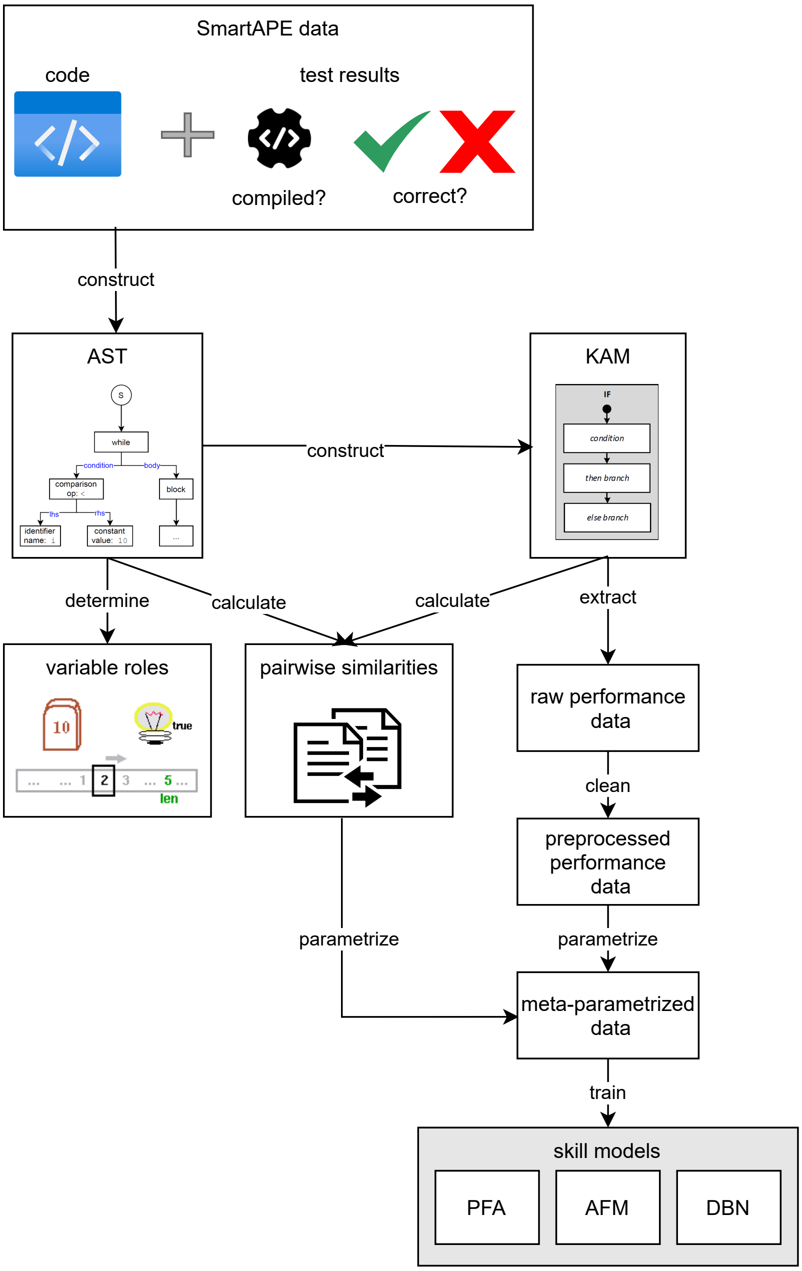


Figure 6.2.: Overview of the setup for case study 1

student	exercise	step	KC_1	KC_2	KC_3	...	KC_n	correct
1	1	1	1	1	2	...	2	1
1	2	2	2	1	3	...	5	1
1	3	3	1	2	2	...	3	0
2	1	1	1	1	1	...	2	1

Table 6.1.: Example for a basic data set

Parameter name	Values	Influence on
KC level	0, 1, 2, 3	number of features
minimum steps	0, 5, 10, 15, 20	number of observations
step definition	first, last, every	number of observations
incorrect KCs	all, diff	value of f_{ij}
KC count	binary, multiple	value of s_{ij} and f_{ij}
Q matrix	all, shared, union, common, used, set, KAM	value of r_{ij}

Table 6.2.: Summary of model parameters

singularities occurred because some KCs always occur together and are, therefore, linearly dependent. In pre-processing, we resolved such singularities by removing the dependent parameters.

As already stated in Section 4.3.1, there are several meta-parameters we can adjust to fine-tune skill models. From the pre-processed performance data set and additional information from the KAMs, we extracted data sets for all combinations of the parameters from Table 6.2. For the KC level meta-parameter, we selected the refinement levels defined in Section 4.1.1 as KC levels. As parameter values for the meta-parameter *minimum steps*, we selected the values 0, 5, 10, 15 and 20.

A record of an extracted data set contains the student id, the exercise id, the step number, for each KC a binary value indicating whether the KC is defined as *required* in that model r, the counts of previous successes s and failures f for that KC, and whether the solution was solved correctly (see Table 6.3). In total, we got 1680 extracted data sets where each one represents a single combination of meta-parameters.

Since the meta-parameters *step* and *minimum steps* have an influence on the number of records, i.e., observations, we provide a summary on the data set characteristics in Table 6.4.

student	exercise	r_{i1}	s_{i1}	f_{i1}	r_{i2}	s_{i2}	f_{i2}	r_{i3}	s_{i3}	f_{i3}	...	r_{in}	s_{in}	f_{in}	correct
1	1	1	0	0	1	0	0	1	0	0	...	1	0	0	1
1	1	1	1	0	1	1	0	1	2	0	...	1	2	0	1
1	2	1	3	0	0	2	0	1	5	0	...	0	7	0	0
1	3	3	1	1	2	2	1	5	2	...	1	7	3	1	
2	1	1	0	0	1	0	0	1	0	0	...	1	0	0	1

Table 6.3.: Example for an extracted data set (minimal steps=0, KC count=“multi", wrong KCs=“all")

step		minimum steps				
		0	5	10	15	20
first	# solutions	5534	4222	3045	1973	1022
	# correct solutions	2119	1635	1288	826	432
	# incorrect solutions	3415	2587	1757	1147	50
	# students	280	245	223	199	173
last	# solutions	6152	4832	3644	2565	1158
	# correct solutions	5047	3861	2869	1970	1167
	# incorrect solutions	1105	971	775	595	391
	# students	281	246	223	206	194
every	# solutions	18982	17629	16360	15121	13945
	# correct solutions	5681	5136	4713	4291	3914
	# incorrect solutions	13301	12493	11647	10830	10031
	# students	281	260	249	238	233

Table 6.4.: Data set summary

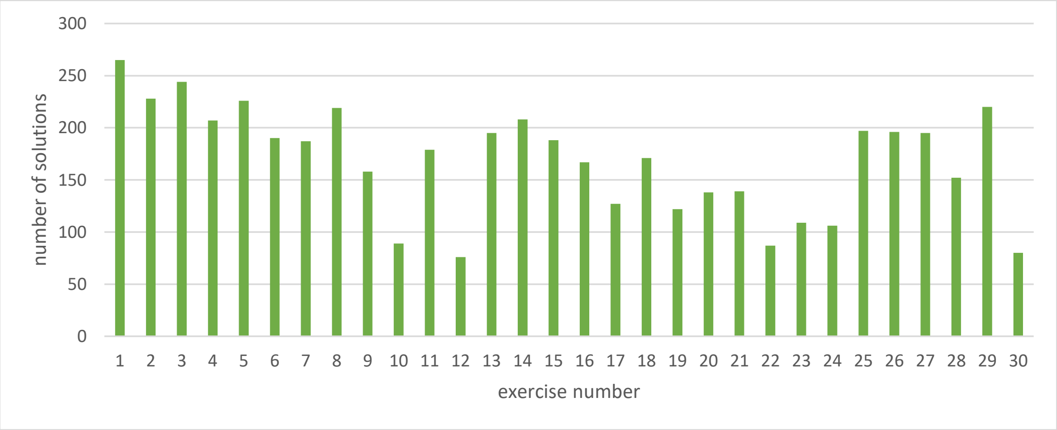


Figure 6.3.: Number of correct solutions for each exercise

Training skill models

The resulting meta-parameterized data sets are used to fit PFA and AFM models by means of logistic regression as defined in Section 4.3.2. We manually define the structure of DBN models as defined in Section 4.3.3 using the GeNIe Modeler [151]. The GeNIe Modeler is a graphical editor which allows defining, learning, and evaluating probabilistic network models. For fitting the parameters of the DBN we use GeNIe's SMILE [152] engine and its wrapper for Java.

6.2.2. Evaluation Criteria

Evaluation of similarity

As students were able to submit multiple solutions for each exercise, we only take the last submission to an exercise which was identified as correct for the evaluation of code variance. We consider only correct solutions since incorrect solutions can be arbitrary far away from a valid solution and would, thus, distort the results. We take only the last correct solution into account since subsequent solutions by the same student typically only contain small changes. Figure 6.3 shows that we received between 80 and 265 correct solutions for each exercise.

Evaluation of meta-parameters

While meta-parameters like KC level, KC count, incorrect KCs, and Q matrix only influence the feature selection and processing, minimum steps and step definition change the data set. But to make models comparable, all models have to be tested on the same data. Since our skill model shall be applicable on each step a student makes, we use data for every step. The same holds for minimum steps. Usually, we want to apply the model directly from the beginning. But since minimum steps=20 is a subset of, e.g., minimum steps=15, we need to be careful when creating test and training sets. Therefore, we divide our data set into step slices: steps 1-4, 5-9, 10-14, 15-19, and >20. We use subject-based 5-fold-cross validation, i.e., students are distinct in each fold, to divide the data of each step slice into 5 folds. From these folds, we assemble our test data by selecting one fold from each step slice. The remaining folds can be used for training. For each value of minimum steps, we select those step slices which correspond to the value, e.g., if minimum steps is 15, tentative training data are the remaining folds of the data slices for steps 15-19 and >20. To get nearly equally-sized training sets, we randomly choose a sample of the tentative training data as training set.

To answer RQ 1.2, we use the meta-parameterized data sets to fit different skill models and compare them to each other. We decided to use PFA and AFM as they can be relatively quickly fitted and can deal with all of our proposed meta-parameters. Since these skill models are rather similar, they can help us evaluating how meta-parameterization affects the prediction behavior in similar models. As a third skill model, we select a completely different skill model type which uses the proportion of previous successes by a student on a KC which we call the proportional model (PPM). The probability that a student i knows a certain KC k is $\frac{s_{ik}}{s_{ik}+f_{ik}}$. Having the probabilities for each KC, the problem is now to determine the probability of answering correctly to a certain learning item. For BKT there exist two common approaches for determining the overall probability: multiplying the probabilities of all required KCs or assuming that the overall performance corresponds to the knowledge of the “weakest" KC, i.e., with the lowest probability [153]. Since the first approach is based on the joint probability and assumes independence between the KCs which is not given for programming KC, we have chosen the second approach for our PPM. The probability that a student i solves an exercise j correctly is:

$$p = \min_{k \in KCs(j)} \frac{s_{ik}}{s_{ik} + f_{ik}} \tag{6.2.1}$$

This model has no parameters that have to be learned and, thus, has not to be trained but is directly applicable. In so far, the definition of a step is not required for that approach, since in practice we would use the prediction on each submission of the student. The same also holds for the meta-parameter *minimum steps*. In that model the step definition is always set to *every* and minimum steps is set to 0. We have not used models based on Bayesian theory for this research questions since we would have to fit a huge amount of models with a high number of parameters which would result in unmanageable computational effort.

We fit the models for all combinations of the meta-parameters, i.e., 1680 AFM models and 1680 PFA models in total, and use that adapted version of 5-fold cross validation. For PPM models, we get 112 models in total because we do not consider the meta-parameters

minstep	count	incorrect	Q matrix	step	KC level			
					0	1	2	3
0	binary	all	shared	all	0.636	0.688	0.583	0.663
5	binary	all	shared	all	0.632	0.675	0.587	0.596
10	binary	all	shared	all	0.634	0.674	0.59	0.594
15	binary	all	shared	all	0.633	0.672	0.571	0.565
20	binary	all	shared	all	0.634	0.674	0.584	0.583
0	multiple	all	shared	all	0.634	0.686	0.706	0.702
5	multiple	all	shared	all	0.634	0.679	0.693	0.693
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 6.5.: Example of paired results of PFA models regarding AUC for meta-parameter *KC level*

step definition and *minimum steps*. Since we have strongly unbalanced data and the correct prediction of both classes is equally important in our case, we use the area under the curce (AUC) and the root mean square error (RMSE) as performance metrics which we average over all folds.

For the evaluation of the single meta-parameters, we split the result data for each meta-parameter by the parameter values. Table 6.5 shows an example of splitting result data (PFA AUC) for the meta-parameter *KC level* into four data sets. Since the data sets are not independent of each other because they were built on the same data, they are paired such that they distinguish only in the currently regarded meta-parameter value. As the other parameters are fixed for each row, the variation between the results in a row is solely explained by the currently regarded meta-parameter, i.e., in the example by the chosen KC level. Our result data do not fulfill the normal distribution assumption of ANOVA, thus we use Friedman test [154] and a pairwise Wilcoxon signed rank test [155] with Bonferroni correction [156] for checking whether the differences between the models are significant. To measure how much a parameter influences the result, we use Kendall's coefficient of concordance (Kendall's W) [157] which is proposed by [158] as effect size for the Friedman test. We distinguish between *small* ($W \leq 0.3$), *moderate* ($0.3 < W \leq 0.5$), *high* ($0.5 < W \leq 0.7$), and *very high* ($W > 0.7$) effects [159]. To evaluate the single parameter values, we calculate the effect size by Cohen's d for each pair of significantly different values. According to Cohen [160], $0.2 < d < 0.5$ describes a *small* effect, $0.5 \leq d \leq 0.8$ describes a *medium* effect, and $d > 0.8$ describes a *large* effect. Sawilowsky [161] extended the scale by $0.01 < d < 0.2$ as a *very small* effect, $1.2 < d < 2$ as *very large* effect and $d \geq 2$ as a *huge* effect.

Skill model comparison

With the evaluation of meta-parameters we determine which parameter values seem to be best. However, these parameters are determined among a broad variation of models. We want to investigate how good a model with taking the best separate parameters (“meta”) is in comparison to the overall best model (“best”) and in comparison to the model when it uses the most wide-spread meta-parameterization in previous work (“common”). Furthermore, we add the majority class model which always predicts the major class (in our data it is *incorrect*) as a baseline model, and also evaluate the performance of our DBN models.

For the “meta”-models we took the models with KC level=1, KC count=multiple, Q matrix=shared, incorrect KCs=diff, step definition=every and minimum steps=0. For the “common”-models, we used KC count=binary, Q matrix=used, incorrect KCs=all, step definition=every, and minimum steps=0. As the KC level was never mentioned in previous work, we provide the results for the worst and the best models (KC level in brackets).

We also fit a “meta”- and a “common” model for DBNs. A “best” model for DBNs is not possible since we do not perform a meta-parameter evaluation on them. Furthermore, we distinguish between DBNs with variable roles included and without variable roles.

We use the complete data set for training now and a new data set based on data from the winter term 2017/18 as validation set.

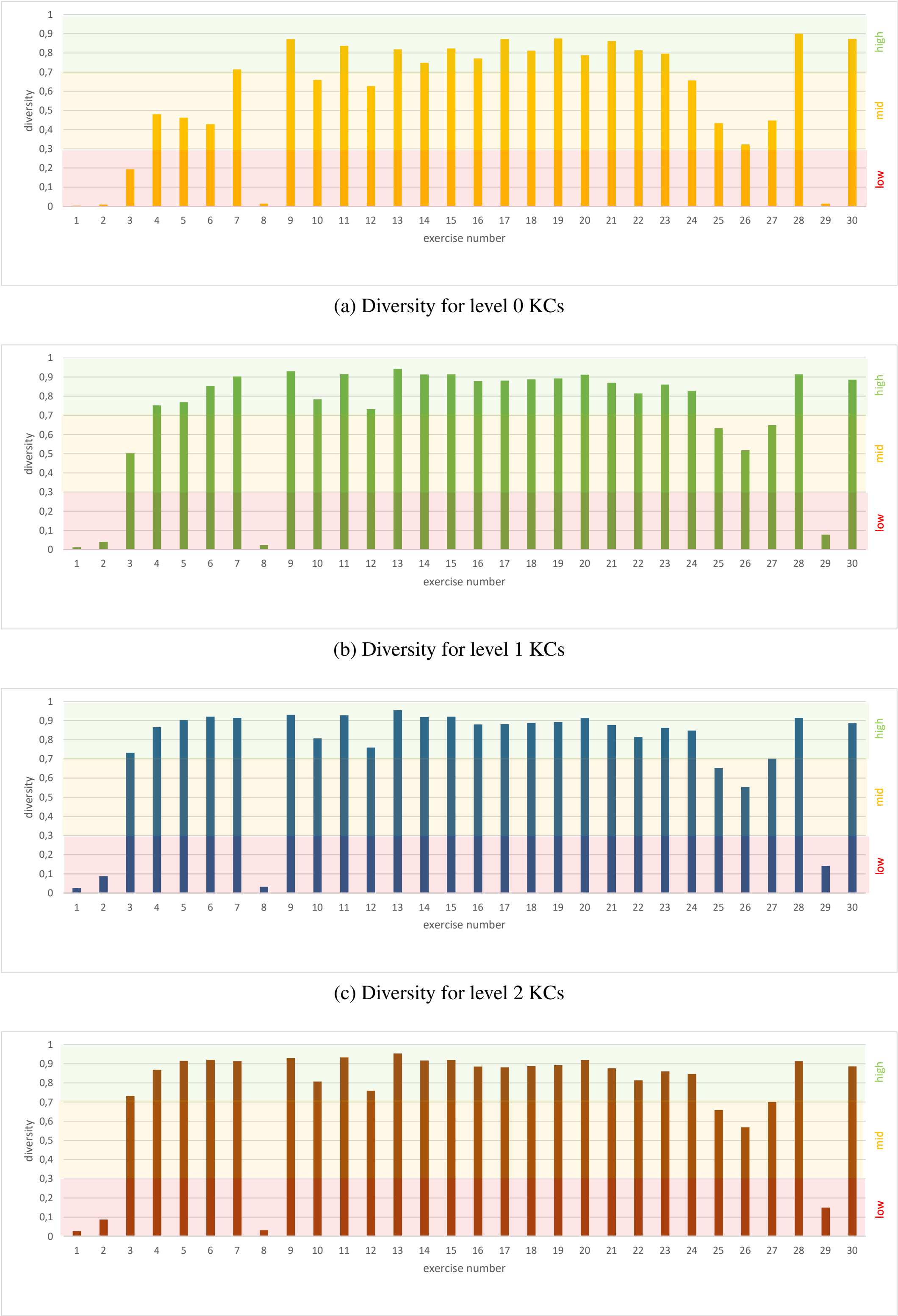
6.2.3. Results

Similarity

Figure 6.4 shows that the diversity as defined in Section 4.2.4 increases with higher KC levels. The average diversity for level 0 is about 0.6 which is surprising since we have not expected that much variation within only four KCs. There is almost no difference in diversity between level 2 and level 3 where over 2/3 of the exercises have a high (> 0.7) diversity. The average diversity of 0.72 on level 1 is only slightly lower than the average diversity on level 2 and 3 (0.75). More than half of the exercises have a high diversity on all levels. Four exercises (exercise 1, 2, 8, and 29) have a low diversity (< 0.3) on all levels. Those are exercises in which a more or less constant value has to be printed out, e.g., "Hello World!", MIN_CHAR and MAX_CHAR, sizes of primitive data types, etc. (see Appendix D).

From an average diversity of about 70% we can conclude that students actually tend to write different code.

To determine to which extent the solutions differ, we compared all correct solutions to each other. Figure 6.5 shows the distribution of the similarities for KC level 0 and 3. The complete results can be found in Appendix E.1. For set similarity on level 0, we can see that there is almost no difference between the solutions throughout all exercises. Except



(a) Diversity for level 0 KCs

(b) Diversity for level 1 KCs

(c) Diversity for level 2 KCs

(d) Diversity for level 3 KCs

Figure 6.4.: Diversity for each exercise

for exercises 25, 26, 27, and 28, the median similarity is 1. For level 1 KCs the median of similarities is about 0.87. Similar to the observations for diversity, there is no great difference between the similarity for level 2 and level 3 where the median is located at 0.8 on average. Exercises 1 and 8 have a median similarity of 1 on all levels. The minimum median value is about 0.67 for exercise 28, whereas the absolute minimum set similarity is 0.254.

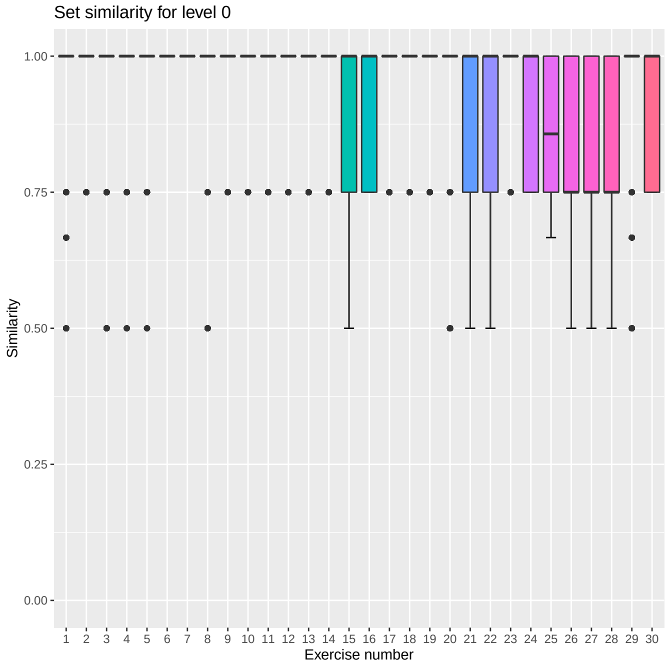
Figure 6.6 shows the distribution of the KAM similarities for KC level 0 and 3. The complete results can be found in Appendix E.1. The KAM similarity is smaller than the set similarity on all levels. However, the difference between the single levels is not that great. The mean median is 0.768 on level 0, 0.748 on level 1, 0.719 on level 2, and 0.717 on level 3. For two KAMs, differences in structure have a higher influence on the edit distance (> 1), and thus on KAM similarity, than differences in used KCs within a node (< 1). But as the structure basically remains the same on all levels and just the KCs get more fine-grained, this results in only small difference between the KAM similarities. Similar to set similarity, exercises 1 and 8 have a median KAM similarity of 1 on all levels. The minimum median is 0.617 which is the median of exercise 20. 14 out of 933403 compared solution pairs have a KAM similarity of 0. For exercises 2 and 29 the set similarity is smaller than the KAM similarity.

Figure 6.7 shows the distribution of the AST similarities. Since the AST similarity is not based on KCs but on AST nodes, we do not distinguish between different KC levels. However, the AST nodes are closely related to level 3 KCs. The AST similarity is smaller than the KAM similarity on level 3 except for exercise 25. The mean median is 0.689. The minimum median is 0.567 which is the median of exercise 20 which is the same as for KAM similarity. Also all solutions having a KAM similarity of 0, have an AST similarity of 0.

With an average set similarity of 0.97 on level 0 to 0.798 on level 3 and an average AST similarity of 0.689, the concrete differences between students' solutions are not that high.

Meta-parameter evaluation

Figure 6.8 shows the box-plots for the distribution of AUC and RMSE for the different meta-configurations of the skill models. The least achieved AUC for a PPM is 0.511 (KC level = *0*, KC count = *binary*, incorrect KCs = *diff*, Q matrix = *shared*), the greatest AUC is 0.639 (KC level = *3*, KC count = *binary*, incorrect KCs = *all*, Q matrix = *shared*). The median AUC is 0.589. For PFA models, the median AUC is similar with 0.586. However, the AUC varies between 0.5 (KC level = *2*, minimum steps = *5*, KC count = *multiple*, incorrect KCs = *all*, Q matrix = *all*, step definition = *first*) and 0.711 (KC level = *2*, minimum steps = *20*, KC count = *multiple*, incorrect KCs = *all*, Q matrix = *common*, step definition = *all*). For AFM, the range of the AUC is even larger between 0.5 (KC level = *3*, minimum steps = *20*, KC count = *multiple*, incorrect KCs = *diff*, Q matrix = *all*, step definition = *first*



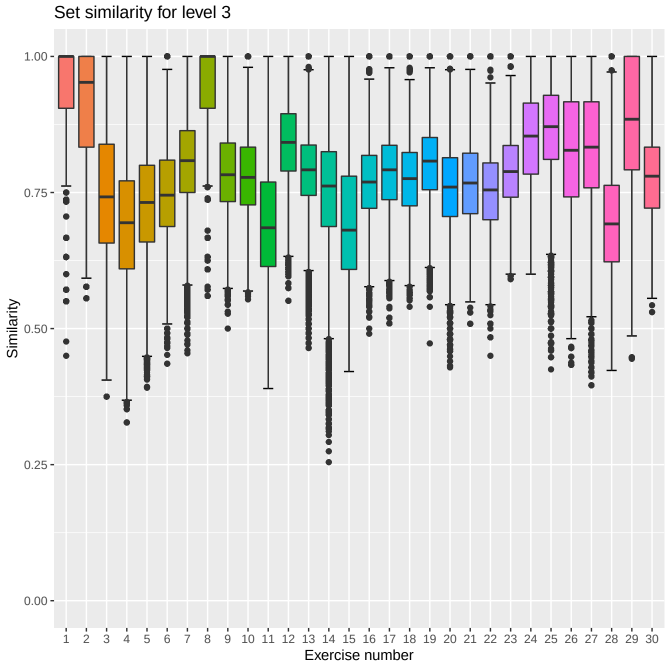


Figure 6.5.: Set similarity between solutions (level 0 and level 3)

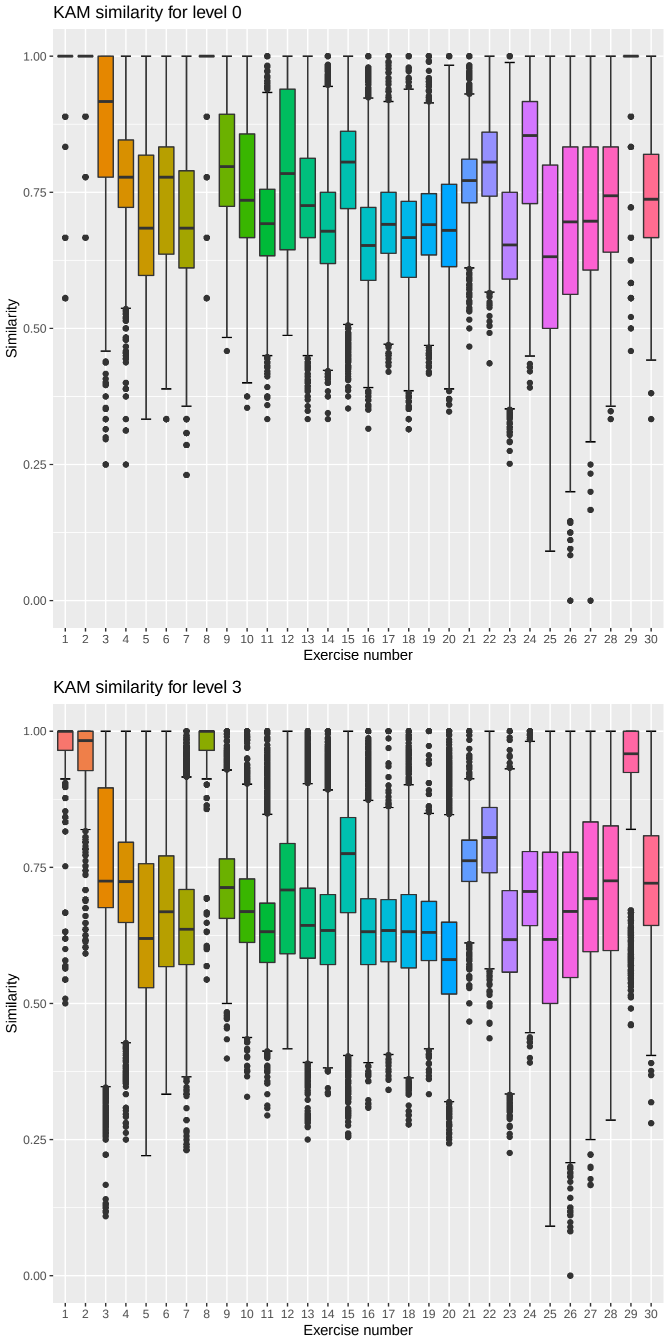


Figure 6.6.: KAM similarity between solutions (level 0 and 3)

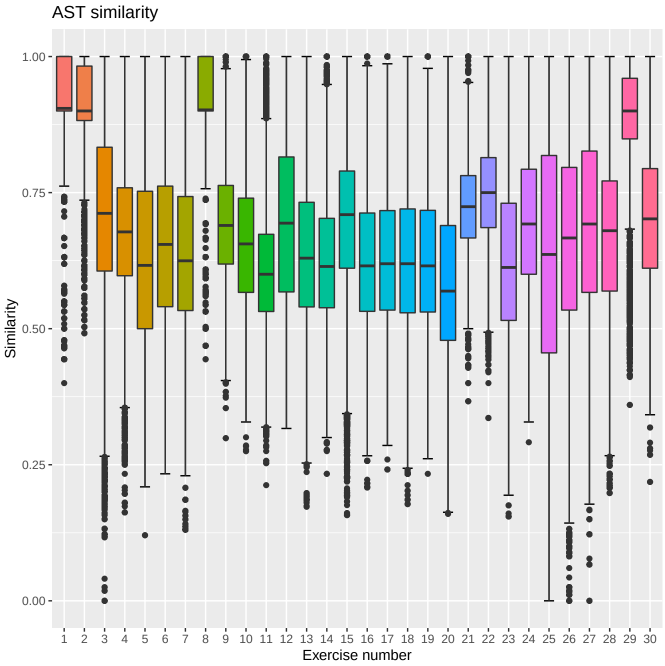
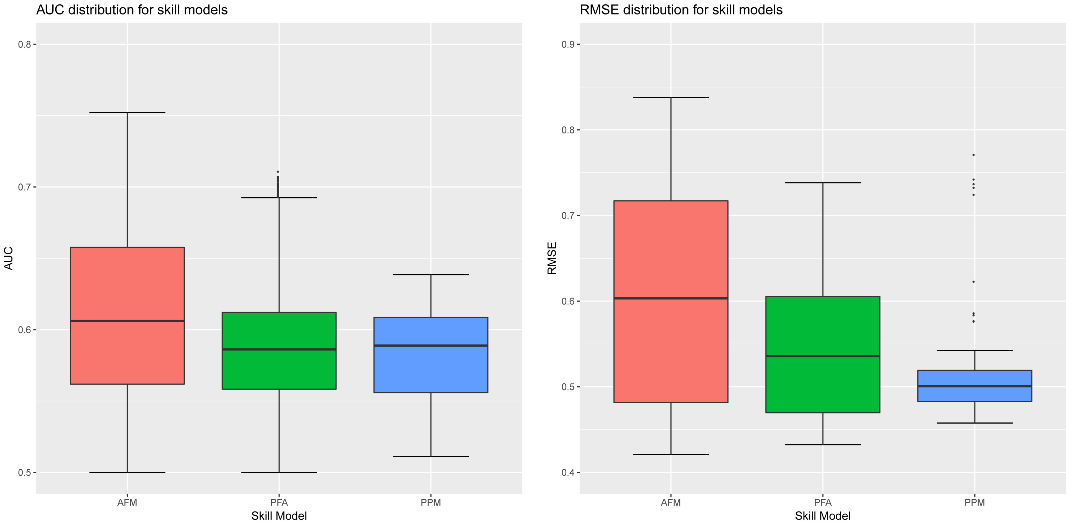


Figure 6.7.: AST similarity between solutions



(a) AUC

(b) RMSE

Figure 6.8.: Distribution of AUC and RMSE for AFM, PFA and PPM models with different meta-parameterization

and 0.752 (KC level = *3*, minimum steps = *0*, KC count = *multiple*, incorrect KCs = *diff*, Q matrix = *set*, step definition = *all*). A similar behavior of the models is also observable for RMSE. While the variance between different meta-configurations is comparably low for PPM models, the meta-configuration has a higher effect on the RMSE of the AFM which varies between 0.421 (KC level = *2*, minimum steps = *0*, KC count = *multiple*, incorrect KCs = *diff*, Q matrix = *set*, step definition = *all*) and 0.838 (KC level = *1*, minimum steps = *20*, KC count = *binary*, incorrect KCs = *all*, Q matrix = *set*, step definition = *first*).

How much a meta-parametrization affects performance metrics like AUC and RMSE depends on the concrete skill model. However, the effect is not negligible and can increase AUC by up to 0.3 and decrease RMSE by up to 0.4.

First, we compare the two similar models PFA and AFM. Table 6.6 shows the results of the comparison between meta-parameters for the metrics AUC and RMSE. The p-values of the Friedman test indicate that there is a significant difference (significance level 1%) of the results for the different meta-parameter values for each meta-parameter and each of the regarded metrics except for the meta-parameter *incorrect KCs* for AUC in PFA models and for AUC as well as RMSE for AFM models. We use Kendall's W to estimate which parameter has a high effect on the results. While the meta-parameter *KC level* has a high effect on the AUC for AFM models, its effect on the AUC of PFA models is only moderate.

	Friedman test p-value		Kendall's W	
parameter	AUC	RMSE	AUC	RMSE
KC level	8.785e-84	1.670e-136	0.308	**0.501**
minimum steps	1.072e-60	5.008e-59	0.213	0.207
step definition	1.752e-121	4.078e-170	0.497	**0.696**
incorrect KCs	0.011	5.035e-27	–	0.138
KC count	3.743e-47	1.476e-36	0.248	0.190
Q matrix	2.944e-65	9.375e-14	0.219	0.051

(a) PFA

	Friedman test p-value		Kendall's W	
parameter	AUC	RMSE	AUC	RMSE
KC level	6.845e-171	7.270e-89	**0.627**	0.327
minimum steps	1.572e-39	3.280e-74	0.140	0.259
step definition	1.565e-109	3.489e-187	0.447	**0.767**
incorrect KCs	0.011	0.013	–	–
KC count	5.163e-29	4.419e-22	0.149	0.111
Q matrix	1.263e-32	6.058e-16	0.113	0.058

(b) AFM

Table 6.6.: P-values of Friedman test and Kendall's W for all meta-parameters in the comparison between PFA and AFM

For the PFA model as well as for the AFM model, the meta-parameters *minimum steps*, *KC count*, and *Q matrix* have only a small effect and the parameter *step definition* has a moderate effect on the AUC.

We look in more detail at the meta-parameters *KC level* and *step definition* since they have at least on one model type a large effect. Furthermore, we analyze the meta-parameter *minimum steps*, since it is not applicable to PPM in further analysis. A complete listing of effect sizes for all meta-parameters can be found in Appendix E.2.

First, we look at models for different levels of KC (see Table 6.7). We use Cohen's d to measure the effect on the prediction power. Pairwise Wilcoxon signed rank test tells us that level 1 PFA models perform significantly better than all other levels ($p = .001$) regarding AUC (large effect). Regarding RMSE, level 1 also performs better than level 2 and level 3 PFA models (large-very large effect). Regarding RMSE, there is no significant difference between level 0 and level 1 PFA models which perform better than the other levels (very large effect). AFM models perform best when the meta-parameter *level* is set to 0. Thus, in the case of the meta-parameter *level*, the best meta-parameterization depends on the concrete skill model.

If we look at how many steps are at least required to make sufficient predictions (see

level	0	1	2
1	+1.145	–	–
2	not sign.	-0.968	–
3	not sign.	-1.207	-0.184

(a) AUC PFA

level	0	1	2
1	-0.725	–	–
2	-1.423	-1.133	–
3	-1.526	-1.241	-0.167

(b) AUC AFM

level	0	1	2
1	not sign.	–	–
2	+1.303	+1.468	–
3	+1.464	+1.716	+0.248

(c) RMSE PFA

level	0	1	2
1	+0.447	–	–
2	+0.727	+0.554	–
3	+0.738	+0.563	+0.078

(d) RMSE AFM

Table 6.7.: Effect sizes for parameter *level*

Table 6.8), we can see clearly that models with *minimum steps* = *0* perform best. However the effect on the overall performance is only small to moderate.

Looking at the results of the evaluation of different definitions of a step (see Table 6.9), we also get a clear result that considering each submission of a student as a single step leads to best performance results. In contrast to *minimum steps*, the effect on the performance is large. When comparing the step definitions *last* and *first*, we see that *first* performs better regarding AUC while *last* performs better regarding RMSE.

To make PFA and AFM models comparable to PPM models, we use only a subset of meta-parametrizations of PFA and AFM models where *minimum steps* is *0* and *step definition* is *every*. Table 6.10 shows the mean values of AUC and RMSE for the different meta-parameters of the skill models. AFMs perform almost in all cases best while PPMs perform worst. The table gives a first impression on which meta-parameter values may have a positive impact on the prediction performance (best value for each meta-parameter bold).

Table 6.11 shows the results of the Friedman test and Kendall's W for all meta-parameters of the skill models. In PFAs the meta-parameter *Q matrix* has a large effect on AUC as well as RMSE. The meta-parameter *incorrect KCs* has no effect on RMSE and only a small effect on AUC. In AFMs however, it has a large effect on AUC, while all other meta-parameters have a small or medium effect. In PPMs the meta-parameter *incorrect KCs* has a large effect on RMSE and the meta-parameter *Q matrix* has a large effect on AUC while *KC level* has no significant effect on AUC and *KC count* has no significant effect on RMSE.

minimum steps	0	5	10	15
5	-0.291	–	–	–
10	-0.208	not sign.	–	–
15	-0.315	-0.108	-0.202	–
20	-0.650	-0.461	-0.663	-0.509

(a) AUC PFA

minimum steps	0	5	10	15
5	not sign.	–	–	–
10	-0.211	not sign.	–	–
15	-0.347	-0.262	-0.231	–
20	-0.794	-0.698	-0.729	-0.696

(b) AUC AFM

minimum steps	0	5	10	15
5	not sign.	–	–	–
10	not sign.	not sign.	–	–
15	+0.248	not sign.	+0.281	–
20	+0.638	+0.548	+0.726	+0.620

(c) RMSE PFA

minimum steps	0	5	10	15
5	+0.296	–	–	–
10	+0.509	+0.332	–	–
15	+0.664	+0.522	+0.414	–
20	+0.802	+0.704	+0.586	+0.286

(d) RMSE AFM

Table 6.8.: Effect sizes for parameter *minimum steps*

step def.	first	last
last	+0.413	–
every	+1.168	+1.071

(a) AUC PFA

step def.	first	last
last	+0.301	–
every	+1.150	+1.684

(b) AUC AFM

step def.	first	last
last	+0.291	–
every	-1.485	-1.890

(c) RMSE PFA

step def.	first	last
last	+1.691	–
every	-0.964	-2.413

(d) AUC AFM

Table 6.9.: Effect sizes for parameter *step definition*

		AUC			RMSE		
parameter	value	PFA	AFM	PPM	PFA	AFM	PPM
KC level	0	.596	**.697**	.585	.452	**.432**	.560
	1	**.657**	.692	**.589**	**.443**	.438	**.496**
	2	.624	.622	.582	.506	.541	.505
	3	.644	.619	.578	.484	.545	.508
incorrect KCs	all	**.634**	.647	**.592**	.473	.492	**.484**
	diff	.627	**.668**	.575	**.470**	**.485**	.550
KC count	binary	.613	.626	.578	.490	.530	.530
	multiple	**.648**	**.689**	**.589**	**.453**	**.448**	**.504**
Q matrix	all	.569	.651	.555	.480	.486	.525
	shared	**.659**	.660	**.611**	**.460**	**.477**	.522
	union	.642	.653	.564	.467	.480	.519
	common	.651	.656	.573	.464	.479	.513
	used	.618	.627	.585	.480	.536	.522
	set	.641	**686**	.597	.475	4.83	**.509**
	KAM	.632	.670	.598	.475	.481	**.509**

Table 6.10.: Mean values of AUC and RMSE for all meta-parameter values

Which meta-parameter has the largest effect on prediction performance highly depends on the concrete skill model. Even for similar models the same meta-parameter can be of completely different importance.

Looking at models for different levels of KCs (see Table 6.12), we see that level 1 PFAs models perform better than level 0 PFA models ($d = +1.859$), at least as good as the other levels for AUC, and even better than the other levels for RMSE. For AFM models there is no significant difference between level 0 and level 1 models which perform much better than models on level 2 or 3 (large effect). The meta-parameter *KC level* has no effect on AUC for PPM and there are no significant differences between the KC levels for RMSE except that level 2 is much better than level 3 ($d = +1.708$). Thus, we recommend to use models based on level 1 KCs since they perform best regarding all metrics and all skill models.

When we compare the two simple approaches for selecting which KC shall be taken as applied unsuccessfully (see Table 6.13), the Wilcoxon signed rank test shows us that models which assume all KCs from an incorrectly solved exercise as unsuccessfully applied performed better than those which were determined by the difference between the KC set of the student's solution and the KC set of the most similar correct solution for PFAs and PPMs, but the effect is only between negligible and medium. For AFMs selecting *diff* for the meta-parameter *incorrect KCs* however has a large effect on AUC ($d = 1.117$). Thus, the selection of the “best" value for the *incorrect KCs* meta-parameter depends on the concrete skill model.

	Friedman test p-value		Kendall's W	
parameter	AUC	RMSE	AUC	RMSE
KC level	0.001e-1	3.51e-136	0.249	0.334
incorrect KCs	0.003	0.033	0.154	–
KC count	6.10e-5	0.003	0.287	0.154
Q matrix	9.92e-11	6.31e-9	**0.607**	**0.514**

(a) PFA

	Friedman test p-value		Kendall's W	
parameter	AUC	RMSE	AUC	RMSE
KC level	0.007e-1	2.69e-8	0.201	0.454
incorrect KCs	7.90e-10	5.53e-06	**0.675**	0.369
KC count	0.001	0.001	0.184	0.184
Q matrix	2.87e-11	0.009e-1	0.427	0.234

(b) AFM

	Friedman test p-value		Kendall's W	
parameter	AUC	RMSE	AUC	RMSE
KC level	0.015	0.007e-1	–	0.201
incorrect KCs	0.001	5.35e-13	0.184	**0.930**
KC count	6.10e-5	0.593	0.287	–
Q matrix	3.96e-12	3.06e-05	**0.679**	0.318

(c) PPM

Table 6.11.: P-values of Friedman test and Kendall's W for all meta-parameters

level	0	1	2
1	+1.859	–	–
2	not sign.	not sign.	–
3	+0.949	not sign.	not sign.

(a) AUC PFA

level	0	1	2
1	not sign.	–	–
2	-0.955	-1.004	–
3	-1.032	-1.070	not sign.

(b) AUC AFM

level	0	1	2
1	-1.784	–	–
2	not sign.	+0.987	–
3	+1.103	+1.443	not sign.

(c) RMSE PFA

level	0	1	2
1	not sign.	–	–
2	+1.068	+1.067	–
3	+1.131	+1.102	not sign.

(d) RMSE AFM

level	0	1	2
1	not sign.	–	–
2	not sign.	not sign.	–
3	not sign.	not sign.	+1.708

(e) RMSE PPM

Table 6.12.: Effect sizes for parameter *level*

incorrect KCs	all
diff	-0.270

(a) AUC PFA

incorrect KCs	all
diff	+1.117

(b) AUC AFM

arg1

incorrect KCs	all
diff	-0.568

(c) AUC PPM

incorrect KCs	all
diff	-0.198

(d) RMSE AFM

incorrect KCs	all
diff	+0.737

(e) RMSE PPM

Table 6.13.: Effect sizes for parameter *incorrect KCs*

KC count	binary
multiple	+0.828

(a) AUC PFA

KC count	binary
multiple	+0.876

(b) AUC AFM

KC count	binary
multiple	+0.555

(c) AUC PPM

KC count	binary
multiple	-0.697

(d) RMSE PFA

KC count	binary
multiple	-0.837

(e) RMSE AFM

Table 6.14.: Effect sizes for parameter *KC count*

For answering whether KCs should be counted once or multiple times in a solution, the Wilcoxon signed rank test states that counting the KCs multiple times has a medium to large effect on prediction performance throughout all skill models and for all metrics (see Table 6.14).

Looking at the different definitions for a Q matrix (see Table 6.15 and 6.16), models which use the actually used KCs for each exercise perform worst throughout all models. Also using just all KCs as required leads to poor performance. The Q matrix definition *shared* performs best or at least as good as any other Q matrix. The ranking of the remaining Q matrix definitions varies among the skill models. For AFM models setting the Q matrix based on the set similarity also leads to better performance results regarding AUC.

While the effect size of a particular meta-parameter can be completely different among different skill model types, the tendency of which parameter value leads to best results is often similar. With regard to our results the best selection of meta-parameters for a skill model in the programming domain would be to build skill models based on KCs on level 1 refinement, use the actual number of occurrences of the KC as count, and define the Q matrix as the intersection of KCs in all correct solutions. The selection of the meta-parameter incorrect KCs, however, depends on the concrete skill model.

Skill model comparison

Table 6.17 shows the AUC und RMSE of the selected models. Each model performs better than the Majority class model regarding AUC. Regarding RMSE, the PPM models are worse. The “best” models perform significantly worse than the “meta” models for PFA as well as AFM models with a difference in AUC up to 0.049 on the validation set. For PPM models the “best” model performs only slightly better than the “meta” model.

The models based on meta-parameterization that is commonly used in related work performs worst even when assuming the best KC level selection. The difference can be up to 0.058 in AUC for the best KC level selection and up to 0.191 for the worst KC level selec-

Q matrix	all	shared	union	common	set	KAM
shared	+1.888	–	–	–	–	–
union	not sign.	not sign.	–	–	–	–
common	not sign.	not sign.	not sign.	–	–	–
set	not sign.	not sign.	not sign.	not sign.	–	–
KAM	+1.459	-2.104	not sign.	not sign.	not sign.	–
used	not sign.	-2.670	not sign.	-1.501	-1.528	-1.063

(a) AUC PFA

Q matrix	all	shared	union	common	set	KAM
shared	not sign.	–	–	–	–	–
union	not sign.	not sign.	–	–	–	–
common	not sign.	not sign.	not sign.	–	–	–
set	+1.418	not sign.	+1.337	+1.333	–	–
KAM	not sign.	not sign.	not sign.	not sign.	not sign.	–
used	not sign.	not sign.	not sign.	not sign.	-0.931	not sign.

(b) AUC AFM

Q matrix	all	shared	union	common	set	KAM
shared	+1.372	–	–	–	–	–
union	not sign.	-1.260	–	–	–	–
common	not sign.	not sign.	not sign.	–	–	–
set	+1.330	not sign.	+1.190	not sign.	–	–
KAM	+1.625	not sign.	+1.530	+1.278	not sign.	–
used	not sign.	-1.448	not sign.	not sign.	not sign.	-1.809

(c) AUC PPM

Table 6.15.: Effect sizes for parameter *Q matrix* and metric AUC

Q matrix	all	shared	union	common	set	KAM
shared	-0.887	–	–	–	–	–
union	not sign.	not sign.	–	–	–	–
common	not sign.	not sign.	not sign.	–	–	–
set	not sign.	not sign.	not sign	not sign.	–	–
KAM	not sign.	+0.809	not sign.	not sign.	not sign.	–
used	not sign.	+0.924	not sign.	not sign.	not sign.	not sign.

(a) RMSE PFA

Q matrix	all	shared	union	common	set	KAM
shared	-0.395	–	–	–	–	–
union	-0.309	+0.101	–	–	–	–
common	-0.275	+0.352	not sign.	–	–	–
set	not sign.	+0.163	not sign.	not sign.	–	–
KAM	not sign.	+0.130	not sign.	not sign.	not sign.	–
used	not sign.	+0.723	+0.667	+0.689	+0.609	not sign.

(b) RMSE AFM

Q matrix	all	shared	union	common	set	KAM
shared	not sign.	–	–	–	–	–
union	not sign.	not sign.	–	–	–	–
common	not sign.	not sign.	not sign.	–	–	–
set	not sign.	not sign.	not sign.	not sign.	–	–
KAM	not sign.	not sign.	not sign.	not sign.	not sign.	–
used	not sign.	not sign.	not sign.	not sign.	+2.286	+3.910

(c) RMSE PPM

Table 6.16.: Effect sizes for parameter *Q matrix* and metric RMSE

tion. For RMSE the difference can be up to 0.12 for the best KC level selection and up to 0.298 for the worst KC level selection.

Also for the DBN models, those which are fitted to the "common" meta-parameterization perform worse. DBN models which include variable roles as KCs perform better than those without variable roles for all KC levels. With an AUC of 0.759 and an RMSE of 0.389 the DBN model with KC level 1 and variable roles included performs similarly well as the best AFM model.

Models which are fit using the meta-parameters which were identified as best individual meta-parameter values perform better on the validation set than models which performed best on the test set or used common meta-parameterization from related work. Including variable roles increases the prediction performance of DBNs.

6.3. Case Study 2: Programming Errors

The second case study is about programming errors students make. By manual analysis, we identify common error types and determine the error landscape (see Section 5.2). This case study is an extension of our previous published work [137] and contains parts of it. The results are updated and extended where necessary, but the key findings remain the same.

6.3.1. Setup

Figure 6.9 depicts the setup of our second case study. Using the SmartAPE data (see Section 6.1), we perform a manual error analysis of students' code as described in Section 5.1. As a result, we get a list of error types which are assigned to one or more error categories and a data set which contains for each student's submission a list of errors which it contains. Then we determine the state of each error at each time step. Based on the resulting data set, we calculate the error frequency, duration, severity, and re-occurrence as defined in Section 5.2. Furthermore, we are interested in the distribution of error categories, especially the proportion of *sloppiness* in students' errors. We aggregate the data to obtain the student-based, type-based, and category-based error landscape.

The error data contains 4705 manually labeled errors from 271 students, classified into 105 error types.

6.3.2. Results

Student-based error landscape

Figure 6.10 shows a box-plot of the student-based error landscape. The median student-based error frequency is 16 (mean 18.1), i.e., a student makes about 16 errors during the course. However, the calculation of the median does not consider students who did not

Skill model	AUC	RMSE	level	KC count	Q matrix	incorrect KCs
Majority class	0.5	0.457				
PFA best	0.686	0.437	2	multiple	common	diff
PFA meta	**0.711**	**0.432**	1	multiple	shared	all
PFA common	0.554(2) - 0.656(1)	0.444 (1) - 0.566 (2)		binary	used	all
AFM best	0.703	0.430	2	multiple	set	diff
AFM meta	**0.752**	**0.421**	1	multiple	shared	diff
AFM common	0.561(3) - 0.694(0)	0.431(0) - 0.719(2)		binary	used	all
PPM best	**0.638**	**0.458**	1	binary	shared	all
PPM meta	0.630	0.463	1	multiple	shared	all
PPM common	0.589(0) - 0.602(3)	0.473(3) - 0.502(0)		binary	used	all
DBN meta	0.603(0) - 0.692(1)	0.432(1) - 0.461(0)		multiple	shared	diff
DBN meta + roles	0.624(3) - **0.759(1)**	**0.389(1)** - 0.426(2)		multiple	shared	diff
DBN common	0.541(3) - 0.628(0)	0.443(0) - 0.547(3)		binary	used	all
DBN + roles common	0.557(3) - 0.652(0)	0.430(0) - 0.548(2)		binary	used	all

Table 6.17.: Comparison of skill models

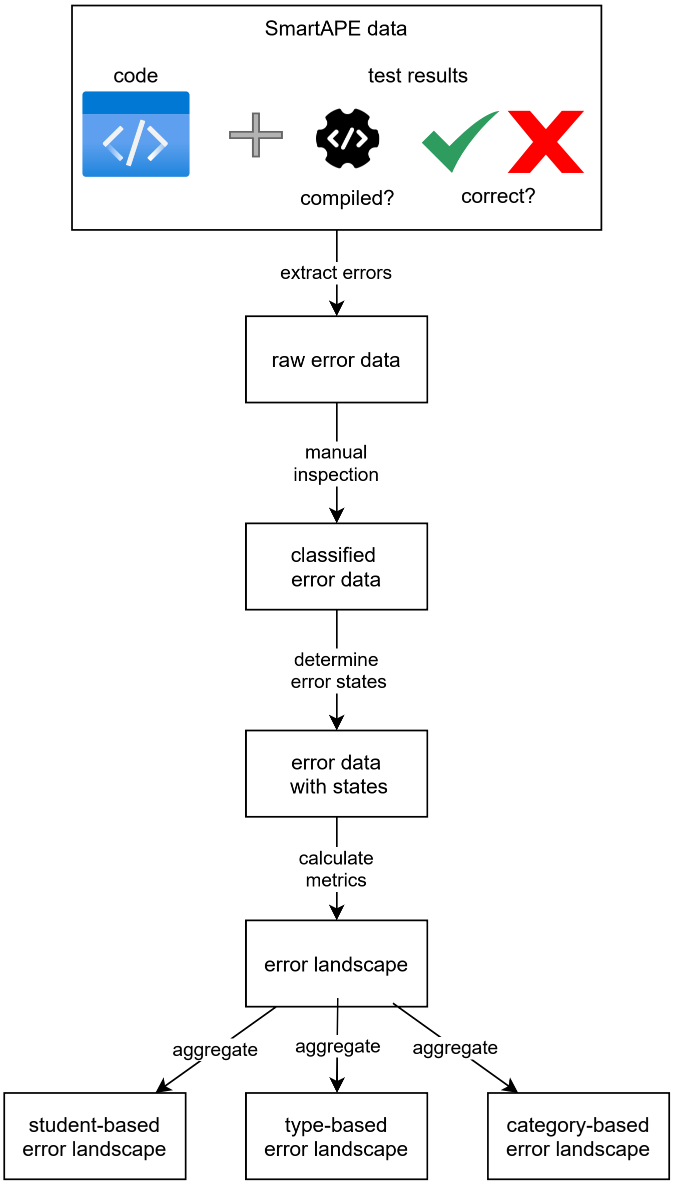


Figure 6.9.: Overview of the setup for case study 2

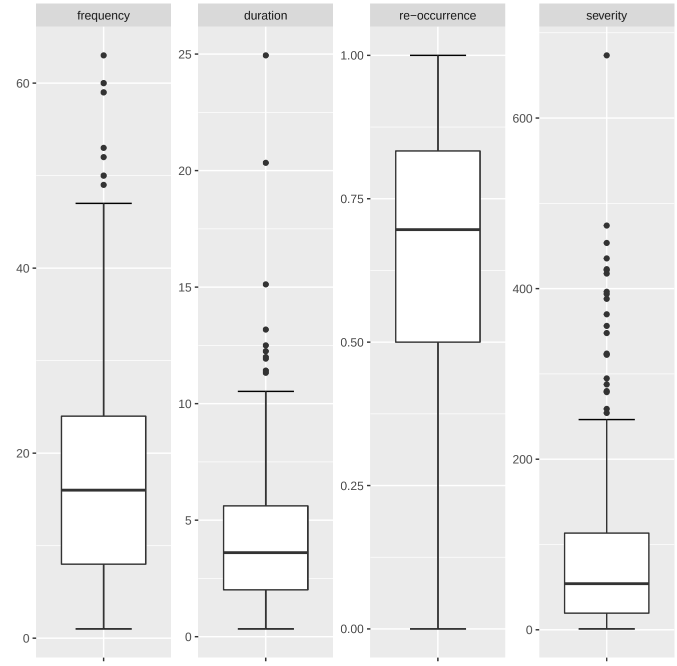


Figure 6.10.: Distribution of error frequency, duration, re-occurrence, and severity among students

do any mistake (11 students in total). The variance is large and the maximum frequency of a student is 63. The median duration is about 3.6 (mean 4.3), i.e., a student needs on average 3 to 4 submission till he/she fixed an error. The maximum duration is about 25. The students who have a large frequency are not identical with those who have a long duration. The median re-occurrence is about 0.7, so students tend to do the same errors again. Since severity is the product of frequency and duration, it is not surprising that the students with high severity are those who have a high frequency or a high duration.

The frequency varies a lot between the students while the duration is more stable. Because of the high re-occurrence rates, it is likely that students will do the same errors again. Since severity depends on frequency, it also varies a lot. Students with a high frequency do not necessarily also have a high duration.

Type-based error landscape

Table 6.18 shows the 25 top error types regarding severity. The complete type-based results can be found in Appendix E.3. A description of the top 50 error types with examples can be found in the supplementary material [162] to our publication.

Many of the errors are related to input and output. The error occurred most often is *wrong output format*. Since our exercises are mostly assessed by regular expressions, it is crucial to adhere to the prescribed output template. Students often do not read the exercise description thoroughly, introduce typos, or just ignore the template which leads to failed tests.

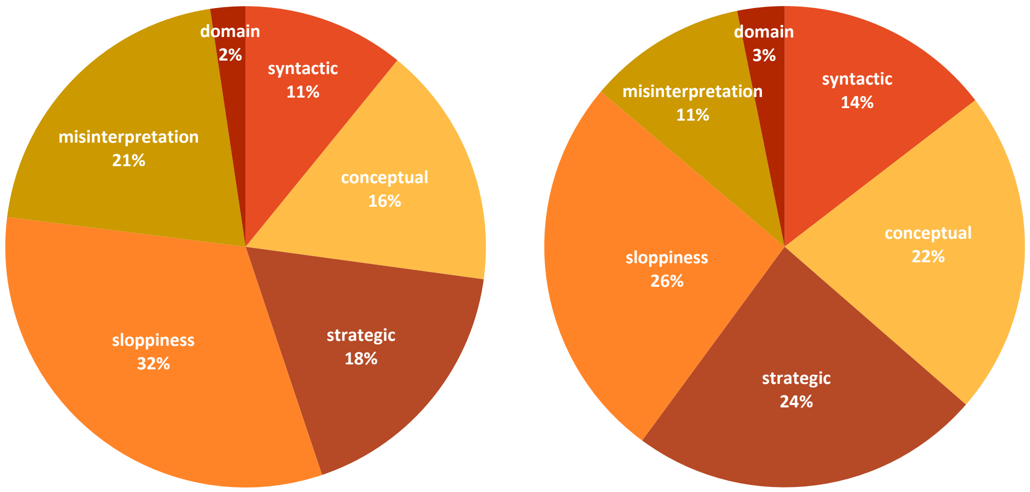
Students often also do not check for boundary conditions, e.g., whether the input is valid (error e3), i.e., of the correct type, or if array size is exceeded during input (error e12), and omit boundary cases (error e11). An error that also occurred quite often is that students do not understand how `getchar()` and `scanf()` consume data in the input buffer (error e24). That results in certain variations of nesting these constructs. A typical variation is using `getchar() != EOF` as a looping condition to check for a user's end of input and calling `scanf` inside the loop to read-in the data. Typical strategic errors are also that students often miss some additional conditions when branching or looping (error e40) or do not loop when it is required (error e23). Sometimes they even omit to implement a complete subgoal of the exercise (error e20). In general, half of the top 10 errors can be avoided as they are often a result of sloppiness .

The error with the highest duration refers to the student's believe that a condition is checked continuously in a loop and the loop is left immediately after the change of the guard variable (e55). Furthermore, errors referring to pointers like *missing malloc* (e53), *wrong allocation size* (e68), and *using address operator wrong* (e60) belong to the top 10 duration error types. However, these errors are rare (frequency of 1 or 2), such that their severity is not that high. More severe are errors like *using = instead of ==* (e37) with a mean duration of 9.83 or initializing a variable with an incorrect value (e35) with a duration of 7.83. Errors having a relatively low duration and a high frequency are, e.g., *missing escaping in String* (e46, $f_t = 19$, $d_t = 3.14$), *missing/wrong include* (e15, $f_t = 100$, $d_t = 3.67$), or *missing terminating character* (e13, $f_t = 109$, $d_t = 3.66$).

The error types *missing terminating character* (e13) and *de-referencing something that is not a pointer* (e30) have a re-occurrence rate of 1, i.e., every students that makes one of these makes, does that mistake more than once. In general, it is likely that an error will re-occur. There are only few errors with a re-occurrence rate smaller than 0.5 and those are often not so frequent.

	error type	s_t	f_t	d_t	r_t	syntactic	concecptul	strategic	sloppiness	misinterpretation	domain
e1	wrong output format	4622.64	1193	3.87	0.92				X	X	
e2	missing semicolon	1416.09	319	4.44	0.81	X			X		
e3	missing check for invalid input	1266.24	235	5.39	0.73			X			
e4	undeclared variable	1142.37	182	6.28	0.76	X			X		
e5	confuse EOF and '\n'	1078.74	189	5.71	0.53		X			X	
e6	wrong array size	978.49	172	5.69	0.52			X		X	X
e7	off-by-one-error	856.08	173	4.95	0.67		X	X	X		
e8	wrong boundaries	843.39	182	4.63	0.6			X		X	
e9	unexpected output	809.88	185	4.38	0.6				X		
e10	wrong escaping	705.9	144	4.9	0.42	X					
e11	boundary case omitted	485.57	77	6.31	0.57			X			
e12	no check for array limits during input	481.57	74	6.51	0.63			X			
e13	missing terminating character	398.85	109	3.66	1				X		
e14	missing error output	392.49	89	4.41	0.62				X		
e15	missing/wrong include	366.84	100	3.67	0.9		X		X		
e16	uninitialized variable	348.25	64	5.44	0.57		X		X		
e17	order of conditions	345.6	52	6.65	0.69		X				
e18	wrong claculation	338.81	65	5.21	0.52			X			X
e19	wrong type	331.02	55	6.02	0.44		X				
e20	missing subgoal	328.19	63	5.21	0.49			X			
e21	unnecessary if	322.68	63	5.12	0.51			X		X	
e22	missing/wrong pointer de-reference	283.6	65	4.36	0.79		X				
e23	missing loop	282.12	49	5.76	0.48			X		X	
e24	misconception of input buffer	280.29	53	5.29	0.59		X	X			
e25	conflicting/incompatible types	249.33	66	3.78	0.67		X				

Table 6.18.: Top 25 error types ordered by severity



(a) incl. error e1

(b) excl. error e1

Figure 6.11.: Distribution of error categories

The most common errors relate to input and output or missing boundary checks. Students often need longer time to fix the error *using = instead of ==*. A missing terminating character is done by all students who did this error more than once.

Category-based error landscape

Figure 6.11a shows the resulting distribution of categories regarding their frequencies. About half of all errors are caused by sloppiness (32%) or misinterpretation (21%). This high percentage is mainly a result of error e1 because this error covers about 25% of all resulting errors. As this error is mainly a result of our testing method of students' solutions, we removed it from evaluation to get a more unbiased impression. The resulting distribution is depicted in Figure 6.11b. The value of *misinterpretation* gets much lower (11%) and also the portion of sloppiness decreases (26%). But still, sloppiness is the most often occurring category. Among the categories related to programming knowledge, strategic errors are the most common (24%) and syntactic errors are the least common made errors (14%). The portion of errors due to a lack of domain knowledge is fortunately low (3%) which means that students can focus more on the programming task than struggling with domain problems. There are no significant differences in duration or re-occurrence rate between the categories, even when not considering e1. We also find out that 16.1% of students' incorrect solutions are solely incorrect because of sloppiness.

The most frequent error categories are sloppiness and strategic errors. The duration and re-occurrence is similar throughout all categories.

6.4. Case Study 3: Development of Programming Knowledge Over Time

This case study targets RQ 3. We focus on the evolution of KC knowledge and errors and look for different patterns among students.

6.4.1. Setup

We use the DBN which we trained in case study 1 (Section 6.2) to create learning curves as defined in Section 4.4. We manually analyze the learning curves for each KC and group similar curves. The slope of a curve indicates the difficulty of the KC. In traditional BKT, a knowledge level of 0.95 is considered as mastery of a skill [52]. So if the final knowledge level of a KC is below this threshold, the student needs more practice.

We use the error landscape from case study 2 (see Section 6.3) to derive the temporal evolution of the frequency and duration for the different error categories. We do not use the re-occurrence rate, since it is calculated over the complete course and not on a per-exercise basis, and severity, since it is already encoded in the frequency and duration. We use k-means clustering as defined in Section 5.3 to group similar error patterns of students. To determine the number of clusters, we use the average silhouette approach [163]. Afterwards, we evaluate the relationship between the error patterns, i.e., clusters, and the final exam outcome.

6.4.2. Results

KC Learning Curves

We have identified 8 different types of learning curves. Figure 6.12 shows a representative learning curve for each type. Table 6.19 summarizes which KC has which type of learning curve. The learning curves for all KCs can be found in Appendix E.4.

The first learning curve type represents KCs which start at a knowledge level of 0.75, reach a level of about 0.9 after 5 exercises and about 0.95 at the end of the course. KCs with such a curve are basic KCs like, e.g., *block*, *declaration*, or *expression*, which occur in almost all exercises and are, thus, most often practiced. The second learning curve type is quite similar to the first one, but shifted about 0.15-0.2 of knowledge downwards, starting at about 0.55-0.6 and ending at about 0.8. Type 3 learning curves are also similar to type 1 curves, starting at a knowledge level of about 0.65-0.75 and ending at 0.8-0.9. But while the knowledge in type 1 curve increases exponentially till exercise 5 and slows down afterwards, the knowledge level increases linearly in type 3 curves throughout the complete course. Type 4 corresponds to a linear version of type 2. Type 5 is similar to type

Learning curve type	KCs
1	block, comparison, data type, declaration, declarator, expression, jump, primary expression, statement
2	assignment, function call, increment/decrement, iteration, member access
3	include, initialization, preprocessor, selection
4	arithmetic expression
5	logical expression
6	define, expression list, label, type cast
7	sizeof-operator, type qualifier
8	bitwise operator, conditional operator, storage class

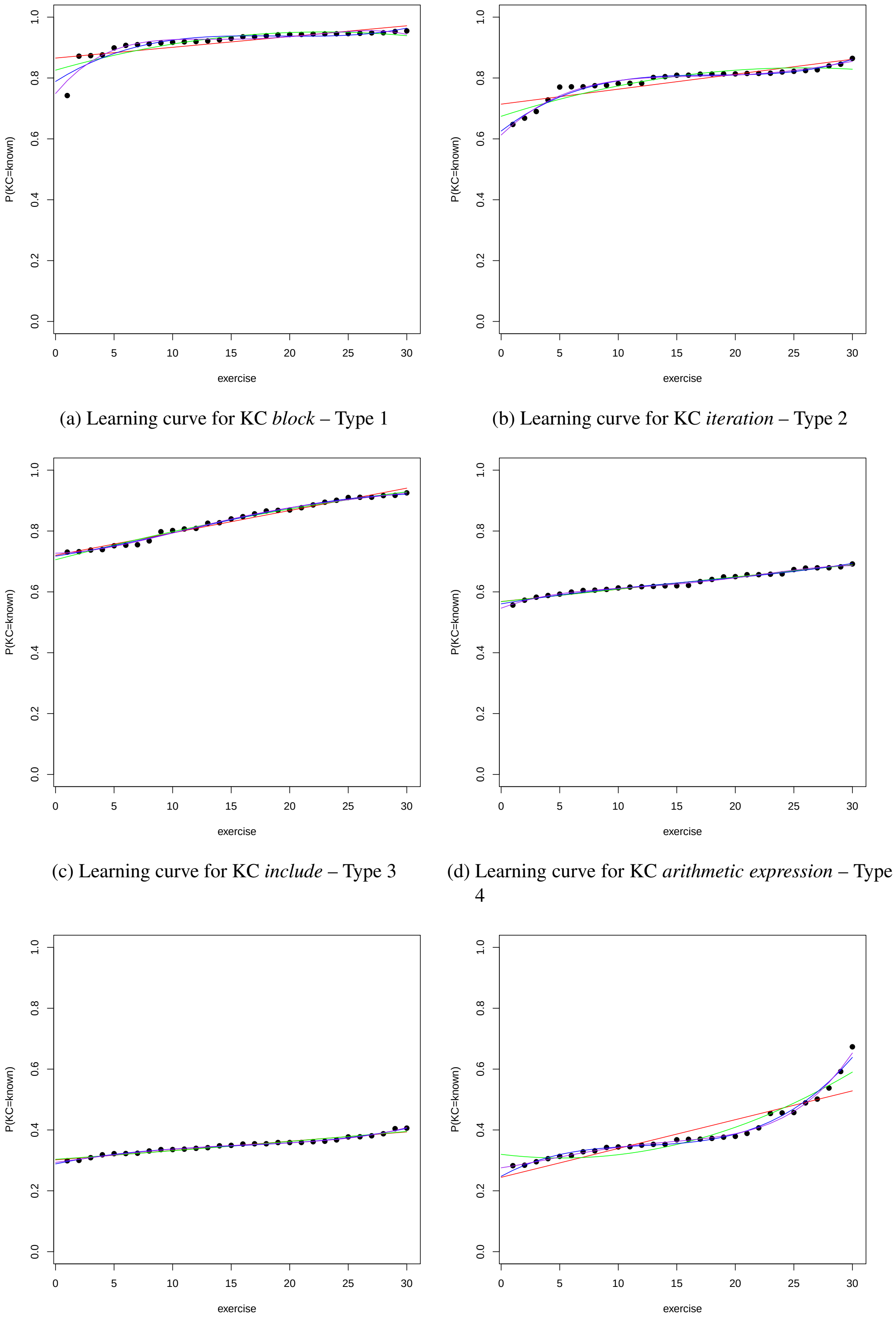
Table 6.19.: Assignment of KCs to learning curve types

4, but on a lower knowledge level, starting at about 0.3 and ending at about 0.4. KCs with a learning curve of of type 6 start on a low level of knowledge of about 0.2 and are learned slowly such that they achieve only a level of about 0.6 at the end. In contrast type 7 KCs achieve a final knowledge level of about 0.8 although also starting at a low level of about 0.2. Type 8 learning curve KCs are completely unknown at the beginning and achieve a final knowledge level of about 0.4.

The learning curves for variable roles all look quite similar, starting at knowledge level of 0 and increasing up to 0.6. The major difference between them is the "tail" at the start of the course since some roles are only necessary for later exercises, and thus, their knowledge level starts growing only later in the course (see Figure 6.13a). Only the roles *stepper* and *fixed value* achieve a knowledge level above 0.8 at the end (see Figure 6.13b). The learning curves for all variable roles can be found in Appendix E.4.

Despite syntactic KCs and variable roles we have a further hidden node in our DBN representing the student's problem solving ability. Figure 6.14 shows the learning curve of the problem solving ability. It starts at a low knowledge level of about 0.4 and increases linearly but slowly until reaching a level of about 0.6 at the end.

Only KCs with learning curves of type 1 and the variable role *stepper* achieve the mastery level of >0.95. All other KCs and roles and the problem solving ability require more practice. KCs of learning curve type 2 and 3 have a high initial knowledge which indicates that students basically understand the concepts even before practice. KCs with learning curve type 7 and variable roles are easy to learn. The remaining KCs are hard to learn.



(a) Learning curve for KC *block* – Type 1

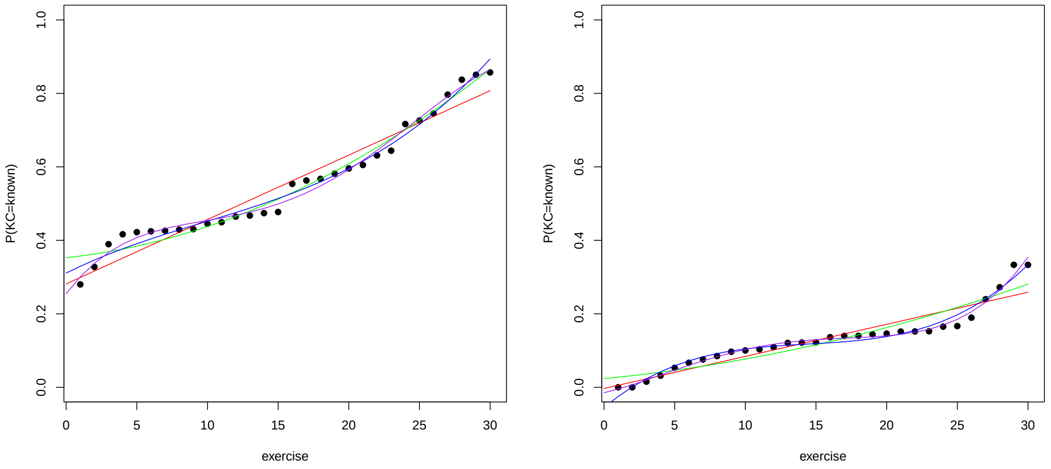
(b) Learning curve for KC *iteration* – Type 2

(c) Learning curve for KC *include* – Type 3

(d) Learning curve for KC *arithmetic expression* – Type 4

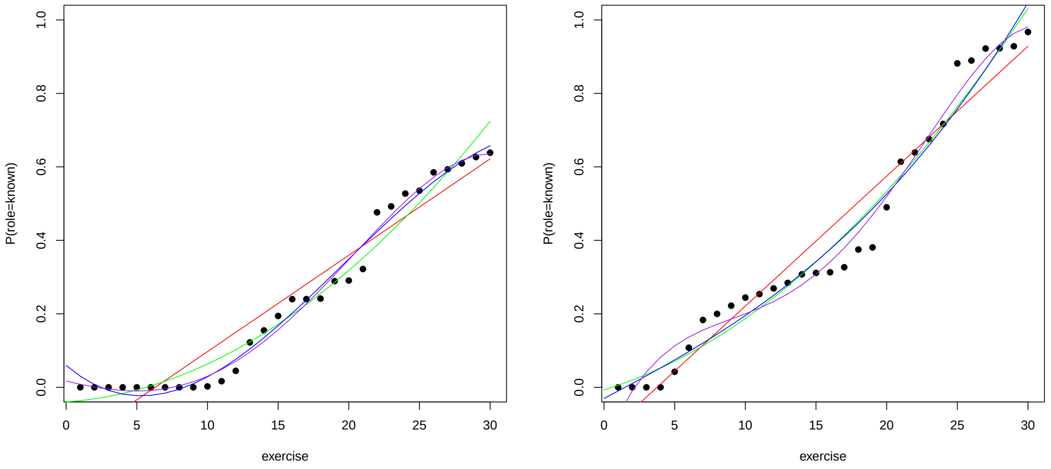
(e) Learning curve for KC *logical expression* – Type 5

(f) Learning curve for KC *type cast* – Type 6



(g) Learning curve for KC *sizeof-operator* – Type 7 (h) Learning curve for KC *bitwise operator* – Type 8

Figure 6.12.: Types of learning curves for KCs



(a) Learning curve for variable role *most-wanted holder* (b) Learning curve for variable role *stepper*

Figure 6.13.: Learning curves for variable roles

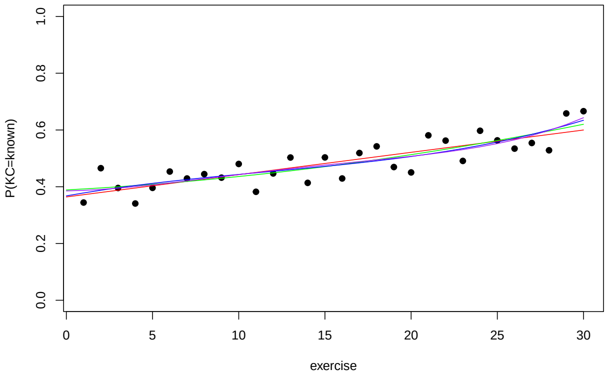


Figure 6.14.: Learning curve for the *problem solving ability*

Error curves

Figure 6.15 shows how the frequency of errors from different error categories changes over time. The frequency of sloppiness and misinterpretation errors starts quite high with about 0.58 resp. 0.47 errors per student on average in exercise 1 and decreases continuously throughout the course. Errors of categories conceptual, syntactic, and strategic start at a quite low level of about 0.1-0.15 errors per student. While syntactic error occurrence increases only for the first three exercises before it decreases continuously, the frequency of conceptual and strategic errors increases till exercises 5 to 7 until it decreases. Domain errors are always below a frequency of 0.06 throughout the course. At about exercise 20 all categories fall below an average frequency of 0.04.

For the duration of error categories (see Figure 6.16), no clear trend can be discerned. The duration slightly increases throughout the course for all categories. However, for the single exercises the duration is quite jumpy, especially for domain errors, such that we can assume that the actual error duration depends on the concrete exercise rather than on the error category.

Sloppiness and misinterpretation errors are descreasing continuously. In the beginning, the number of conceptual and strategic errors increases until it starts decreasing after about 8 exercises.

Error patterns

The average silhouette approach proposes us a number of $k = 3$ clusters. Cluster 1 contains 72 students with about 11-29 (22.18 on average) missing exercises, the second cluster contains 27 students who have only 0-2 missing (0.11 on average) exercises, and the third cluster contains 182 students with 0-13 (1.34 on average) missing exercises. In general, we can say that cluster 2 is the cluster with students making the most errors and taking the most

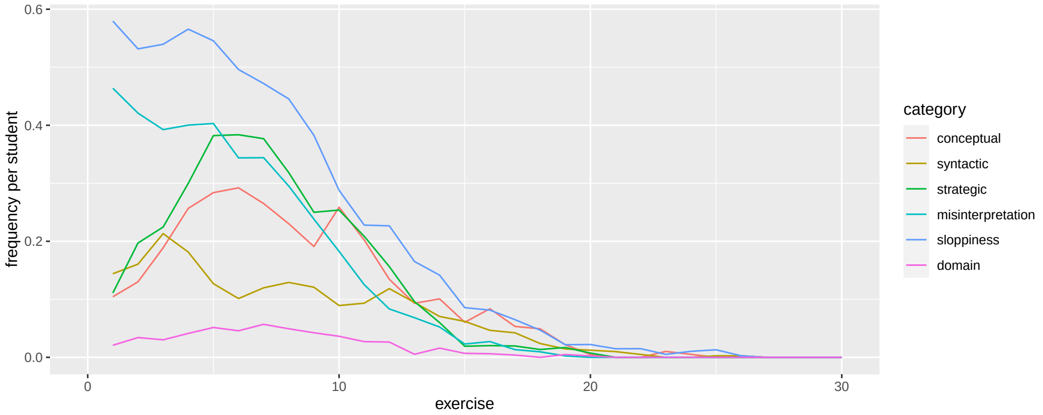


Figure 6.15.: Evolution of the average frequency of errors from different categories

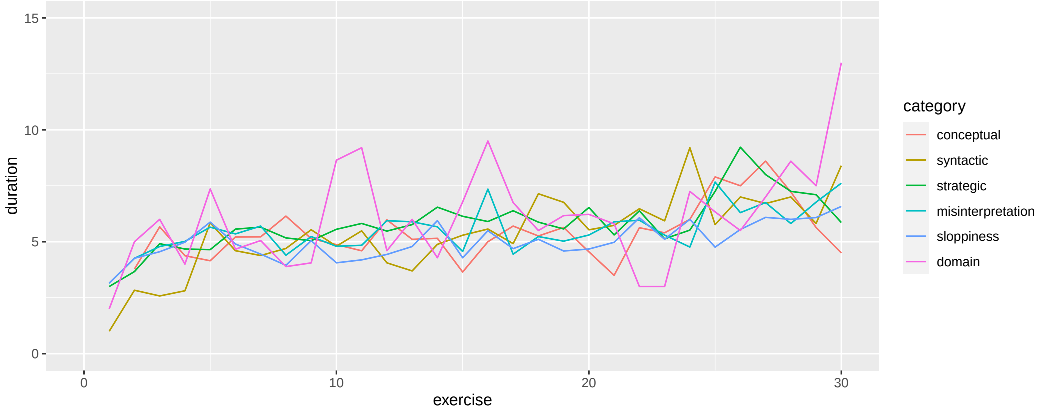
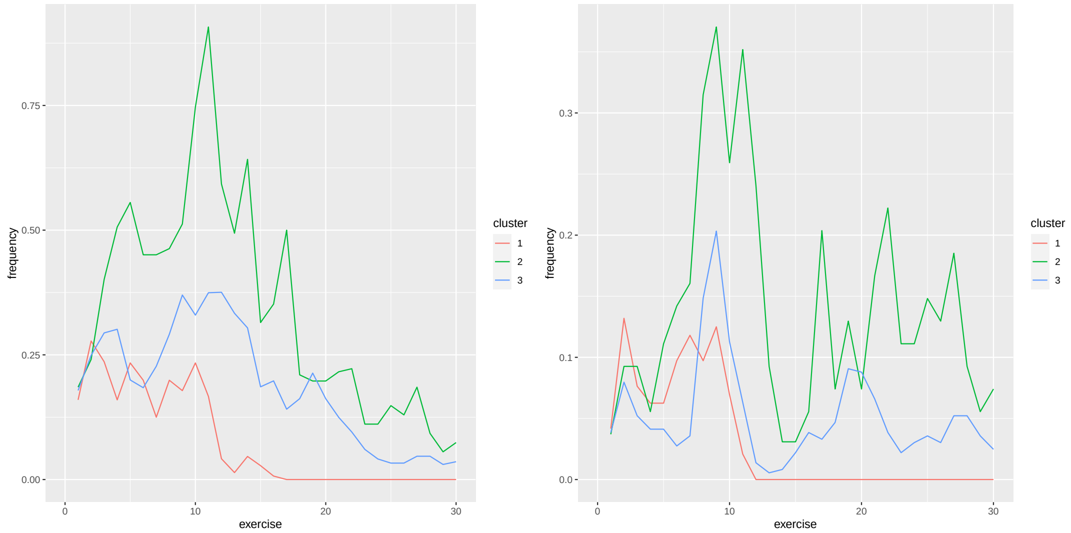


Figure 6.16.: Evolution of the duration of errors from different categories



(a) Frequency for category *sloppiness*

(b) Frequency for category *syntactic*

Figure 6.17.: Error patterns of the different clusters

time for fixing them throughout all error categories and cluster 1 describes students who drop out early in the course. All error curves for the clusters can be found in Appendix E.5.

Cluster 1 and cluster 3 students behave quite similar in the beginning for about 8 to 10 exercises for errors in the categories *conceptual*, *strategic*, *misinterpretation*, and *domain* with regard to frequency as well as duration. Cluster 3 students do more sloppiness errors than cluster 1 students (see Figure 6.17a). In contrast, with regard to syntactic errors the error frequency of cluster 2 students is higher than the frequency of cluster 3 students until cluster 2 students do not submit solutions anymore (see Figure 6.17). Cluster 1 students do a lot of syntactic errors even at the end of the course. The same holds also for the duration of syntactic errors. The frequency and duration of the categories *strategic, misinterpretation, and domain* gets 0 for all clusters at exercise 24.

Table 6.20 shows the results of the final exam for each cluster. There is no significant difference between the pass rates for the particular clusters. However, more students from cluster 1 are not participating at the exam. Students from cluster 2 on the other hand have a higher fail rate in the exam.

	cluster 1	cluster 2	cluster 3
pass	73.6%	74.0%	77.4 %
fail	18.1 %	26.0%	21.4 %
miss	8.3 %	0.0%	1.2 %

Table 6.20.: Exam results by clusters

The error curves of the students can be clustered into three groups. One group is representing students which only work on maximum up to 19 exercises. These students are more likely to quit the course in advance. The second group refers to students who are hardworking but produce a lot of errors. Thus, their risk of failing the exam is higher than for the other groups. The third group is a mixture of both groups. Students are doing many of the exercise but their error frequency is lower than the one of group 2.

7. Discussion

In this section, we discuss the findings of our case studies and provide answers to our research questions. Moreover, we report the strengths and limitations of our approach and show potential threats to validity of our studies. In addition, we use the results to derive implications for teaching.

7.1. Answers to Research Questions

We provide the answers to our research questions from Section 1.1 by summarizing the results of our case studies and putting them into context.

7.1.1. RQ1: How can we construct a skill model for the programming domain?

Since skill models are the core component of each adaptive learning system and required for gaining an estimate of a student's knowledge, our goal is to construct a skill model which fits the specific needs of the programming domain. One major property of programming is that there may exist an unlimited number of potential solutions. So we first need to get an impression how different students' solutions actually are, which leads to RQ 1.1. The results from our similarity analysis in Section 6.2.3 show that there are actually differences between student's solutions, but these differences are relatively small. The mean similarity is, depending on the similarity metric, located between 0.689 to 0.798 for level 3 KCs. So we need to consider this fact when defining a skill model for the programming domain.

Answer to RQ 1.1 – *How great is the difference between students' code?*: While the diversity of students' solutions is quite high, the actual similarity between the similarities is relatively low.

When constructing a skill model, we have several options to account for the properties of the programming domain by defining *meta-parameters*. We combined multiple configurations of meta-parameters and evaluated them by statistical tests. The results show that the correct meta-parameterization can drastically increase the performance of skill models for the prediction of student's performance on future exercises. Less surprising is that training a skill model on data from each submission being a single step leads to best prediction results since we want to predict every solution of a student. When the goal is predicting a student's first solution to an exercise probably looking at student's first submissions would be more beneficial. The selection of a minimum step size to avoid potential noise in the beginning does not play a crucial role in skill modeling. Our results show that the optimum minimum step size is *0*, i.e., the data right from the beginning is good enough to be used for prediction.

The definition of a Q matrix can have a moderate to large effect on the prediction performance. In our set-up it was not beneficial to use those KCs as required KCs which were actually used by the student in his/her solution. We think this is due the fact that when a student's solution is incorrect, there are incorrect or missing parts and, thus the KCs taken for knowledge estimation are incorrect. Best results are achieved when using the KCs which are used in every solution path. We think that these KCs are the core KCs for solving an exercise correctly. Together with the fact that we only have two very modest approaches for the determination of the incorrectly applied KCs, we believe that evaluating only the shared KCs leads to better prediction results. This is also in accordance with the results of Yudelson et al. [34] and Huang et al.[95] who showed that a reduced set of required KCs leads to better performance results than using all KCs that were actually used.

Also the choice of the KC level has an influence on the prediction performance. Our results indicate that using KCs on refinement level 1 leads to best results which means that a too fine-grained definition of KCs leads to too much noise. Although using concrete counts of used KCs rather than a binary counting results in better prediction results, it is surprising that the effect is only negligible to small. The most inconsistent meta-parameter is *incorrect KCs*. Its effect ranges from negligible to very large dependent on the concrete skill model and performance metric.

Answer to RQ 1.2 – *Which effect does meta-parameterization have on the prediction performance of skill models?*: Meta-parameterization can have large effect on the prediction performance. Therefore, the choice of the concrente meta-parameters should be carefully evaluated when creating a new skill model. When extensive evaluation is not possible, our results can be a good starting point, i.e., starting with KCs which are not too fine-grained, including each submission of a student as a step, and using the intersection of KCs of all correct solutions for the definition of the Q matrix.

In skill modeling, interpretable models are preferred as they contain additional information about students or course content. PFA and AFM are interpretable models as the parameters of the logistic regression model contain information on the difficulty of KCs or a student's individual learning rate. However, they do not consider dependencies between KCs. But for programming there clearly exist some, e.g., pointers and arrays are related in C, as well as the different forms of loops. Since logistic regression models assume independent variables, the estimated values of the parameters are pointless for interpretation even when the model's prediction capability is high. To integrate KC dependency into skill models, we use DBNs for skill modeling. We define a topology for the DBN consisting of three major parts which target specific properties of the programming domain: the domain model containing dependencies between KCs and allowing to use multiple KCs in one model; an item-KC-mapping structure that uses AND gates such that the DBN is able to deal with a dynamic Q matrix; and the explicit modeling of the problem solving ability which allows us also to monitor student's progress in that ability. In addition, by using virtual evidences, the model is able to deal with multiple application of the same KC in one step. As a result, the DBN is able to deal with all properties of the programming domain. The results of the skill model comparison (see Section 6.2.3) show that DBNs (AUC=.692, RMSE=.461) can be as good predictors as, e.g., PFA models (AUC=.711, RMSE=.432). When integrating variable roles into the DBN, they even achieve as good results (AUC=.759, RMSE=.389) as AFM models (AUC=.752, RMSE=.421). This indicates that additional semantic information like the one obtained by variable roles can lead to more sophisticated models.

Answer to RQ 1.3 – *How can we modify DBNs to reflect the properties of the programming domain?*: DBNs inherently provide a possibility to model dependencies between multiple KCs. We use a combination of an evidental node and a noisy AND gate to describe whether a KC is required in that step to incorporate the possibility of a dynamic Q matrix. We use virtual evidences for our observation nodes to integrate multiple applications of one KC in the same step. A comparison to other common skill models indicates that the model can keep up with them with regard to prediction performance while providing a better interpretability.

7.1.2. RQ2: Which errors do students make during programming?

To get a holistic view on students' programming errors, we do not rely on diagnostic messages of compilers because, first, they can not identify strategic errors, and, second, diagnostic message and the actual error type often do not fit together [88]. Instead, we perform a manual analysis of all incorrect solutions. Moreover, we are not only interested in how often a student makes errors, but also how long he/she needs to fix them, and how often they are repeated. For that purpose, we calculate the metrics error frequency, duration, and re-occurrence rate to which we refer as error landscape.

McCall and Kölling [138] also manually analyzed error frequency and duration, but they only analyzed non-compilable solutions. Errors which were frequent in their study like, e.g., *typos*, *undeclared variables*, *missing semicolons* also occurred quite often in our analysis. Also our third most often occurred error *missing check for invalid input* is comparable to their top 10 error *unhandled exception* since there are no exceptions in C.

The error with the highest frequency in our study is the *wrong output format* error which is mainly a result of our testing method. Students often do not read the description of the exercise thoroughly or introduce typos. Ettles et al. observed a similar problem. In their study, they found out that students have inserted `printf` into their solution although it was not requested in 83% of the solutions. This led the test cases fail although their solutions basically were correct. In addition, they identified three major misconceptions in students' programming code: *uninitialized variable*, *off-by-one error*, and *boundary error* which also occurred relatively frequently in our data.

We also identified several errors which correspond to some of the plan composition problems identified by Spohrer and Soloway [82]. Not explicitly derivable from our error types is the *previous-experience problem*. A very frequent result of this problem is the *confusing EOF and '\n'* error. While checking for `'\n'` in input is sufficient for some exercises, others explicitly require an `EOF`. Students who previously always used `'\n'` in their solutions believe that it is the common way to check for an end of input. Spohrer and Soloway's *interpretation problem* is so common that we have defined an own category for it. Also very frequently the *boundary problem* occurred. It describes that students are unable to find the correct boundaries during solution, e.g., the initialization value of counting variables in loops and when looping has to be ended. It is more than just an *off-by-one-error* since it does not result from the misconception of constructs, e.g., loops or arrays, or sloppiness but often is a lack in strategic knowledge. The last problem covered by our errors is the *unexpected cases problem*. This problem is mainly reflected by *missing check for invalid input* and *boundary case omitted*.

Since our most frequent error *wrong output format* is a result of our testing method and would skew the distribution of error categories, we excluded it to get an adjusted distribution. The results show that sloppiness and strategic errors are the most frequent categories of errors. Our results estimate that about 16.1 % of all solutions are incorrect solely because of sloppiness. Although sloppiness has such a large impact, to the best of our knowledge, none of the previous studies has considered sloppiness in their analysis of programming errors. We assume that many researchers and educators underestimate how often it actually occurs. Another reason may be that they do not perceive it as a "real" error since it is made unintentionally and there is not much that can be done against it from the educator's site. Our results let conclude that a lack in strategic knowledge is one of the main reasons why programming is perceived challenging by students. However, strategic knowledge is independent of a concrete programming language and, thus, students need to develop appropriate cognitive structures for problem solving. We suggest that students should be confronted with algorithmic problems on practical examples first, e.g., sorting books, to develop a strategy first before being confronted with difficulties arising from syntactic and conceptual knowledge.

Answer to RQ 2.1 – *What are the most common errors made by students?*: Most errors students make are due to sloppiness or a lack in strategic knowledge.

The results show that there is no significant difference between the duration of errors from different error categories. This is somehow surprising. We would have assumed that syntactic or sloppiness errors are easier to fix since the compiler usually points to the concrete location of a syntactic error and sloppiness errors are usually made unintentionally and could be identified with a more thorough look at the code. The long duration for syntactic errors could be a clue that students have problems with understanding diagnostic messages of compilers. Integrating the interpretation of diagnostic messages into curriculum may be therefore a potential option. On average, a student needs 3 to 4 submission to fix an error which seem to be a long time and more trial and error than directed debugging. An investigation of the debugging abilities of students could provide more insights. The errors *using = instead of ==* ($d_t = 11.2$) has a high duration and occurs quite often which indicates that it is actually hard to fix. The reason could be that the = is more natural form for defining equality and beginner students have problems distinguishing it from assignment. When you are not aware of the difference, fixing the error is difficult because you do not recognize it as an error in the code. On the other hand, the errors *missing escaping in String* ($d_t = 2.67$) and *missing terminating character* ($d_t = 3.79$) have a relatively low duration and are, thus, easy to fix. The reason is probably that the diagnostic messages and the error locations are clear.

Answer to RQ 2.2 – *Which errors are hard/easy to fix?*: There are no significant differences in the error duration between different error categories. The error type *using = instead of ==* is hard to fix. In contrast, the error types *missing escaping in String* and *missing terminating character* can be fixed easily .

The results from our second case study indicate that the duration is similar for all error categories. However, the errors *missing terminating character* and *de-referencing something that is not a pointer* both have a re-occurrence rate of 1, i.e., a student makes the error more than once. While the first error is a very common sloppiness error, the second error indicates that students need a longer time to actually understand the concept of a pointer and being able to distinguish between pointers and non-pointers.

Answer to RQ 2.3 – *Which errors re-occur often?*: There are no significant differences in the error re-occurence between different error categories. The error types *missing terminating character* and *de-referencing something that is not a pointer* always re-occur .

7.1.3. RQ3: How does the programming knowledge of students change over time?

The third research question deals with the temporal development of student's programming knowledge. We distinguish between knowledge and errors. RQ 3.1 targets the knowledge component and asks which KCs are hard/easy to learn. After showing that DBNs are an appropriate skill model when it comes to interpretability in case study 1, we use the knowledge estimates generated by the DBN to derive learning curves for each KC. The learning difficulty of a KC can be deduced from three parameters of the curve: the initial knowledge of the KC, the slope of the curve, and the final knowledge level. We have generated learning curves for syntactic KCs, variable roles, and the problem solving ability.

The results show that only few KCs reach the mastery level of 0.95. These KCs are the basic building blocks of almost each exercise like, e.g., *declaration*, *expression*, and *statement*, but also jump statements (because of `return`). Only the variable role *stepper* achieves mastery, because it is also required in almost all exercises when iterating through an array. In general, all variable roles have a small initial knowledge. This may be due to the fact that variable roles are a composition of multiple basic KCs which have to be learned step by step by students. However, the learning curves of variable roles are almost linear with a relatively large slope. This indicates that they can be learned easily when having enough practice opportunities. The learning curves of the *logical expression*, *bitwise-operator*, *conditional operator*, and *storage class* is relatively plain which means that students have difficulties with learning that KCs. Also the problem solving ability is difficult to learn. For teaching this means, that those KCs have to be targeted more, either by providing more exercises requiring the use of that KCs, or alternatively, since the error curve is quite plain, the KCs need to be discussed more in the lecture.

Answer to RQ 3.1 – *Which KCs are hard/easy to learn?*: Most KCs require more practice than we offer with our 30 exercises to achieve mastery. Only the basic KCs like expression, statement, and declaration as well as the variable role *stepper* achieve mastery. Variable roles can be learned easily, while the problem solving ability is hard to learn. Futhermore, the KCs *bitwise operator*, *logical expression*, and *conditional operator* are hard to learn.

RQ 3.2 targets the error component and asks how programming errors change over time. To evaluate the research question, we used the metrics frequency and duration aggregated over the different categories for each exercise and got an error curve describing the temporal development of the metrics. The results show that the frequency of sloppiness and misinterpretation errors continuously decrease. We think, the reason is that students get more familiar with how exercises are described and what is expected from them. The number of conceptual and strategic errors first increases for about 8 exercises until it also starts to decrease. The increase in the beginning may be due to increasing complexity of the exercises and, thus, higher requirements on the concepts used and strategic knowledge. After a short period of familiarization, the learning then starts. Syntactic errors also have an increase in the beginning but only for about 3 exercises. The number of domain errors is quite low throughout the complete exercise. After about 20 exercises, the frequency falls below 0.2 errors per student. The decrease of the error frequency and reaching a very low frequency in the end indicates that students learn from their mistakes and their programming knowledge increases over time.

Although the error duration varies between exercises, no real trend can be identified for any of the categories. The duration does not improve over time which is again an indicator for a low debugging ability or at least for no growth in the debugging ability.

Answer to RQ 3.2 – *How do errors change over time?*: The number of errors of all categories decreases over time to a low frequency of 0.2 errors per student. However, syntactic, conceptual, and strategic errors have some kind of familiarization phase in the beginning where the frequency increases until it starts decreasing. The error duration toggles around a constant value throughout the complete course.

So far, we only regarded how errors change for the complete course, i.e., averaging over all students. However, individual students can have completely different error patterns. RQ 3.3 asks whether we can identify different patterns among students. To evaluate the question, we create an error curve for each student, split up into different categories. We use k-means clustering to identify similar error patterns. The features used are the individual frequencies and errors for each exercise and an additional feature describing the number of missing exercises since not all students have solved all exercises. The clustering resulted in 3 clusters defining different kinds of error patterns. We use the centroid of each cluster to analyze how a typical error path looks like for a student of that cluster. The first group of students, i.e., cluster, describes students who miss a lot of exercises (11-29), thus we will call them the *missers*. However, in the exercises which they did they had few errors in comparison to the other two groups. Also their duration was most often lower. The reason why they stopped submitting is not clear. But since still 92% of them participated at the exam and 73.6% passed, we do not think that is was because they were overchallenged and gave up. The second group describes students who submitted solutions to almost all exercises. Unfortunately, they were also clearly the group with most errors and the longest error duration for all categories. This also manifested in their exam outcomes where they had the highest fail rate among all groups. Although they made a lot of errors, they never gave up. Therefore, we call them the *fighters*. The last group refers to students who work on a lot of exercises, but do not produce as many errors as the fighters. They also have the highest pass rate in the exam. Therefore, we call them the *studious*. Although we can identify different patterns among the groups, the final exam outcomes do not differ that much. Since the centroids only represent the average pattern of all students within that group, we need a more elaborated investigation of the missers and fighters. This might help us to identify students which need support either because they are likely to give up (missers) or have a higher probability of failure (*fighters*).

Answer to RQ 3.3 – *How do students differ regarding the errors made? Can we identify different error patterns?*: We identified 3 different groups of student with regard to their error patterns: the *missers*, who miss most of the exercises but have a relatively low error frequency, the *fighters* who work on almost all exercises although they make a lot of errors, and the *studious* who do many exercises but have an error frequency slightly higher than the one of the missers. However, these classification by error patterns alone does not help in identifying students which require support.

7.2. Strengths and Limitations

In educational research, interpretable models are always preferred as they contain additional information about students or course content. PFA and AFM are interpretable models as the parameters of the logistic regression model contain information on the difficulty of KCs or the importance of practicing particular KCs. However, for KCs in the programming domain clearly exist dependencies, e.g., pointers and arrays are related in C, as well as the different forms of loops. Since logistic regression models assume independent variables, the values of estimated parameters are pointless for interpretation. Our DBN model is able to cope with all of the four properties of the programming domain, including the capability of modeling KC dependencies. While the prediction performance is comparable to that of logistic regression models, we get a better interpretable model. However, the better interpretability of the model comes at the cost of high computational effort. While logistic regression models grow linearly in the number of KCs, DBNs have to deal with the "curse of dimensionality" since the number of parameters for a node grows exponentially with the number of its parents. The more KCs we include into the model, the more data we need to achieve reliable parameter estimates. This also makes the evaluation of meta-parameters for a DBN difficult.

A limitation of our meta-parameter evaluation is that the determination of the different Q matrix definitions relies on ASTs resp. KAMs. However, these can only deal with compilable solutions. But, uncompilable solutions contain valuable information on missing knowledge which is at the moment simply ignored in our current modeling approach. In this thesis, we try to fill this gap by the additional analysis of programming errors.

The major drawback of our approach with regard to skill modeling is that our approaches for determining which KCs are applied successfully and which not are very modest at the moment. While it is obvious that in a correct solutions all KCs that are used in that solution are applied correctly, the contrary can not be expected for incorrect solutions. The performance and parameter interpretability of skill models highly depends on a convenient identification of the incorrectly applied KCs.

Through the manual analysis of programming errors, we obtain a data set which does not exist in such a form yet, containing *all* different types of errors and their locations. This leads to a more sophisticated view on programming errors and brings valuable insights. Unfortunately, manual analysis is not feasible in practice, since you can not always manually label about 20,000 solutions by hand to get insights into problems in the course. However, our manually validated data can lay the basis for future research on automatic error localization in programming solutions.

7.3. Threats to Validity

In this section, we discuss potential threats to construct, internal, and external validity of our studies.

7.3.1. Construct Validity

Construct validity describes in how far the selected measurements measure what they are intended to. A typical threat to construct validity can be introduced during coding. To minimize faults in our code for the construction of KAMs and the identification of variable roles, we thoroughly tested the code with multiple examples from the real code base used in our studies.

A threat to construct validity concerning the data we used concerns the classification of students' solutions into correct and incorrect. As we use automatic assessment, we could have misclassified a solution as correct although it is not completely correct, e.g., by omitting a border case in our tests. We used established software testing techniques to derive test cases. Furthermore, the tests were improved over years such that the case of misclassification should occur very rarely and, thus, have only little effect on the overall results. On the other hand, we also may have problems with classifying a correct solution as incorrect because sometimes only small deviations of the expected output can lead to a classification as incorrect. Although this usually means that the solution is actually incorrect, it is only a very small mistake in the output generation. This would be no problem if we had a more precise localization of the incorrectly applied KCs, which directly leads to our next threat to validity.

The approaches we used for the identification of incorrectly applied KCs are very simplistic. So we cannot be sure whether the DBN actually models a student's knowledge. Unfortunately, knowledge can not be measured directly and, therefore, a real validation of the learned DBN is not possible. However, prediction performance is a first indicator of the model's validity. In addition, we checked for parameter plausibility which is based on common assumptions about human learning. For example, if a student knows all required KCs for the solution of an exercise, the model should infer that he/she will answer the exercise correctly, and if the student does not know one the required KCs, the model should infer that he/she will answer incorrectly [70] or if a student practices more, his/her knowledge should not decrease [164]. Since our learning curves also seem quite plausible, we think that this threat to construct validity can at least be not be that high.

Considering the error analysis, a threat to construct validity is how we calculated the category-based error metric. If an error type can be the result of different categories, we assume an equal distribution between those categories for the determination of the expected frequency. Nevertheless, for some error types one category is more likely than an other. For example, a missing semicolon is more likely caused because of sloppiness than by a lack in syntactic knowledge. While we can usually assume that the error is due to sloppiness by checking whether the students omitted all semicolons in his solution or only in one or two statements, for other error types this is unfortunately usually not possible. For that reason, assigning the same chance for all categories seems to be fair for us, as we cannot determine the concrete distribution solely from data.

7.3.2. Internal Validity

Internal validity describes in how far the casual relationship established for the study is trustworthy and if there exist no further explanations. A threat to internal validity concerning our data is that students are also able to test their solutions offline. Thus, intermediate steps of some students are missing and the upload frequency, and thus the progress step with each submission, can vary. When considering the step definition *every* this can lead to an unbalanced distribution of successes and failures between different students. However, a good skill model should be able to deal with this imbalance since it can not be distinguished between a student who immediately is able to solve everything and a student who needs several attempts offline before submitting a correct solution. During manual analysis of the programming errors, we had not the impression that students tested their solutions in advance offline and if, then would be only a small portion. Thus, the results should not be affected that much.

A threat to internal validity of our error analysis is that the type of errors made highly depends on the exercise. For example, if we only have few exercises involving pointers, pointer errors will be less frequent than, e.g., errors related to input and output. Although we cannot claim that errors we found are the most common errors in general, we still can conclude that if an exercise offers a chance to produce one of the errors, it is likely that such an error actually occurs.

7.3.3. External Validity

External validity concerns to what extent the results of a study are generalizable. One major threat to external validity is that we used only data from one university. Although we showed skill models are generalizable to different years and we have a broad diversity within the solutions in our data set, we cannot be sure that our results are generalizable for other institutions. Because of the high effort, the error analysis was only performed on the data of one year. So we can not be sure that it is generalizable to future years. However, the fact that skill models are generalizable to future years can be an indicator that it would also work for the errors. In addition, our data contains only data for C programming. So, in Java other errors and KC learning curves can be found since it is a completely different programming paradigm.

Another threat to external validity is that the meta-parameters we investigated are not completely independent of each other. Thus we can only assure that the results are valid in equal set-ups, i.e., same sets of meta-parameters and parameter values. For example, having a method to locate the concrete incorrectly applied KCs could lead to completely different results.

8. Conclusion

In this section, we conclude our thesis by providing a summary and giving an outlook on potential future work.

8.1. Summary

In this thesis, we aimed at getting an impression of students' programming knowledge and error development. Therefore, we defined what we consider as programming knowledge. We focused on KCs which we can be derived from ASTs – as a syntactic component – and variable roles – as a semantic component. Since knowledge is not directly measurable, we decided to use skill models to estimate the knowledge. We started with showing that students actually write different code and we, therefore, need a skill model which is able to deal with a dynamic Q matrix. Since there are different meta-parameters which have to be considered when developing a skill model, we evaluated different meta-parameter settings. we configured PFA and AFM models with all combinations of potential meta-parameter values and used data from our introductory C programming course to evaluate how the meta-parameters affected the prediction performance. We showed that meta-parameterization plays a crucial role when it comes to prediction performance. We decided to use DBNs as our skill model since it allows to explicitly model dependencies between nodes, i.e., KCs. We proposed a topology for the DBN which is able to deal with all properties of the programming domain that we identified. We showed that the DBN performs as well as the models based on logistic regression. We used the DBN to derive a student's learning curve from observational data. By aggregating over all students we got a learning curve for each KC which we used to identify KCs which are easy resp. hard to learn. One major finding of the learning curve analysis was that most KCs do not achieve a mastery level, i.e., are not learned completely. For teaching this means, that we should provide more exercises, and thus practice opportunities to the students.

In a second step, we focused on the programming errors made by the students. Therefore, we manually inspected all solutions of the students and identified 105 different error types in total. We assigned the different error types to 6 error categories, namely syntactic, conceptual, strategic, sloppiness, misinterpretation and domain errors. Based on the manual classification of errors, we determined the error frequency, duration, severity, and re-occurrence rate on a per-student, per-error-type, and per-category basis. We found out that most errors made by students are due to sloppiness and due to a lack in strategic knowledge which is coherent to the findings by Spohrer and Soloway [82] who identified plan composition as the major problem of programming beginners. For error duration and re-occurrence rate, we could not find any significant differences between the different categories.

In a third step, we had a look at the development of error frequency and duration for the different error categories over time. The results show that the number of occurrences of all error categories decrease towards the end of the course. However, the number of syntactic, conceptual, and strategic errors first increases before its starts decreasing. So these errors need some kind of familiarization in the beginning before students learn to avoid them.

In the last step, we aimed at identifying common patterns in the error development for students. We used the frequency and duration at each time step, i.e., exercise and a feature which describes how many exercises a student misses as input for k-means clustering to group students by their error patterns. We identified 3 groups of students. The *missers* are students who missed a large portion of the exercise. However the missers had only a low error frequency. The *fighters* refer to students who worked on almost all exercises although they produced a lot of errors and need longer time to fix the errors. We called the third group of students *studious*. As fighters they worked on a lot of exercises, but their error frequency and duration was much lower than the one of the fighters.

8.2. Outlook

While a thorough analysis of the error landscape can deliver valuable insights into problematic concepts, the approach of determining the error landscape has to be applicable in practice. For that purpose, the manual analysis has to be replaced as much as possible by an automated analysis. Our manually labeled error data provides a good foundation for future research. It can be used to evaluate for which of the errors an automatic detector can be developed. For instance, a missing check for an invalid input could be easily checked when having a reference implementation which we usually have in teaching.

Although the prediction performance of our skill models is acceptable, we believe that it can be improved. In our view, the major weakness of our models is the approach for determining which KCs are applied correctly and which not. One potential approach would be to use AST or KAM differencing to transform an incorrect solutions into the most similar correct one. The differences between both solutions are potential locations of defects. A more fine-grained comparison of these locations can be used to identify the concrete fault and, thus, the affected KC. Alternatively, when appropriate error detectors can be developed as proposed above, we could assign each error type with the affected KCs to have more precise reasons for an incorrect solutions than only syntactic differences.

One limitation of our skill modeling approach is that we only consider compilable solutions. Future work can elaborate on how to construct KAMs from an incomplete AST. Alternatively, we could integrate programming errors into our skill model. However, this requires an appropriate error detector and would also raise the complexity of our model coming with high computational costs. Furthermore, we have defined the dependencies between KCs, i.e., the domain model, manually. However, not only parameters can be learned from data but also the structure [165]. This may also reduce parameters that have to be learned, when the algorithm identifies some dependencies as not important by omitting them.

We used k-means clustering to identify different error patterns among students. But, the patterns are only limited helpful in identifying students who need support. Instead of clustering, we could use classification approaches on student's error landscape and their individual knowledge development to identify students who are at risk from dropping or failing the course. Together with the improvements and extensions mentioned above, this would provide a convenient basis for an adaptive system.

During the manual inspection of the programming errors, we noticed several flaws in students' code, e.g., unused variables or solutions which are more complex than necessary. In future work, we can also consider code smells and "algorithmic smells" to train students not only to write correct, but also efficient and readable programs. Furthermore programming abilities are more than the knowledge of syntactic elements of the language. We made a first step towards a more holistic model by integrating variable roles as a semantic layer to our model and explicitly modeling the problem solving ability. In future work, we can also consider higher level knowledge like cliches or units (see Section 2.1) as cliches often are a composition of several SAPPs. In this work, we mainly focused on the code *writing* ability. An extension could also be the code reading ability, especially the evaluation of testing and debugging skills.

Bibliography

[1] C. J. Butz, S. Hua, and R. B. Maguire, “A web-based bayesian intelligent tutoring system for computer programming,” *Web Intelligence and Agent Systems*, vol. 4, no. 1, pp. 61–81, 2006.

[2] N. Pennington and B. Grabowski, “The Tasks of Programming,” in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore, Eds. Elsevier, 1990, pp. 45–62.

[3] K. Bertels, P. Vanneste, and C. Backer, “A cognitive model of programming knowledge for procedural languages,” in *International Conference on Computer Assisted Learning*. Springer Verlag, 1992, vol. 602 LNCS, pp. 124–135.

[4] D. Koller and N. Friedman, *Probabilistic Graphical Methods Principles and Techniques*. Cambridge, MA, USA: The MIT Press, 2009.

[5] E. Millán and J. L. Pérez-De-La-Cruz, “A Bayesian diagnostic algorithm for student modeling and its evaluation,” *User Modeling and User-Adapted Interaction*, vol. 12, no. 2-3, pp. 281–330, 2002.

[6] A. Ko and B. Myers, “Development and evaluation of a model of programming errors,” in *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*. IEEE, 2003, pp. 7–14.

[7] M. Prensky, “Digital Natives, Digital Immigrants,” Tech. Rep. 5, 2001.

[8] A. Merkel. Rede von Bundeskanzlerin Merkel zur Eröffnung der CeBIT 2017 am 19. März 2017. Accessed on: 2021-01-28. [Online]. Available: https://www.bundeskanzlerin.de/bkin-de/aktuelles/rede-von-bundeskanzlerin-merkel-zur-eroeffnung-der-cebit-2017-am-19-maerz-2017-789606

[9] Coursera. Accessed on: 2021-01-28. [Online]. Available: https://de.coursera.org/

[10] edX. Accessed on: 2021-01-28. [Online]. Available: https://www.edx.org/

[11] Udacity. Accessed on: 2021-01-28. [Online]. Available: https://www.udacity.com/

[12] J. Mostow and J. Beck, “Some useful tactics to modify, map and mine data from intelligent tutors,” *Natural Language Engineering*, vol. 12, no. 2, pp. 195–208, jun 2006.

[13] D. Sleeman and J. S. Brown, *Intelligent Tutoring Systems*. London: Academic Press, 1982.

[14] K. R. Koedinger, A. T. Corbett, and C. Perfetti, “The Knowledge-Learning-Instruction Framework: Bridging the Science-Practice Chasm to Enhance Robust Student Learning,” *Cognitive Science*, vol. 36, no. 5, pp. 757–798, jul 2012.

[15] E. Alepis, M. Virvou, and K. Kabassi, “Mobile education: Towards affective bi-modal interaction for adaptivity,” in *2008 Third International Conference on Digital Information Management*. IEEE, nov 2008, pp. 51–56.

[16] R. S. Baker, “Modeling and understanding students’ off-task behavior in intelligent tutoring systems,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, apr 2007, pp. 1059–1068.

[17] A. Balakrishnan, “On Modeling the Affective Effect on Learning,” in *Multi-disciplonary Trends in Artificial Intelligence*, C. Sombattheera, A. Agarwal, S. K. Udgata, and K. Lavangnananda, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 225–235.

[18] C. Conati and X. Zhou, “Modeling Students’ Emotions from Cognitive Appraisal in Educational Games,” in *Intelligent Tutoring Systems*, S. A. Cerri, G. Gouardères, and F. Paraguaçu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 944–954.

[19] C. Conati and H. Maclaren, “Empirically building and evaluating a probabilistic model of user affect,” *User Modeling and User-Adapted Interaction*, vol. 19, no. 3, pp. 267–303, aug 2009.

[20] A. Kofod-Petersen, S. Abbas Petersen, G. Griff Bye, L. Kolås, and A. Staupe, “Learning in an Ambient Intelligent Environment. Towards Modelling Learners through Stereotypes,” *Revue d’intelligence artificielle*, vol. 22, no. 5, pp. 569–588, oct 2008.

[21] R. S. Baker, A. T. Corbett, K. R. Koedinger, and A. Z. Wagner, “Off-task behavior in the cognitive tutor classroom: When students "game the system",” in *Conference on Human Factors in Computing Systems - Proceedings*, 2004, pp. 383–390.

[22] S. Cetintas, Luo Si, Yan Ping Xin, and C. Hord, “Automatic Detection of Off-Task Behaviors in Intelligent Tutoring Systems with Machine Learning Techniques,” *IEEE Transactions on Learning Technologies*, vol. 3, no. 3, pp. 228–236, jul 2010.

[23] B. Jia, S. Zhong, T. Zheng, and Z. Liu, “The Study and Design of Adaptive Learning System Based on Fuzzy Set Theory,” in *Transactions on Edutainment IV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–11.

[24] C. Carmona, G. Castillo, and E. Millán, “Designing a Dynamic Bayesian Network for Modeling Students’ Learning Styles,” in *2008 Eighth IEEE International Conference on Advanced Learning Technologies*. IEEE, 2008, pp. 346–350.

[25] N. Salim and N. Haron, “The Construction of Fuzzy Set and Fuzzy Rule for Mixed Approach in Adaptive Hypermedia Learning System,” in *Technologies for E-Learning and Digital Entertainment*, Z. Pan, R. Aylett, H. Diener, X. Jin, S. Göbel, and L. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 183–187.

[26] H. D. Surjono and J. R. Maltby, “Adaptive Educational Hypermedia Based on Multiple Student Characteristics,” in *Advances in Web-Based Learning - ICWL 2003, Second International Conference,*, 2003, pp. 442–449.

[27] M. C. Desmarais and R. S. J. d. Baker, “A review of recent advances in learner and skill modeling in intelligent learning environments,” *User Modeling and User-Adapted Interaction*, vol. 22, no. 1-2, pp. 9–38, apr 2012.

[28] J. L. Stansfield, B. P. CArr, and I. P. Godstein, “Wumpus Advisor I. A First Implementation of a Program That Tutors Logical and Probabilistic Reasoning Skills,” p. 68, oct 1976.

[29] M. J. Mayo, “Bayesian Student Modelling and Decision-Theoretic Selection of Tutorial Actions in Intelligent Tutoring Systems,” Ph.D. dissertation, University of, 2001.

[30] C. Romero and S. Ventura, “Educational Data Mining: A Review of the State of the Art,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 6, pp. 601–618, nov 2010.

[31] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. Á. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll, “Educational Data Mining and Learning Analytics in Programming,” in *Proceedings of the 2015 ITiCSE on Working Group Reports*. New York, NY, USA: ACM, jul 2015, pp. 41–63.

[32] M. Berges and P. Hubwieser, “Evaluation of Source Code with Item Response Theory,” in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, vol. 2015-June. New York, NY, USA: ACM, jun 2015, pp. 51–56.

[33] J. Kasurinen and U. Nikula, “Estimating programming knowledge with Bayesian knowledge tracing,” in *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education - ITiCSE ’09*. New York, NY, USA: Association for Computing Machinery, aug 2009, pp. 313–317.

[34] M. Yudelson, R. Hosseini, A. Vihavainen, and P. Brusilovsky, "Investigating Automated Student Modeling in a Java MOOC," *Proceedings of the 7th International Conference on Educational Data Mining.*, pp. 261–264, 2014.

[35] R. Pettit, J. Homer, R. Gee, S. Mengel, and A. Starbuck, "An Empirical Study of Iterative Improvement in Programming Assignments," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, vol. 5. New York, NY, USA: ACM, feb 2015, pp. 410–415.

[36] K. Sharma, P. Jermann, and P. Dillenbourg, "Identifying Styles and Paths toward Success in MOOCs," in *8th International Conference on Educational Data Mining*, 2015, pp. 408–411.

[37] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall, "Analyzing Student Work Patterns Using Programming Exercise Data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15*. New York, New York, USA: ACM Press, 2015, pp. 18–23.

[38] J. Spacco, D. Fossati, J. Stamper, and K. Rivers, "Towards improving programming habits to create better computer science course outcomes," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education - ITiCSE '13*. New York, New York, USA: ACM Press, 2013, p. 243.

[39] N. J. Falkner and K. E. Falkner, "A fast measure for identifying at-risk students in computer science," in *Proceedings of the ninth annual international conference on International computing education research - ICER '12*. New York, New York, USA: ACM Press, 2012, p. 55.

[40] I. Koprinska, J. Stretton, and K. Yacef, "Students at Risk : Detection and Remediation," in *Proceeding of the 8th International Conference on Educational Data Mining, EDM15*, 2015, pp. 512–515.

[41] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice Java programmers through the analysis of online protocols," in *Proceedings of the seventh international workshop on Computing education research.* New York, NY, USA: ACM, aug 2011, pp. 85–92.

[42] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11*. New York, New York, USA: ACM Press, 2011, p. 499.

[43] L. Werner, C. McDowell, and J. Denner, "Middle school students using alice: What can we learn from logging data?" in *SIGCSE 2013 - Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 2013, pp. 507–512.

[44] B. Adcock, P. Bucci, W. D. Heym, J. E. Hollingsworth, T. Long, and B. W. Weide, “Which pointer errors do students make?” in *Proceedinds of the 38th SIGCSE technical symposium on Computer science education - SIGCSE ’07*. New York, New York, USA: ACM Press, 2007, p. 9.

[45] A. Altadmri and N. C. Brown, “37 Million Compilations,” in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, feb 2015, pp. 522–527.

[46] P. Denny, A. Luxton-Reilly, and E. Tempero, “All syntax errors are not equal,” in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education - ITiCSE ’12*. New York, New York, USA: ACM Press, 2012, p. 75.

[47] C. Norris, F. Barry, J. B. Fenwick Jr., K. Reid, and J. Rountree, “ClockIt,” in *Proceedings of the 13th annual conference on Innovation and technology in computer science education - ITiCSE ’08*. New York, New York, USA: ACM Press, 2008, p. 37.

[48] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein, “Modeling how students learn to program,” in *Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE ’12*. New York, New York, USA: ACM Press, 2012, p. 153.

[49] J. L. Devore and K. N. Berk, *Modern Mathematical Statistics with Applications*, ser. Springer Texts in Statistics. New York, NY: Springer New York, 2012.

[50] U. Braga-Neto, *Fundamentals of Pattern Recognition and Machine Learning*. Cham: Springer International Publishing, 2020.

[51] B. Shneiderman and R. Mayer, “Syntactic/semantic interactions in programmer behavior: A model and experimental results,” *International Journal of Computer & Information Sciences*, vol. 8, no. 3, pp. 219–238, jun 1979.

[52] J. R. Anderson, *The Architecture of Cognition*. Cambridge, MA, USA: Harvard University Press, 1983.

[53] B. Shneiderman, “Teaching programming: A spiral approach to syntax and semantics,” *Computers & Education*, vol. 1, no. 4, pp. 193–197, jan 1977.

[54] P. Bayman and R. E. Mayer, “Using conceptual models to teach BASIC computer programming.” *Journal of Educational Psychology*, vol. 80, no. 3, pp. 291–298, sep 1988.

[55] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs," *Communications of the ACM*, vol. 26, no. 11, pp. 853–860, nov 1983.

[56] V. Jonckers, "A Framework for Modeling Programming Knowledge," *AI Communications*, vol. 2, no. 2, pp. 72–87, 1989.

[57] J. Sajaniemi, "An empirical analysis of roles of variables in novice-level procedural programs," in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE Comput. Soc, 2002, pp. 37–39.

[58] K. D. Cooper and L. Torczon, "Intermediate Representations," in *Engineering a Compiler*. Elsevier, 2012, pp. 221–268.

[59] ——, "Parsers," in *Engineering a Compiler*. Elsevier, 2012, pp. 83–159.

[60] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. New York, NY, USA: ACM, sep 2014, pp. 313–324.

[61] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 493–504, jun 1996.

[62] I. Ben-Gal, "Bayesian Networks," in *Encyclopedia of Statistics in Quality & Reliability*, F. Ruggeri, F. Faltin, and R. Kenett, Eds. Cham: Wiley & Sons, 2007.

[63] S. L. Lauritzen and D. J. Spiegelhalter, "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 50, no. 2, pp. 157–194, jan 1988.

[64] T. Kaser, S. Klingler, A. G. Schwing, and M. Gross, "Dynamic Bayesian Networks for Student Modeling," *IEEE Transactions on Learning Technologies*, vol. 10, no. 4, pp. 450–462, oct 2017.

[65] G. F. Cooper and E. Herskovits, "A Bayesian method for the induction of probabilistic networks from data," *Machine Learning*, vol. 9, no. 4, pp. 309–347, oct 1992.

[66] J. Pearl and T. S. Verma, "A theory of inferred causation," in *Studies in Logic and the Foundations of Mathematics*, 1995, vol. 134, no. C, pp. 789–811.

[67] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*, ser. Lecture Notes in Statistics. New York, NY: Springer New York, 1993, vol. 81.

[68] A. T. Corbett and J. R. Anderson, “Knowledge tracing: Modeling the acquisition of procedural knowledge,” *User Modelling and User-Adapted Interaction*, vol. 4, no. 4, pp. 253–278, 1995.

[69] Z. A. Pardos and N. T. Heffernan, “KT-IDEM: Introducing Item Difficulty to the Knowledge Tracing Model,” in *User Modeling, Adaption and Personalization*, J. A. Konstan, R. Conejo, J. L. Marzo, and N. Oliver, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 243–254.

[70] R. S. J. d. Baker, A. T. Corbett, and V. Aleven, “More Accurate Student Modeling through Contextual Estimation of Slip and Guess Probabilities in Bayesian Knowledge Tracing,” in *Intelligent Tutoring Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5091 LNCS, pp. 406–415.

[71] C. Conati, A. Gertner, and K. Vanlehn, “Using Bayesian networks to manage uncertainty in student modeling,” *User Modelling and User-Adapted Interaction*, vol. 12, no. 4, pp. 371–417, nov 2002.

[72] Y. Huang, J. D. G. Hollstein, and P. Brusilovsky, “Modeling Skill Combination Patterns for Deeper Knowledge Tracing,” in *The 6th Intl. Workshop on Personalization Approaches in Learning Environments (PALE 2016) in the 24th Conf. on User Modeling, Adaptation and Personalization (UMAP 2016)*, 2016.

[73] W. J. van der Linden and R. K. Hambleton, *Handbook of Modern Item Response Theory.* New York, NY, USA: Springer New York, sep 1997.

[74] H. Cen, K. Koedinger, and B. Junker, “Learning Factors Analysis – A General Method for Cognitive Model Evaluation and Improvement,” in *Intelligent Tutoring Systems*, M. Ikeda, K. D. Ashley, and T.-W. Chan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 164–175.

[75] ——, “Comparing Two IRT Models for Conjunctive Skills,” in *Intelligent Tutoring Systems*, B. P. Woolf, E. Aïmeur, R. Nkambou, and S. Lajoie, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 796–798.

[76] P. I. Pavlik, H. Cen, and K. R. Koedinger, “Performance Factors Analysis – A New Alternative to Knowledge Tracing,” in *Proceedings of the 2009 Conference on Artificial Intelligence in Education: Building Learning Systems That Care: From Knowledge Representation to Affective Modelling*. Amsterdam, The Netherlands: IOS Press, 2009, pp. 531–538.

[77] Y. Xu and J. Mostow, “Using logistic regression to trace multiple subskills in a dynamic bayes net,” in *EDM 2011 - Proceedings of the 4th International Conference on Educational Data Mining*, M. Pechenizkiy, T. Calders, C. Conati, S. Ventura, C. Romero, and J. Stamper, Eds., Eindhoven, The Netherlands, 2011, pp. 241–245.

[78] J. P. González-Brenes and Y. Huang, "General features in knowledge tracing: Applications to multiple subskills, temporal item response theory, and expert knowledge," in *Proceedings of the 7th International Conference on Educational Data Mining.*, J. Stamper, Z. Pardos, M. Mavrikis, and B. M. Mclaren, Eds., 2014, pp. 84–91.

[79] C. Piech, J. Spencer, J. Huang, S. Ganguli, M. Sahami, L. Guibas, and J. Sohl-Dickstein, "Deep Knowledge Tracing," *Advances in Neural Information Processing Systems*, vol. 2015-Janua, pp. 505–513, jun 2015.

[80] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," in *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, vol. 35, no. 1, jan 2003, pp. 153–156.

[81] W. L. Johnson, E. Soloway, B. Cutler, and S. Draper, "Bug Catalogue: I," Yale University, New Haven, CT, USA, Tech. Rep., 1983.

[82] J. C. Spohrer and E. Soloway, "Novice mistakes: are the folk wisdoms correct?" *Communications of the ACM*, vol. 29, no. 7, pp. 624–632, jul 1986.

[83] D. Zehetmeier, A. Böttcher, A. Brüggemann-Klein, and V. Thurner, "Development of a Classification Scheme for Errors Observed in the Process of Computer Programming Education," in *HEAd'15. Conference on Higher Education Advances.* Editorial Universitat Politècnica de València, jun 2015, pp. 475–484.

[84] O. Anderson and D. Krathwohl, "Taxonomy for Learning, Teaching, and Assessing (A Revision Of Bloom's Taxonomy of Educational Objecives)," Tech. Rep., 2001.

[85] N. C. Brown and A. Altadmri, "Investigating novice programming mistakes," in *Proceedings of the tenth annual conference on International computing education research - ICER '14.* New York, New York, USA: ACM Press, 2014, pp. 43–50.

[86] J. Jackson, M. Cobb, and C. Carver, "Identifying Top Java Errors for Novice Programmers," in *Proceedings Frontiers in Education 35th Annual Conference.* IEEE, 2015, pp. T4C–24–T4C–27.

[87] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the 2006 international workshop on Computing education research - ICER '06*, vol. 2006. New York, New York, USA: ACM Press, 2006, p. 73.

[88] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *Proceedings - Frontiers in Education Conference, FIE*, vol. 2015-Febru, no. February, 2015.

[89] A. Ettles, A. Luxton-Reilly, and P. Denny, "Common logic errors made by novice programmers," in *Proceedings of the 20th Australasian Computing Education Conference on - ACE '18*. New York, New York, USA: ACM Press, jan 2018, pp. 83–89.

[90] M. Mayo and A. Mitrovic, "Using a Probabilistic Student Model to Control Problem Difficulty," in *Intelligent Tutoring Systems*, G. Gauthier, C. Frasson, and K. VanLehn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 524–533.

[91] S. Ohlsson, "Constraint-Based Student Modeling," in *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, J. E. Greer and G. I. McCalla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 167–189.

[92] I.-H. Hsiao, S. Sosnovsky, and P. Brusilovsky, "Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming," *Journal of Computer Assisted Learning*, vol. 26, no. 4, pp. 270–283, jul 2010.

[93] R. Hosseini and P. Brusilovsky, "JavaParser; A Fine-Grain Concept Indexing Tool for Java Problems," in *AIED Workshops*, Memphis, TN, USA, 2013.

[94] Y. Huang, J. Guerra-Hollstein, J. Barria-Pineda, and P. Brusilovsky, "Learner Modeling for Integration Skills," in *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, ser. UMAP '17. New York, NY, USA: ACM, jul 2017, pp. 85–93.

[95] Y. Huang, Y. Xu, and P. Brusilovsky, "Doing More with Less: Student Modeling and Performance Prediction with Reduced Content Models," in *User Modeling, Adaptation, and Personalization*, V. Dimitrova, T. Kuflik, D. Chin, F. Ricci, P. Dolog, and G.-J. Houben, Eds. Cham: Springer International Publishing, 2014, pp. 338–349.

[96] K. Rivers, E. Harpstead, and K. Koedinger, "Learning Curve Analysis for Programming," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*. New York, NY, USA: ACM, aug 2016, pp. 143–151.

[97] K. Rivers and K. R. Koedinger, "Automating Hint Generation with Solution Space Path Construction," in *Intelligent Tutoring Systems*, S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, Eds. Cham: Springer International Publishing, 2014, pp. 329–339.

[98] P. Blikstein, "Using learning analytics to assess students' behavior in open-ended programming tasks," in *Proceedings of the 1st International Conference on Learning Analytics and Knowledge - LAK '11*. New York, New York, USA: ACM Press, 2011, p. 110.

[99] R. Hosseini, A. Vihavainen, and P. Brusilovsky, “Exploring Problem Solving Paths in a Java Programming Course,” *Proceedings of the Psychology of Programming Interest Group Annual Conference*, pp. 65–76, 2014.

[100] L. Wang, A. Sy, L. Liu, and C. Piech, “Learning to represent student knowledge on programming exercises using deep learning,” in *Proceedings of the 10th International Conference on Educational Data Mining, EDM 2017*, 2017, pp. 324–329.

[101] H. Meier, E. Tonisson, M. Lepp, and P. Luik, “Behaviour Patterns of Learners while Solving a Programming Task: an Analysis of Log Files,” in *2020 IEEE Global Engineering Education Conference (EDUCON)*, vol. 2020-April. IEEE, apr 2020, pp. 685–690.

[102] R. E. Pattis, J. Roberts, and M. Stehlik, *Karel the robot (2nd ed.): a gentle introduction to the art of programming*, 2nd ed. New York, NY, USA: John Wiley & Sons, Ltd., 1995.

[103] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, “Conditions of Learning in Novice Programmers,” *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 37–55, feb 1986.

[104] M. Yee-King, L. McCallum, M. T. Llano, V. Ruzicka, M. D’Inverno, and M. Grierson, “Examining Student Coding Behaviours in Creative Computing Lessons using Abstract Syntax Trees and Vocabulary Analysis,” in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, vol. 20. New York, NY, USA: ACM, jun 2020, pp. 273–279.

[105] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, “Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming,” *Journal of the Learning Sciences*, vol. 23, no. 4, pp. 561–599, oct 2014.

[106] A. Ahadi, R. Lister, S. Lal, J. Leinonen, and A. Hellas, “Performance and Consistency in Learning to Program,” in *Proceedings of the Nineteenth Australasian Computing Education Conference on - ACE ’17*. New York, New York, USA: ACM Press, 2017, pp. 11–16.

[107] R. Smith and S. Rixner, “The error landscape: Characterizing the mistakes of novice programmers,” in *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, vol. 7. ACM, 2019, pp. 538–544.

[108] ISO/IEC, “ISO/IEC 9899:2011 - Programming languages — C,” Tech. Rep., 2011. [Online]. Available: https://www.iso.org/standard/57853.html

[109] Eclipse Foundation. Eclipse CDT. Accessed on: 2021-02-16. [Online]. Available: https://www.eclipse.org/cdt/

[110] R. Rist, "Schema creation in programming," *Cognitive Science*, vol. 13, no. 3, pp. 389–414, sep 1989.

[111] P. Byckling and J. Sajaniemi, "A role-based analysis model for the evaluation of novices' programming knowledge development," in *Proceedings of the 2006 international workshop on Computing education research - ICER '06*. New York, New York, USA: ACM Press, 2006, p. 85.

[112] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, jul 1987.

[113] A. Sheneamer and J. Kalita, "A Survey of Software Clone Detection Techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, mar 2016.

[114] M. Novak, M. Joy, and D. Kermek, "Source-code Similarity Detection and Detection Tools Used in Academia," *ACM Transactions on Computing Education*, vol. 19, no. 3, pp. 1–37, jun 2019.

[115] B. Hartmann, D. Macdougall, J. Brandt, and S. R. Klemmer, "What Would Other Programmers Do? Suggesting Solutions to Error Messages," in *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 1019–1028.

[116] J. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE Comput. Soc, 2001, pp. 103–112.

[117] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *ACM SIGCSE Bulletin*, vol. 8, no. 4, pp. 30–41, dec 1976.

[118] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato, "A plagiarism detection system," *ACM SIGCSE Bulletin*, vol. 13, no. 1, pp. 21–25, feb 1981.

[119] J. Faidhi and S. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Computers & Education*, vol. 11, no. 1, pp. 11–19, jan 1987.

[120] C. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *2008 16th IEEE International Conference on Program Comprehension*. IEEE, jun 2008, pp. 172–181.

[121] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360).* IEEE, 1999, pp. 109–118.

[122] M. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, may 1999.

[123] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," in *Journal of Universal Computer Science*, vol. 8, no. 11, 2002, pp. 1016–1038.

[124] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software: Practice and Experience*, vol. 37, no. 2, pp. 151–175, feb 2007.

[125] I. D. Baxter, A. Yahin, L. Moura, M. Sant' Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees. I," in *Proceedings of ICSM'98.* IEEE, 1998.

[126] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *29th International Conference on Software Engineering (ICSE'07).* IEEE, may 2007, pp. 96–105.

[127] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im, "Software plagiarism detection," in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM '13.* New York, New York, USA: ACM Press, 2013, pp. 1577–1580.

[128] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 2006, 2006, pp. 872–881.

[129] J. Crussell, C. Gibler, and H. Chen, "Attack of the Clones: Detecting Cloned Applications on Android Markets," in *Computer Security - ESORICS 2012.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–54.

[130] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering.* IEEE Comput. Soc, 2001, pp. 301–309.

[131] P. Jaccard, "Lois de distribution florale dans la zone alpine," *Bulletin de la Société vaudoise des sciences naturelles*, vol. 38, no. 144, p. 72, 1902.

[132] V. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707 – 710, 1966.

[133] R. A. Wagner and M. J. Fischer, “The String-to-String Correction Problem,” *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, jan 1974.

[134] E. Millán, T. Loboda, and J. L. Pérez-de-la Cruz, “Bayesian networks for student model engineering,” *Computers & Education*, vol. 55, no. 4, pp. 1663–1683, dec 2010.

[135] B. Martin, A. Mitrovic, K. R. Koedinger, and S. Mathan, “Evaluating and improving adaptive educational systems with learning curves,” *User Modeling and User-Adapted Interaction*, vol. 21, no. 3, pp. 249–283, aug 2011.

[136] T. Effenberger, R. Pelánek, and J. Čechák, “Exploration of the robustness and generalizability of the additive factors model,” in *Proceedings of the Tenth International Conference on Learning Analytics & Knowledge*. New York, NY, USA: ACM, mar 2020, pp. 472–479.

[137] E. Albrecht and J. Grabowski, “Sometimes It’s Just Sloppiness - Studying Students’ Programming Errors and Misconceptions,” in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, feb 2020, pp. 340–345.

[138] D. McCall and M. Kölling, “A new look at novice programmer errors,” *ACM Transactions on Computing Education*, vol. 19, no. 4, pp. 1–30, nov 2019.

[139] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, 1967, pp. 281–296.

[140] E. Albrecht, F. Gumz, and J. Grabowski, “Experiences in Introducing Blended Learning in an Introductory Programming Course,” in *Proceedings of the 3rd European Conference of Software Engineering Education*. New York, NY, USA: ACM, jun 2018, pp. 93–101.

[141] E. Albrecht and J. Grabowski, “Towards a framework for mining students’ programming assignments,” in *2016 IEEE Global Engineering Education Conference (EDUCON)*, vol. 10-13-Apri. IEEE, apr 2016, pp. 1096–1100.

[142] GCC, the GNU Compiler Collection - GNU Project. Accessed on: 2021-08-03. [Online]. Available: https://gcc.gnu.org/

[143] Clang C Language Family Frontend for LLVM. Accessed on: 2021-08-03. [Online]. Available: https://clang.llvm.org/

[144] Splint - Annotation-Assisted Lightweight Static Checking. Accessed on: 2021-08-03. [Online]. Available: https://splint.org/

[145] Cppcheck - A tool for static C/C++ code analysis. Accessed on: 2021-08-03. [Online]. Available: http://cppcheck.sourceforge.net/

[146] Clang Static Analyzer. Accessed on: 2021-08-03. [Online]. Available: https://clang-analyzer.llvm.org/

[147] vera++. Accessed on: 2021-08-03. [Online]. Available: https://bitbucket.org/verateam/vera/wiki/Home

[148] CUnit - A Unit Testing Framework for C. Accessed on: 2021-08-03. [Online]. Available: http://cunit.sourceforge.net/

[149] DejaGnu - GNU Test Framework. Accessed on: 2021-08-03. [Online]. Available: https://www.gnu.org/software/dejagnu/

[150] Linux Containers. Accessed on: 2021-08-03. [Online]. Available: https://linuxcontainers.org/

[151] L. BayesFusion. GeNIe Modeler. Accessed on: 2021-08-03. [Online]. Available: https://www.bayesfusion.com/genie/

[152] ——. SMILE: Structural Modeling, Inference, and Learning Engine. Accessed on: 2021-08-03. [Online]. Available: https://www.bayesfusion.com/smile/

[153] Y. Gong, J. E. Beck, and N. T. Heffernan, "Comparing Knowledge Tracing and Performance Factor Analysis by Using Multiple Model Fitting Procedures," in *Intelligent Tutoring Systems*, V. Aleven, J. Kay, and J. Mostow, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–44.

[154] M. Friedman, "A Comparison of Alternative Tests of Significance for the Problem of m Rankings," *The Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, mar 1940.

[155] F. Wilcoxon, "Individual Comparisons by Ranking Methods," in *Breakthroughs in Statistics: Methodology and Distribution*, S. Kotz and N. L. Johnson, Eds. New York, NY: Springer New York, 1992, pp. 196–202.

[156] H. Abdi, "The Bonferroni and Sidak Corrections for Multiple Comparisons," in *Encyclopedia of measurement and statistics*, N. Salkind, Ed. Thousand Oaks (CA): Sage, 2007, pp. 103–107.

[157] M. G. Kendall and B. B. Smith, "The Problem of m Rankings," *The Annals of Mathematical Statistics*, vol. 10, no. 3, pp. 275–287, sep 1939.

[158] M. Tomczak and E. Tomczak, “The need to report effect size estimates revisited. An overview of some recommended measures of effect size,” *Trends in Sport Sciences*, vol. 1, no. 21, pp. 19–25, 2014.

[159] S. Cafiso, A. Di Graziano, and G. Pappalardo, “Using the Delphi method to evaluate opinions of public transport managers on bus safety,” *Safety Science*, vol. 57, pp. 254–263, aug 2013.

[160] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences.* Routledge, may 2013.

[161] S. S. Sawilowsky, “New Effect Size Rules of Thumb,” *Journal of Modern Applied Statistical Methods*, vol. 8, no. 2, pp. 597–599, nov 2009.

[162] E. Albrecht, “Supplementary Material to "Sometimes It’s Just Sloppiness - Studying Student’s Programming Errors and Misconceptions", SIGCSE 2020,” 2020. [Online]. Available: https://www.researchgate.net/publication/337566114_Supplementary_Material_to_Sometimes_It’s_Just_Sloppiness_-_Studying_Student’s_Programming_Errors_and_Misconceptions_SIGCSE_2020

[163] L. Kaufman and P. J. Rousseeuw, Eds., *Finding Groups in Data*, ser. Wiley Series in Probability and Statistics. Hoboken, NJ, USA: John Wiley & Sons, Inc., mar 1990.

[164] A. Newell and P. S. Cn Rosenbloom, “Mechanisms of Skill Acquisition and the Law of Practice: Allen Newell and Paul S. Rosenbloom,” in *Cognitive Skills and Their Acquisition.* Psychology Press, oct 2013, pp. 12–66.

[165] S. Gao, Q. Xiao, Q. Pan, and Q. Li, “Learning Dynamic Bayesian Networks Structure Based on Bayesian Optimization Algorithm,” in *Advances in Neural Networks – ISNN 2007.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4492 LNCS, no. PART 2, pp. 424–431.

[166] J. D. Gould, “Some psychological evidence on how people debug computer programs,” *International Journal of Man-Machine Studies*, vol. 7, no. 2, pp. 151–182, mar 1975.

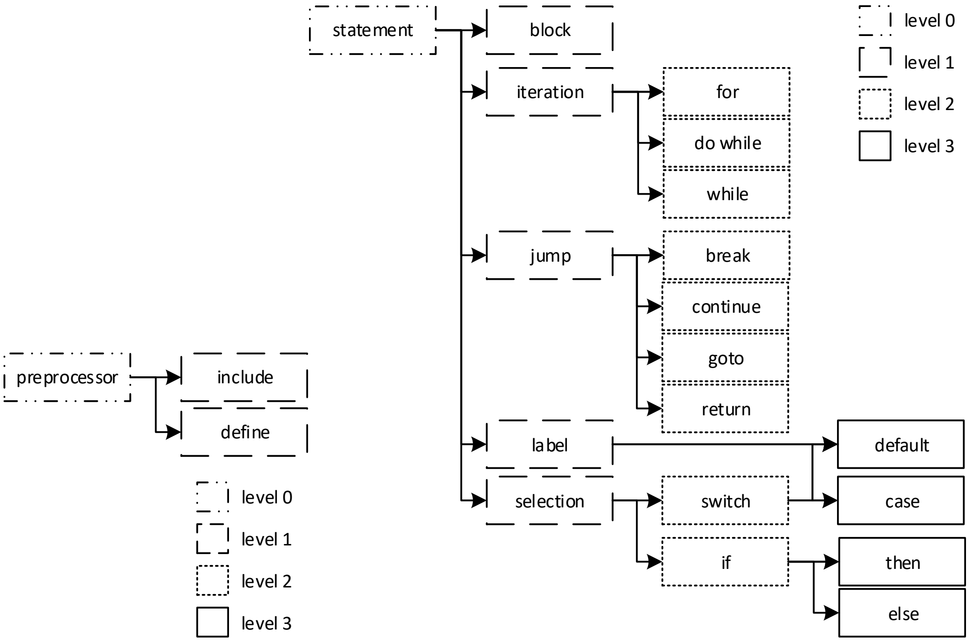
[167] M. Eisenberg and H. A. Peelle, “APL learning bugs,” in *Proceedings of the international conference on APL - APL ’83.* New York, New York, USA: ACM Press, 1983, pp. 11–16.

[168] D. E. Knuth, “The errors of tex,” *Software: Practice and Experience*, vol. 19, no. 7, pp. 607–685, jul 1989.

[169] M. Eisenstadt, “Tales of Debugging from the Front Lines,” *Empirical Studies of Programmers, 5th Workshop*, vol. 5365, pp. 86–112, 1993.

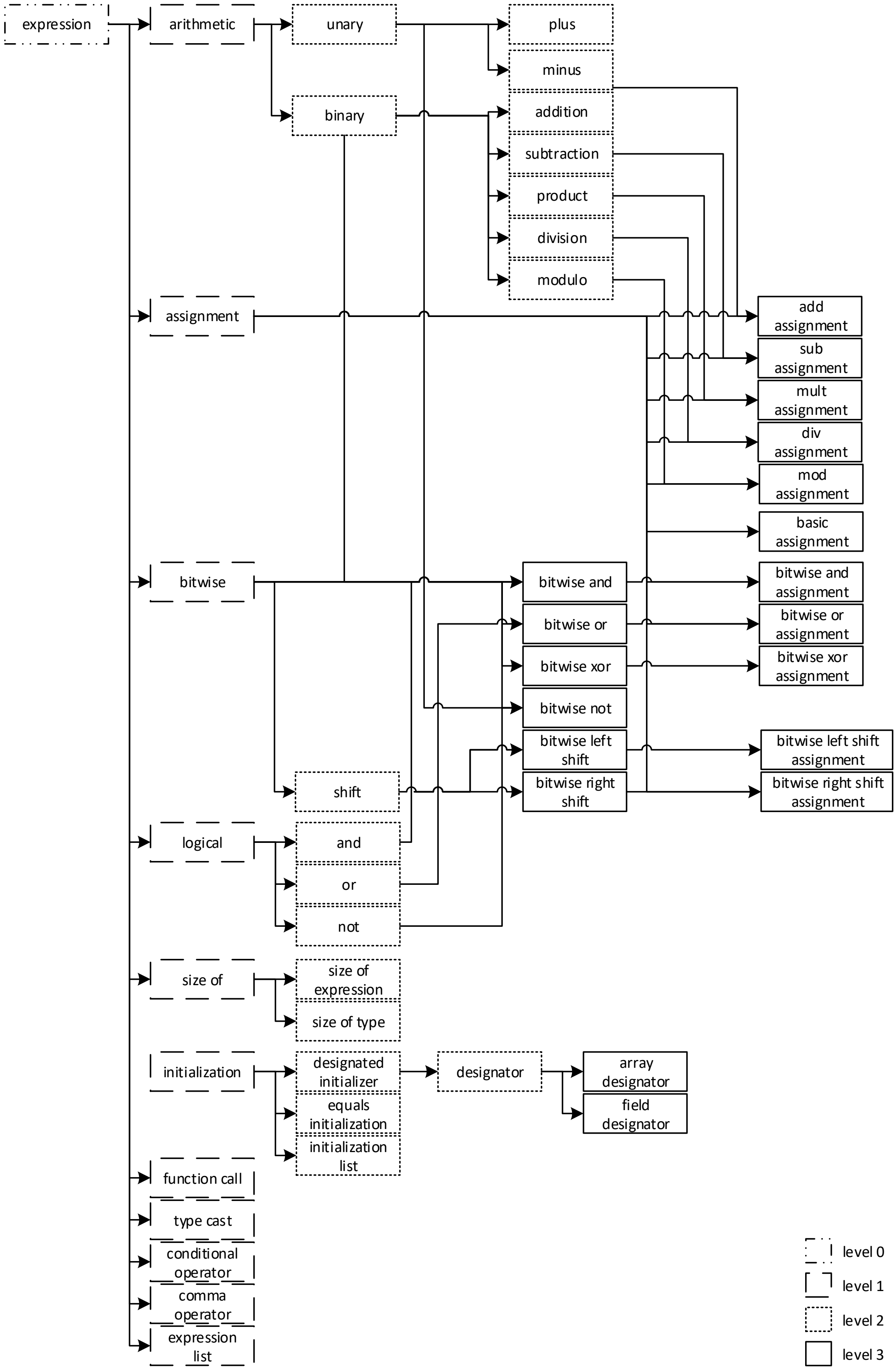
[170] R. R. Panko, “What We Know About Spreadsheet Errors,” *Journal of Organizational and End User Computing*, vol. 10, no. 2, pp. 15–21, apr 1998.

A. KC Hierarchy

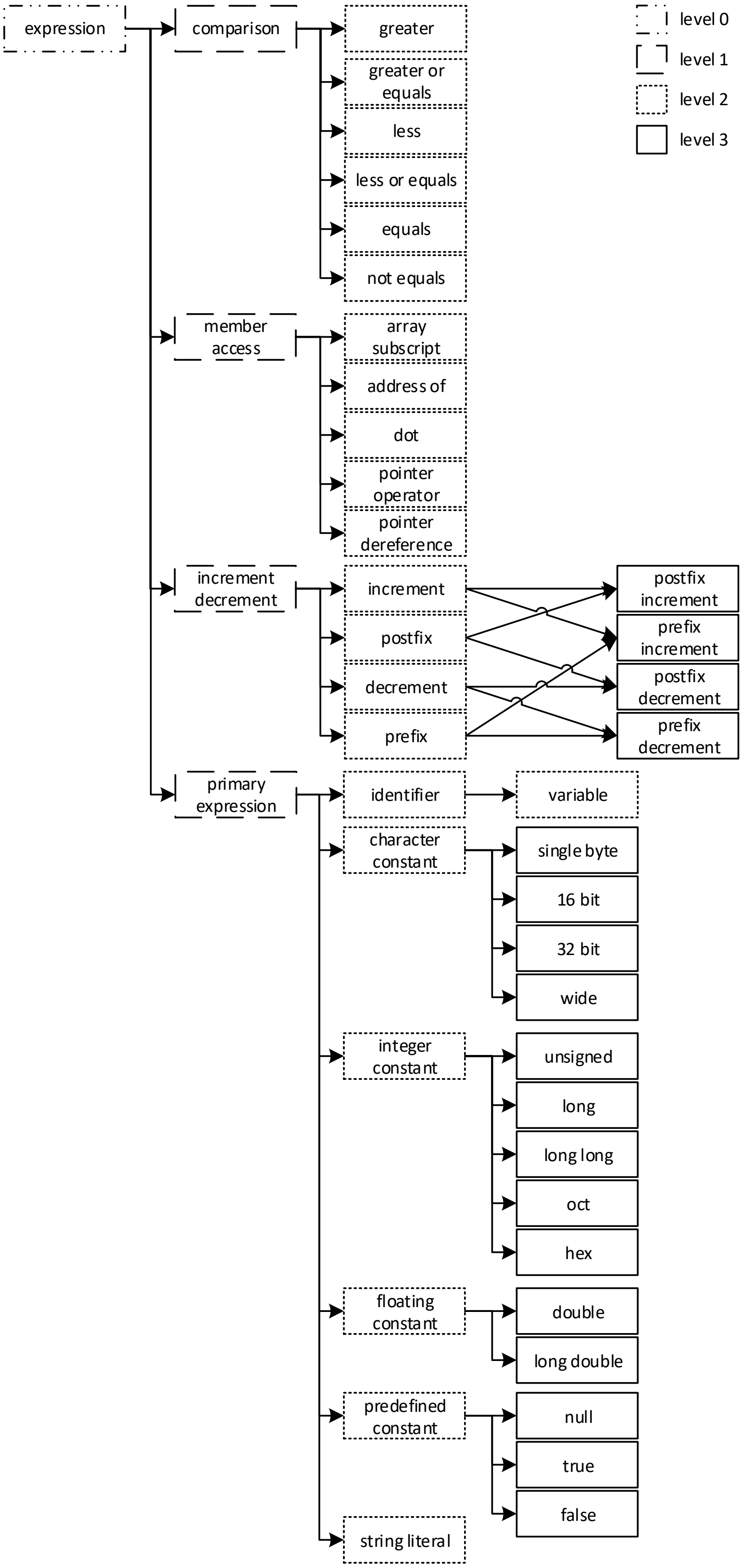


(a) preprocessor

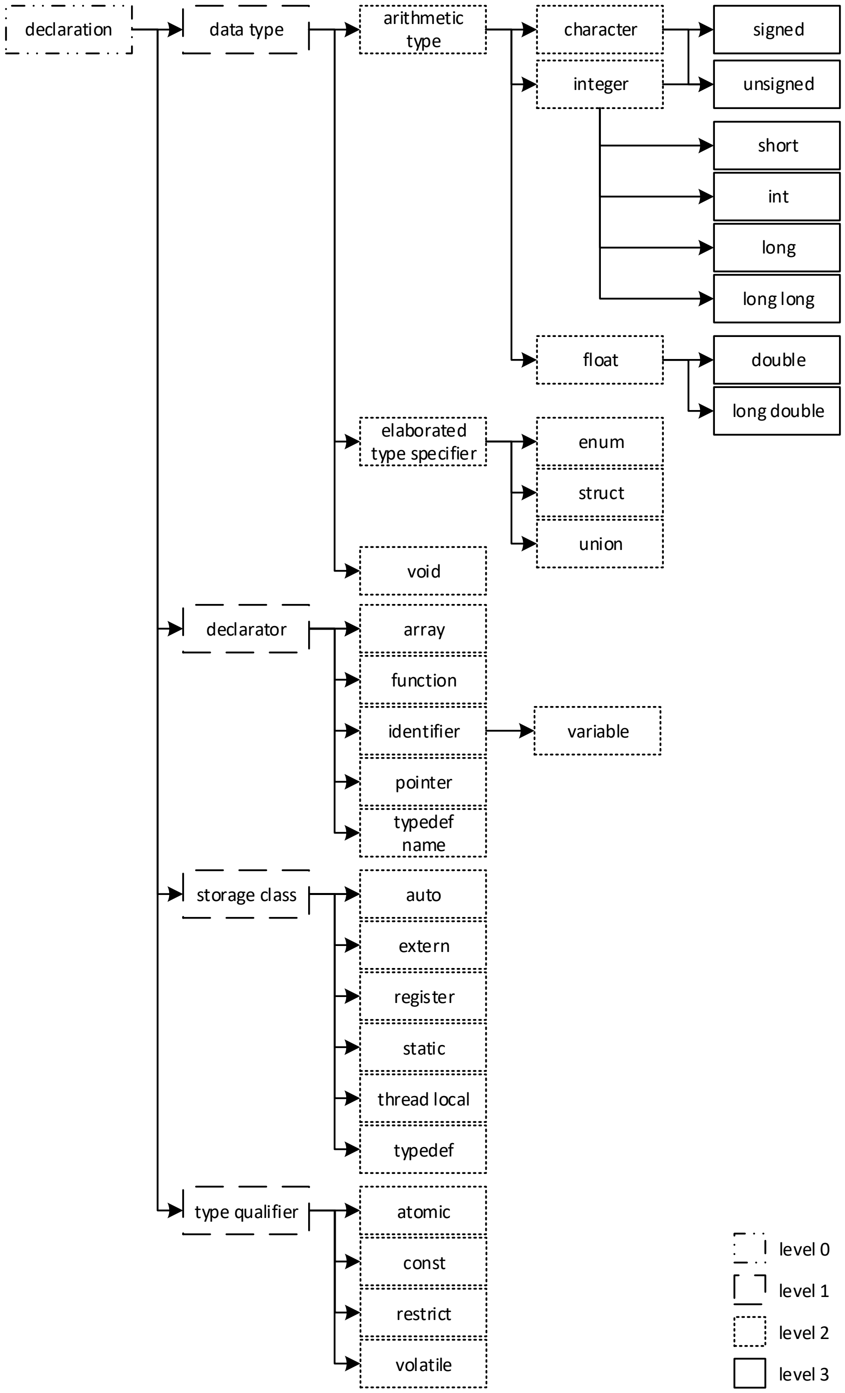
(b) statement



(c) expression



(d) expression



(e) declaration

Figure A.-2.: Hierarchy of KCs

B. Programming Error Categories

	Error	Description
Gould (1975) [166]	Assignment bug	Errors assigning variable values
	Iteration bug	Errors iterating
	Array bug	Errors accessing data in arrays
Eisenberg (1983) [167]	Visual bug	Clustering semantically related parts of expression
	Naive bug	Using branching & iteration instead of parallel processing
	Logical bug	Omitting or misusing logical connectives or relationals
	Dummy bug	Experience with other languages interfering
	Illiteracy bug	Difficulties with order of operations
	Gestalt bug	Not foreseeing side effects of commands
Johnson et al. (1983) [81]	Missing	Omitting required program element
	Spurious	Including unnecessary program element
	Misplaced	Putting necessary program element in wrong place
	Malformed	Putting incorrect program element in right place
Spohrer and Soloway (1986) [82]	Natural language problem	Confusing semantics of constructs because of their natural-language naming
	Inconsistency problem	Misunderstanding differences of a construct in different situations
	Human interpreter problem	Assuming computer has similar interpretation of code
	Summarization problem	Summarizing combinations of plans without considering implications of secondary functions in later plan compositions
	Optimization problem	Optimizing code without checking if optimization really can be carried out
	Previous experience problem	Plans from previous experience are re-used in situations where the plan does not completely fit
	Specialization problem	Abstract plan is inappropriate or not instantiated correctly
	Interpretation problem	Misunderstandings of the problem specification
	Boundary problem	Difficulties deciding on appropriate boundary points
	Unexpected cases problem	Omitting uncommon, unlikely, or boundary cases
	Cognitive load problem	Small but important parts of plans are dropped out or plan interactions are overlooked
Bayman and Mayer (1988) [54]	Syntactic	Incorrect syntax or rules of a programming language
	Conceptual	Misconception of programming constructs
	Strategic	Wrong plan or algorithm to achieve goal
Knuth (1989) [168]	Algorithm awry	Improperly implemented algorithms
	Blunder or botch	Accidentally writing code not to specifications
	Data structure debacle	Errors using and changing data structures

	Forgotten function	Missing implementation
	Language liability	Misusing or misunderstanding language/environment
	Mismatch between modules	Imperfectly knowing specs, interface; reversed arguments
	Reinforcement of robustness	Not handling erroneous input
	Surprise scenario	Unforeseen interactions in program elements
	Trivial typos	Incorrect syntax, reference, etc.
Eisenstadt (1993) [169]	Clobbered memory bugs	Overwriting memory, subscript out of bounds
	Vendor problems	Buggy compilers, faulty hardware
	Design logic bug	Unanticipated case, wrong algorithm
	Initialization bug	Erroneous type or initialization of variables
	Variable bugs	Wrong variable or operator used
	Lexical bugs	Lexical problem, bad parse, ambiguous syntax
	Language	Misunderstandings of language semantics
Panko (1998) [170]	Omission error	Omitting facts
	Logic error	Incorrect algorithm or incorrectly implemented algorithm
	Mechanical error	Typos
	Overload error	Working memory unable to complete task without errors
	Strong but wrong error	Functional fixedness (fixed mindset)
	Translation error	Misreading of specification
Hristova et al. (2003) [80]	Syntax Errors	Incorrect syntax of a programming language
	Semantic Errors	Improper use of programming constructs
	Logic Errors	Errors due to not respecting specification
Zehetmeier et al. (2015) [83]	Mental typo	Sloppiness
	Knowledge gap	Not knowing definitions or terms
	Misconception	Faulty understanding of a construct/concept
	Wrong choice	Inappropriate selection of solution process in a given setting
	Structural blindness	Inability to structure components in a given setting
	Quality gap	Inability to stick to quality standards
	Lack of innovation	Inability to construct a new solution from previous knowledge in a new contex
McCall and Kölling (2019) [138]	Incorrect attempt to use variable	Errors related to incorrect usage of a variable
	Incorrect variable declaration	Errors in the declaration of an variable
	Incorrect method call	Errors when calling a method, e.g., wrong number of parameters
	Incorrect method declaration	Errors related to the declaration of methods, e.g., missing return type
	Incorrect constructor call	Incorrect call of a constructor, e.g., wrong number of parameters
	Incorrect constructor declaration	Errors when declaring a constructor, e.g., call to “super” note first statement
	Incorrect use of class or type	Errors related to the use of a class, e.g., missing import
	Semantic error	Errors not involving variables, methods, or constructors which are related to semantics
	Simple syntactical error	Errors not involving variables, methods, or constructors which are related to syntax

	Statement outside method/block	Statements are place outside of a method or a class
	Uncategorized	All errors that do not fit in any of the other categories
Albrecht and Grabowski (2020) [137]	Sloppiness	Unintentional errors
	Misinterpretation	Incorrect understanding of specification
	Domain	Lack in domain knowledge
	Compile-time errors	Errors arising during compilation
	Run-time errors	Errors only occurring at run-time

Table B.1.: Categories of programming errors (based on [6])

C. Rule Descriptions for Variable Roles

Focal line	no focal line, only initialization
Focal scope	*
Usage	• read-only

(a) Constraints for the variable role *fixed value*

Focal line	(1) *var* = *var op fixedvalue* (2) *var* = *fixedvalue* op_{com} *var* (3) *var op*= *fixedvalue* (4) *var*++ (5) *var*-- (6) ++*var* (7) --*var*
Focal scope	loop - *
Usage	• initialization outside of scope • inside scope read-only

(b) Constraints for the variable role *stepper*

Focal line	(1) *var* = *var* +1 (2) *var* = 1+ *var* (3) *var*++ (4) ++*var*
Focal scope	if - loop - *
Usage	• initialization outside of scope • inside scope read-only

(c) Constraints for the variable role *counter*

Focal line	(1) *var* = getchar() (2) scanf("*formatspec*", &*var*)
Focal scope	* - loop - *
Usage	• initialization outside of scope • inside scope read-only

(d) Constraints for the variable role *MRH*

Focal line	(1) *var* = *var2*
Focal scope	if - * - loop - *
Usage	• initialization outside of scope in comparison in if • inside scope read-only

(e) Constraints for the variable role *MWH*

Focal line	(1) *var* = *var2 op var3*
Focal scope	*
Usage	• inside scope read-only

(f) Constraints for the variable role *transformation*

Focal line	(1) *var* = *var op var2* (2) *var op*= *var2*
Focal scope	loop - *
Usage	• initialization outside of scope • inside scope read-only

(g) Constraints for the variable role *gatherer*

Focal line	no focal line
Focal scope	loop - * if - *
Usage	• initialization inside scope

(h) Constraints for the variable role *temporary*

Focal line	(1) *var* = *fixed value*
Focal scope	*
Usage	• usage as looping condition or • usage as branch condition

(i) Constraints for the variable role *one-way flag*

Table C.1.: Rule Description for Variable Roles

D. Exercise Descriptions

1	Write a program that displays `Hello World!`.

2	Write a program that displays the standard constants (header files `limits.h` resp. `float.h`) in the following way:
	`CHAR_MIN:`
	`CHAR_MAX:`
	`INT_MIN:`
	`INT_MAX`
	`FLT_MIN:`
	`FLT_MAX:`
	`DBL_MIN:`
	`DBL_MAX:`

3	Write a program that displays all characters that are “greater" or equal to ’A’ or “less" or equal to ’z’
4	Write a program where repeatedly (until input of `<Ctrl>-D` non-negative decimal numbers are read and displayed as octal and hexadecimal each. The output should look like this: `OKT:` *octal value* `HEX:` *hexadecimal value*. The program shall be aborted, if the input is invalid.
5	Write a program that reads a non-negative number and calculates and displays its sum of digits. Example: the sum of digits of *387* is *18*. The output should look like this: `Sum of digits:` *result* If the input is not a valid number, the following message shall be displayed: `Invalid input`
6	Write a program which reads any number of floating point numbers (until `<Ctrl>-D`) and calculates their mean value. The numbers do not all have to be stored at the same time, but can be read and edited one after the other. If the input is not a valid number, the following should be displayed `Input error` Use the format string `%lf` and display the result like this: `Solution:` *result*
7	Write a program that reads two lines as strings, concatenates the second string to the first one, and displays the resulting string. The read string are only allowed to have a length of 50 characters (including terminating null character) at maximum. If this is not the case, the program shall be aborted with displaying the following error message: `Invalid input`

8	Write a program that displays the following text: `Er kam lässig heran und sagte nur "Na, wie geht’s?".  Kommentare beginnen mit /* und enden mit */.  Verwechseln Sie das bitte nicht mit \* bzw.  * \!`

9	Write a program that can read up to 100 integers. The numbers have to be entered separately and confirmed by pressing `Enter`. It should not be possible to enter more than 100 numbers, i.e., after entering number 100 the result shall be displayed immediately. The entered numbers have to be stored in an array. Then the content of the array has to be displayed, the array has to be sorted, and the sorted content of the array has to be displayed again.
	Apply “selection sort": If n numbers have to be sorted, you first determine the smallest element and interchange it with the element on position 1. Then you determine the smallest element of the remaining elements on positions $2, \ldots n$ and interchange it with the element on position 2, and so on.
	The output has to look like this:
	`Numbers to sort:` *number3 number1 number2*
	`Numbers sorted:` *number1 number2 number3*
	At the start of the program, the user shall to be asked to enter the numbers like this:
	`Please enter up to 100 integer numbers:`
	If the input is invalid the following message shall be displayed:
	`Invalid input`

10	Write a program that reads the coefficients of a polynomial $p(c)$ with a maximum degree of 32, starting with the smallest degree. The input of the coefficients shall be ended by entering `<CTRL>-D`. After that the user has to be asked to enter places (floating numbers) for which the polynomial shall be calculated: `Please enter places for calculations` These places a shall be read until end of input and the value of $p(a)$ has to be calculated and displayed each time immediately after input like this: `Value of the polynomial at place` *place*`:` *value* Use the format string `%g` to output the result. For the calculation you can use the Horner scheme. Example: $p(x) = 5.1x^3 - 1.8x^2 - 0.02x + 17.3 = 17.3 + x(-0.02 + x(-1.8 + 5.1x))$
11	Write a program that reads a binary number and displays the corresponding decimal value like this: `Decimal:` *value* If the input is invalid, the program shall be aborted with the following message: `Invalid input`
12	Write a program that reads a text from standard input and counts how often each ASCII character occurs. Display the characters in the order of their ASCII code (ascending order) and use a new line for each character. Graphical characters shall be displayed like this: *character* `:` *count of character* Control characters shall be displayed like this: `<CTRL>` *number of character* `:` *count of character*

13	Write a program that reads a string (that may also contain blanks), determines if the string is a palindrome, and accordingly displays `palindrome` or `no palindrome`. It shall not be distinguished between upper case and lower case letters. A palindrome is a string that is the same read backwards and forwards. Example: `smart trams`
14	Write a program that determines all prime numbers less than 1000. There exist two approaches: 1. It is checked by division if the numbers n ($1 < n < 1000$) have a factor k ($1 < k < \sqrt{n}$). 2. With the "sieve of Eratosthenes" (Eratosthenes of Cyrene, Greek mathematician around 225 BC) all numbers to be examined (here 2 to 999) are first written down. Each step of the actual algorithm consists of three individual steps: a) The first not deleted number is searched for. b) This number is noted as the prime number. c) Its multiples are deleted. Implement both algorithms as efficiently as possible. Output the numbers separated by a blank character. For the submission, it is sufficient to upload one of the two solutions.

15	Write a function
	`int date2int(int day, int month)`
	which converts a date into the day in the year (e.g, the 11.2. is the 42nd day in the year) and a function
	`void int2day(int days, int* result)`
	which converts a day of the year into a date. The result is an array where the first element corresponds to the day and the second element corresponds to the month. Assume that February always has 28 days.

16	A permutation of the length n is a mapping of the set $\{1,2,...,n\}$ on itself. Program and test a function `int permtest( const unsigned int* p, int length)` which tests a passed array of length `length` to see if the content represents a permutation and in this case returns the value `1` and `0` otherwise.

17	Implement the following functions:
	`int strend(const cgar *s, const char *t)`
	returns `1` if string `t` is at the end of string `s`, and `0` otherwise.
	`char *strchr(const char *s, int c)`
	returns a pointer to the last occurrence of the character `c` in string `s` or the `NULL` pointer if `c` does not occur in `s`.
	`char *strstr(const char *s, const char *t)`
	returns a pointer to the first occurrence of the string `t` in string `s` or the `NULL` pointer if `t` does not occur in `s`.

18	Call the command `factor 144` in the console. The output
	`144:  2 2 2 2 3 3`
	gives the prime number decomposition of 144. Implement this in C to be able to process any integer up to the size `UINT_MAX`. The number to be processed is passed to the program as argument by command line. Write a main function for the prototype
	`int main(int argc, char *argv[])`
	If the number of passed arguments is invalid, display the following error message
	`Wrong number of arguments`

19	Solve exercise 17 again without using the `[]`-operator, i.e. only using pointer arithmetic.

20	Implement the following functions:
	`char *strinv(const char *s)`
	returns the inverse of the string `s`, i.e. the string `s` read backwards.
	`char *strconcat(const char *s, const char *t)`
	returns a string that is a union of the strings `s` and `t`, i.e. corresponds to the concatenation of the strings `s` and `t`.

21	A permutation matrix is a quadratic, integer matrix where exactly one element is `1` in each row and each column and the other elements are all `0`. Implement a function `int permatcheck(const int** p, int length)` which tests whether a two-dimensional i permutation matrix is involved by means of a two-dimensional integer array of size $length \times length$ is a permutation matrix. Return `1`, if yes, and `0` otherwise.
22	A Sudoku matrix is a 9×9 matrix whose elements are integers between 1 and 9 and which fulfill further conditions. To formulate the conditions we consider the matrix as a block matrix of 9 blocks of 3×3 matrices formed by the rows resp. columns 1 to 3, 4 to 6 and 7 to 9. Each of these 3×3 matrices is referred to as a block. Now we can formulate the conditions that make a Sudoku matrix out of a matrix of the above form: • Each row and each column is a permutation of length 9. • Each block contains each of the values 1 to 9 exactly once. Implement a function `int sudokucheck(const int** s)` which tests whether a matrix described by a two-dimensional integer array of size 9×9 is a Sudoku matrix. Return `1`, if yes, and `0` otherwise.

23	Define the type `fraction_t` as a structure with two integer components `numerator` and `denominator`. Implement the following standard operations as functions for this type: `void fractInput(fraction_t* fract)`(Input a fraction) `void fractOutput(fraction_t* fract)`(Display a fraction) `void fractAddition(fraction_t* fract, fraction_t add)` (Addition of two fractions. The second fraction is added to the first fraction) `void fractSubtraction(fraction_t* fract, fraction_t sub)` (Subtraction of two fractions. The second fracture is subtracted from the first) `void fractMultiplication(fraction_t* fract, fraction_t mult)` (Multiplication of two fractions. The second fraction is multiplied to the first fraction) `void fractDivision(fraction_t* fract, fraction_t div)` (Division of two fractions. The second fraction is divided away from the first fraction) `float fractQuotient(fraction_t fract)` (Calculation of the quotient) `void fractExtend(fraction_t* fract, int factor)` (Extend by a factor) `void fractCancel(fraction_t* fract)` (Cancel a fraction) Cancel the results of addition/subtraction/multiplication/division as much as possible. Always enter integer numbers as a whole, i.e. 1/1, 2/1, and so on.
24	Write a function that implements selection sort on a (sub) array of integers using the following prototype: `int (int *from, int *to)` `from` is a pointer to the first element and `to` a pointer which points to element after the last element of the (sub) array to be sorted. The function returns the number of re-stored values. Only re-store the values if necessary. Only use pointer arithmetic in the function and not the `[]`-operator.
25	Write a recursive function with the prototype `int sum(int n)` that implements the following definition: $sum(n) = \begin{cases} 0 & \text{if } n = 0 \\ sum(n-1)+n & \text{if } n > 0 \\ sum(n+1)+n & \text{if } n < 0 \end{cases}$
26	Write a recursive function `unsigned long ggT(unsigned long a, unsigned long b)` which calculates the largest common divisor of the non-negative integers `a` and `b`. Use the formula $ggt(a,b) = \begin{cases} ggt(b, a\ mod\ b) & \text{if } b > 0 \\ a & \text{if } b = 0 \end{cases}$
27	For an given year number, we want to determine whether it is a leap year or not. Implement a function `int leapyear(int year)` which returns the value `0` if the given year is not a leap year and `1` otherwise. The Gregorian calendar specifies that each year dividable by 4 is a leap year unless the year is dividable by 100. In this case, it is only a leap year if the year is dividable by 400.

28	For the functions of exercise 15, also also exotic entries. Examples: The 366th day in 1991 is 1.1.1992; the 0th day in 1992 is 31.12.1991; and so on. The year number is irrelevant.
29	Use the `sizeof`-operator to determine how many bytes are reserved for values of integer types. The output should look like this: `char:` *size* `short:` *size* `int:` *size* `long:` *size*

30	Modify your solution of exercise 29 such that the year is also taken into account. Supplement the parameters of the two functions:
	`int datum2int(int day, int month, int year)`
	`void int2datum(int days, int year, int* result)`
	The result field should now contain three elements, the third of which corresponds to the year.

E. Complete Case Study Results

E.1. Similarity Results

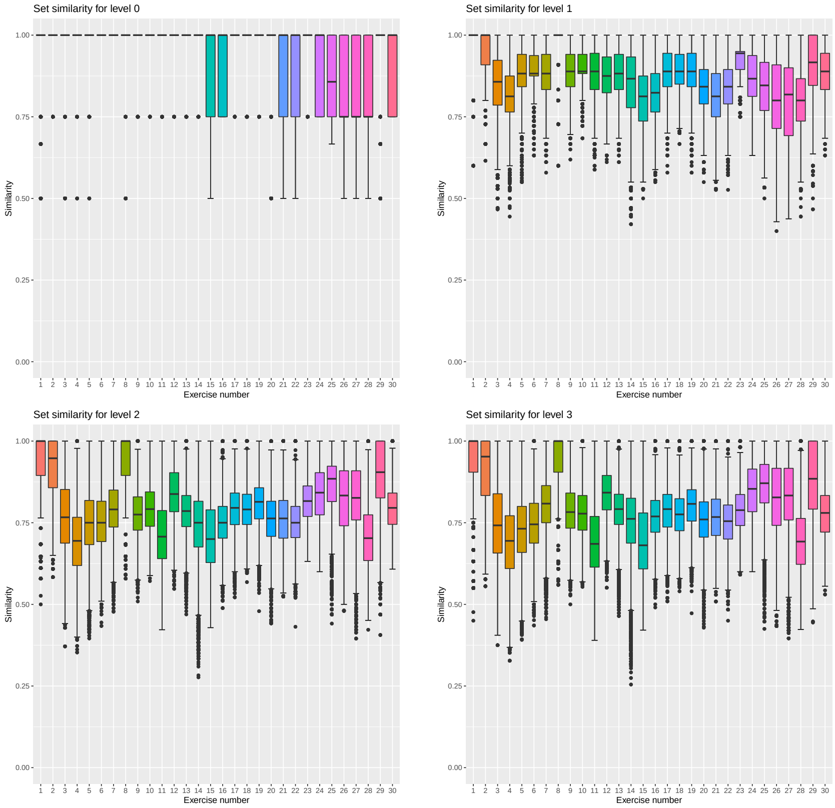


Figure E.1.: Set similarity

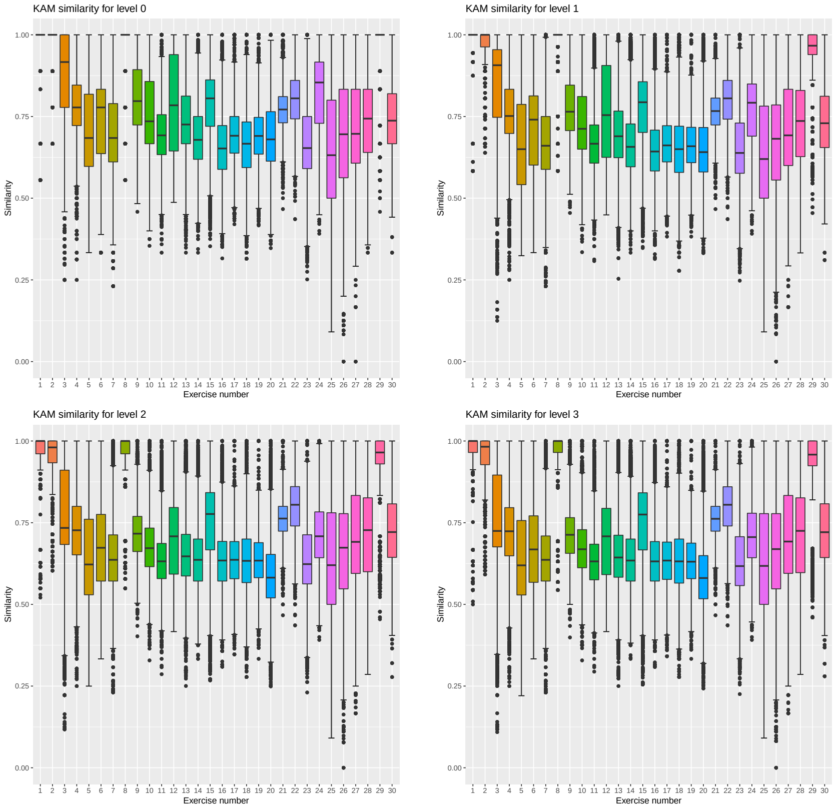


Figure E.2.: KAM similarity

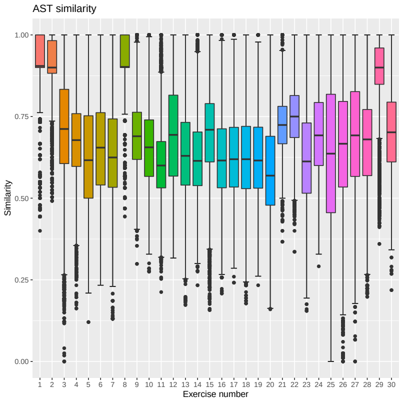


Figure E.3.: AST similarity

E.2. Mata-parameters Results

		AUC		RMSE	
meta-parameter	value	PFA	AFM	PFA	AFM
KC level	0	.577	**.653**	**.499**	**.567**
	1	**.618**	.634	.507	.586
	2	.586	.581	.580	.629
	3	.581	.578	.588	.633
minimum steps	0	**.597**	**.622**	**.535**	**.530**
	5	.592	.620	**.535**	.574
	10	.593	.617	.539	.585
	15	.590	.609	.536	.598
	20	.580	.591	.564	.640
step definition	first	.568	.584	.570	.585
	last	.580	.595	.584	.737
	every	**.623**	**.656**	**.476**	**.490**
incorrect KCs	all	**.592**	.611	**.538**	**.597**
	diff	.589	**.613**	.549	.610
KC count	binary	.580	.600	.556	.616
	multiple	**.601**	**.623**	**.531**	**.591**
Q matrix	all	.557	.610	.546	.605
	shared	**.605**	**.628**	.541	**.588**
	union	.594	.618	.542	.592
	common	.600	.619	**.539**	.595
	used	.589	.592	.540	.629
	set	.596	.613	.552	.609
	KAM	.592	.601	.544	**.609**

Table E.1.: Mean and median values of AUC and RMSE all meta-parameter values

KC count	binary
multiple	+0.588

(a) AUC PFA

KC count	binary
multiple	+0.481

(b) AUC AFM

KC count	binary
multiple	-0.554

(c) RMSE PFA

KC count	binary
multiple	-0.375

(d) RMSE AFM

Table E.2.: Effect sizes for meta-parameter *KC count*

level	0	1	2
1	+1.145	–	–
2	not sign.	-0.968	–
3	not sign.	-1.207	-0.184

(a) AUC PFA

level	0	1	2
1	-0.725	–	–
2	-1.423	-1.133	–
3	-1.526	-1.241	-0.167

(b) AUC AFM

level	0	1	2
1	not sign.	–	–
2	+1.303	+1.468	–
3	+1.464	+1.716	+0.248

(c) RMSE PFA

level	0	1	2
1	+0.447	–	–
2	+0.727	+0.554	–
3	+0.738	+0.563	+0.078

(d) RMSE AFM

Table E.3.: Effect sizes for parameter *level*

step def.	first	last
last	+0.413	–
every	+1.168	+1.071

(a) AUC PFA

step def.	first	last
last	+0.301	–
every	+1.150	+1.684

(b) AUC AFM

step def.	first	last
last	+0.291	–
every	-1.485	-1.890

(c) RMSE PFA

step def.	first	last
last	+1.691	–
every	-0.964	-2.413

(d) AUC AFM

Table E.4.: Effect sizes for parameter *step definition*

incorrect KCs	all
diff	+0.365

(a) RMSE PFA

Table E.5.: Effect sizes for meta-parameter *incorrect KCs*

minimum steps	0	5	10	15
5	-0.291	–	–	–
10	-0.208	not sign.	–	–
15	-0.315	-0.108	-0.202	–
20	-0.650	-0.461	-0.663	-0.509

(a) AUC PFA

minimum steps	0	5	10	15
5	not sign.	–	–	–
10	-0.211	not sign.	–	–
15	-0.347	-0.262	-0.231	–
20	-0.794	-0.698	-0.729	-0.696

(b) AUC AFM

minimum steps	0	5	10	15
5	not sign.	–	–	–
10	not sign.	not sign.	–	–
15	+0.248	not sign.	+0.281	–
20	+0.638	+0.548	+0.726	+0.620

(c) RMSE PFA

minimum steps	0	5	10	15
5	+0.296	–	–	–
10	+0.509	+0.332	–	–
15	+0.664	+0.522	+0.414	–
20	+0.802	+0.704	+0.586	+0.286

(d) RMSE AFM

Table E.6.: Effect sizes for parameter *minimum steps*

Q matrix	all	shared	union	common	set	KAM
shared	+0.761	–	–	–	–	–
union	+0.690	-0.359	–	–	–	–
common	+0.750	-0.191	+0.359	–	–	–
set	+0.667	-0.248	not sign.	not sign.	–	–
KAM	+0.737	-0.374	not sign.	-0.240	not sign.	–
used	+0.687	-0.425	not sign.	-0.316	-0.277	-0.170

(a) AUC PFA

Q matrix	all	shared	union	common	set	KAM
shared	+0.512	–	–	–	–	–
union	+0.444	-0.367	–	–	–	–
common	+0.441	-0.380	not sign.	–	–	–
set	not sign.	-0.255	not sign.	not sign.	–	–
KAM	not sign.	-0.517	-0.358	-0.402	-0.484	–
used	-0.165	-0.548	-0.412	-0.462	-0.393	not sign.

(b) AUC AFM

Q matrix	all	shared	union	common	set	KAM
shared	not sign.	–	–	–	–	–
union	not sign.	not sign.	–	–	–	–
common	-0.200	not sign.	not sign.	–	–	–
set	not sign.	not sign.	+0.225	+0.300	–	–
KAM	not sign.	not sign.	not sign.	not sign.	-0.343	–
used	-0.190	not sign.	not sign.	not sign.	-0.367	not sign.

(c) RMSE PFA

Q matrix	all	shared	union	common	set	KAM
shared	-0.397	–	–	–	–	–
union	-0.341	+0.250	–	–	–	–
common	-0.362	+0.317	not sign.	–	–	–
set	not sign.	+0.187	not sign.	not sign.	–	–
KAM	not sign.	+0.249	not sign.	not sign.	not sign.	–
used	not sign.	+0.307	+0.277	+0.243	+0.204	not sign.

(d) RMSE AFM

Table E.7.: Effect sizes for meta-parameter *Q matrix*

E.3. Error Landscape Results

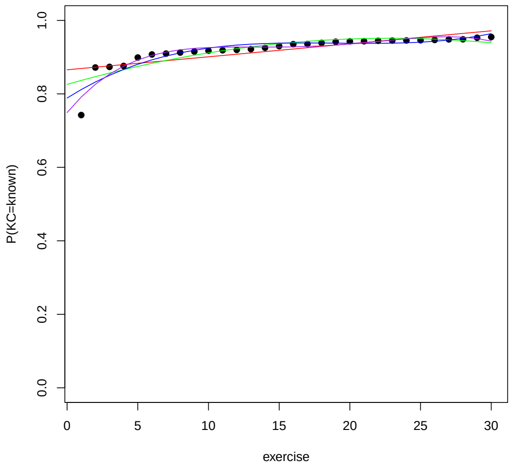
	error type	s_t	f_t	d_t	r_t	synt.	conc.	strat.	slop.	misint.	dom.
e1	wrong output format	4622.64	1193	3.87	0.92				X	X	
e2	missing semicolon	1416.09	319	4.44	0.81	X			X		
e3	missing check for invalid input	1266.24	235	5.39	0.73			X			
e4	undeclared variable	1142.37	182	6.28	0.76	X			X		
e5	confuse EOF and '\n'	1078.74	189	5.71	0.53		X			X	
e6	wrong array size	978.49	172	5.69	0.52			X		X	X
e7	off-by-one-error	856.08	173	4.95	0.67		X	X	X		
e8	wrong boundaries	843.39	182	4.63	0.6			X		X	
e9	unexpected output	809.88	185	4.38	0.6				X		
e10	wrong escaping	705.9	144	4.9	0.42	X					
e11	boundary case omitted	485.57	77	6.31	0.57			X			
e12	no check for array limits during input	481.57	74	6.51	0.63			X			
e13	missing terminating character	398.85	109	3.66	1				X		
e14	missing error output	392.49	89	4.41	0.62				X		
e15	missing/wrong include	366.84	100	3.67	0.9		X		X		
e16	uninitialized variable	348.25	64	5.44	0.57		X		X		
e17	order of conditions	345.6	52	6.65	0.69		X				
e18	wrong claculation	338.81	65	5.21	0.52			X			X
e19	wrong type	331.02	55	6.02	0.44		X				
e20	missing subgoal	328.19	63	5.21	0.49			X			
e21	unnecessary if	322.68	63	5.12	0.51			X		X	
e22	missing/wrong pointer de-reference	283.6	65	4.36	0.79		X				
e23	missing loop	282.12	49	5.76	0.48			X		X	
e24	misconception of input buffer	280.29	53	5.29	0.59		X	X			
e25	conflicting/incompatible types	249.33	66	3.78	0.67		X				
e26	misunderstanding of exercise	164.25	31	5.3	0.58					X	
e27	confusing last index with size in array declaration	155.26	32	4.85	0.72		X				
e28	wrong format specifier	155.1	33	4.7	0.65	X					
e29	misplacement of output	151.14	23	6.57	0.64		X			X	
e30	de-referncing something that is not a pointer	137.17	34	4.03	1		X				
e31	variable-sized object may not be initialized	116.71	31	3.76	0.65	X	X				
e32	missing reset of variable in loop	113.75	14	8.12	0.75		X		X		

	error type	s_t	f_t	d_t	r_t	synt.	conc.	strat.	slop.	misint.	dom.
e33	missing quotes for string/character	110.13	28	3.93	0.67	X					
e34	wrong initialization value	109.67	14	7.83	0.62			X			X
e35	spurious constraint	102	17	6	0.42					X	
e36	spurious code fragment	100.69	29	3.47	0.5		X		X		
e37	using = instead of ==	98.33	10	9.83	0.67	X			X		
e38	misconception of return value of scanf	98	20	4.9	0.5		X				
e39	spelling mistake/typo	88	11	8	0.38				X		
e40	missing condition	86.23	19	4.54	0.46			X		X	
e41	storing input values in array instead of continous calculation	78.2	17	4.6	0.7				X	X	
e42	infinite loop	75.85	17	4.46	0.31		X		X		
e43	missing output	69.33	13	5.33	0.44				X		
e44	using getchar instead of scanf	66	9	7.33	0.5		X				
e45	array out of bounds	64.91	17	3.82	0.36		X				
e46	missing escaping in string	59.71	19	3.14	0.36	X			X		
e47	misconception of checking a type	57.6	18	3.2	0.7		X				
e48	unexpected read-in	51	9	5.67	0.5				X		
e49	division by zero	49	7	7	0.75			X			
e50	using isgraph() instead of isprint()	48.57	10	4.86	0.43		X				
e51	error in include	40.5	15	2.7	0.8	X			X		
e52	missing read-in	38.5	7	5.5	0.4				X		
e53	missing malloc	38	4	9.5	1		X				
e54	wrong domain knowledge	36.67	11	3.33	0.22						X
e55	assumption that condition is coninuesly checked	30	2	15	1		X				
e56	wrong array element	28.67	10	2.87	0.8			X			
e57	nesting loop instead of additional condition	26.67	4	6.67	0.33		X				
e58	wrong assignment value	26.67	5	5.33	0.67			X			
e59	array starting with index 1	25.67	7	3.67	1		X				
e60	using & wrong	23	2	11.5	0		X				
e61	misplacement of increment	22.75	7	3.25	0.75			X			
e62	Using ’ ’ instead of " " for strings	22	8	2.75	1	X					
e63	not using in scanf	21	2	10.5	0	X	X		X		
e64	unnecessary break	20	2	10	1		X	X			
e65	wrong interpretation of sizeof-operator	18.67	8	2.33	0.33		X				
e66	copy paste error	18	3	6	0.5				X		
e67	wong calculation of array length	18	2	9	0			X			

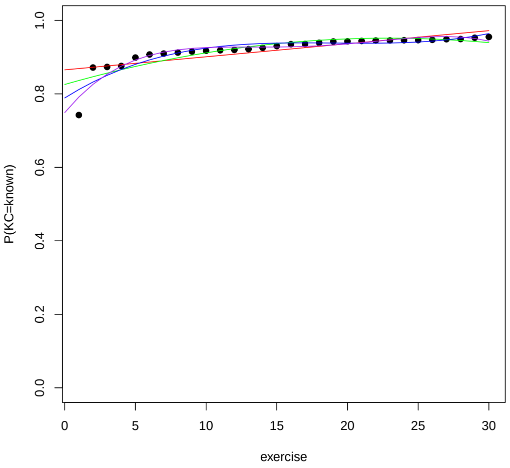
	error type	s_t	f_t	d_t	r_t	synt.	conc.	strat.	slop.	misint.	dom.
e68	wrong alloc size	18	2	9	1			X			
e69	miscplacement of check	17	4	4.25	0			X			
e70	missing else	16.33	7	2.33	0.67			X			
e71	spurious main function	14	7	2	0.75				X		
e72	wrong condition	14	7	2	0.4			X		X	X
e73	== instead of !=	13.5	3	4.5	0.5			X	X		
e74	assumption that array ends with ’\0’ although not string	12	2	6	1		X				
e75	missing if	12	4	3	1			X			
e76	return with value in void-function	12	2	6	1		X				
e77	misconception of order of evaluation	11.67	5	2.33	0.67		X				
e78	storing result in local variable	10.5	3	3.5	0.5		X				
e79	wrong output	10.5	6	1.75	0.5					X	
e80	wrong return value	10.5	3	3.5	0.5			X			
e81	missing braces	10	4	2.5	1	X			X		
e82	instead off ==	8	2	4	1			X			
e83	wrong integer constant used	8	2	4	0			X			X
e84	trying to print array directly	6	2	3	0		X				
e85	type missing in function declaration	6	2	3	1	X					
e86	Using ‖ instead of	6	3	2	0.5		X				
e87	wrong function for display output	6	6	1	1		X				
e88	setting of pointer value wrong	5	1	5	0		X	X			
e89	unnecessary loop	5	1	5	0			X		X	
e90	misconception of recursion	4	2	2	1		X				
e91	mixing up bounds	4	2	2	1			X			
e92	multiple comparison	4	2	2	1	X	X				
e93	parameter missing	4	2	2	0				X		
e94	wrong check for type	4	2	2	1	X	X				
e95	= instead of +=	3	1	3	0			X	X		
e96	misplacement of statement	3	1	3	0		X	X			
e97	printf instead of return	3	3	1	0.5		X			X	
e98	wrong return type	3	1	3	0				X		
e99	do-while instead of while	2	2	1	1		X	X			
e100	misconception of return	2	2	1	1		X				
e101	misplacement of braces	2	2	1	1				X		
e102	missing cases	2	2	1	1			X			

	error type	s_t	f_t	d_t	r_t	synt.	conc.	strat.	slop.	misint.	dom.
e103	mixing up looping variables	2	2	1	1			X	X		
e104	wrong variable	2	1	2	0			X			
e105	misplacement of variable reset	1	1	1	0		X	X			

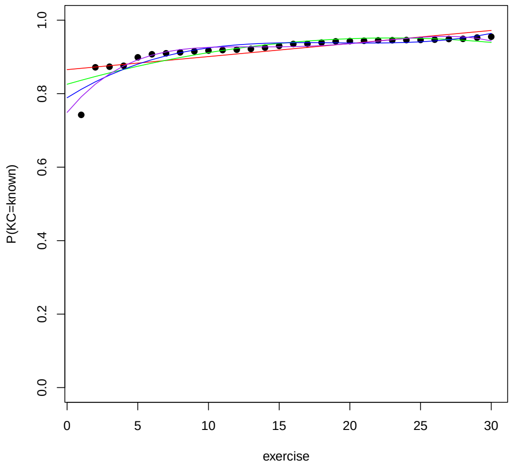
E.4. Learning Curves



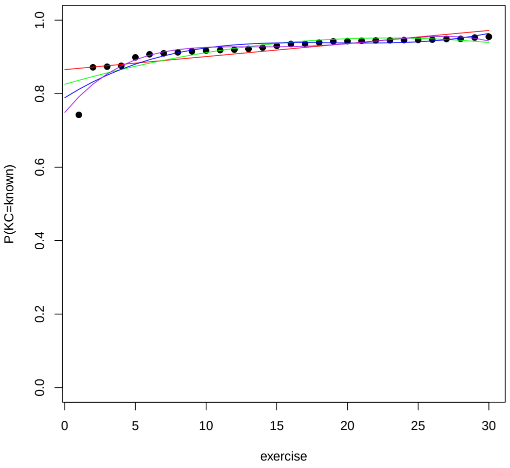
(a) KC *block* based on DBN



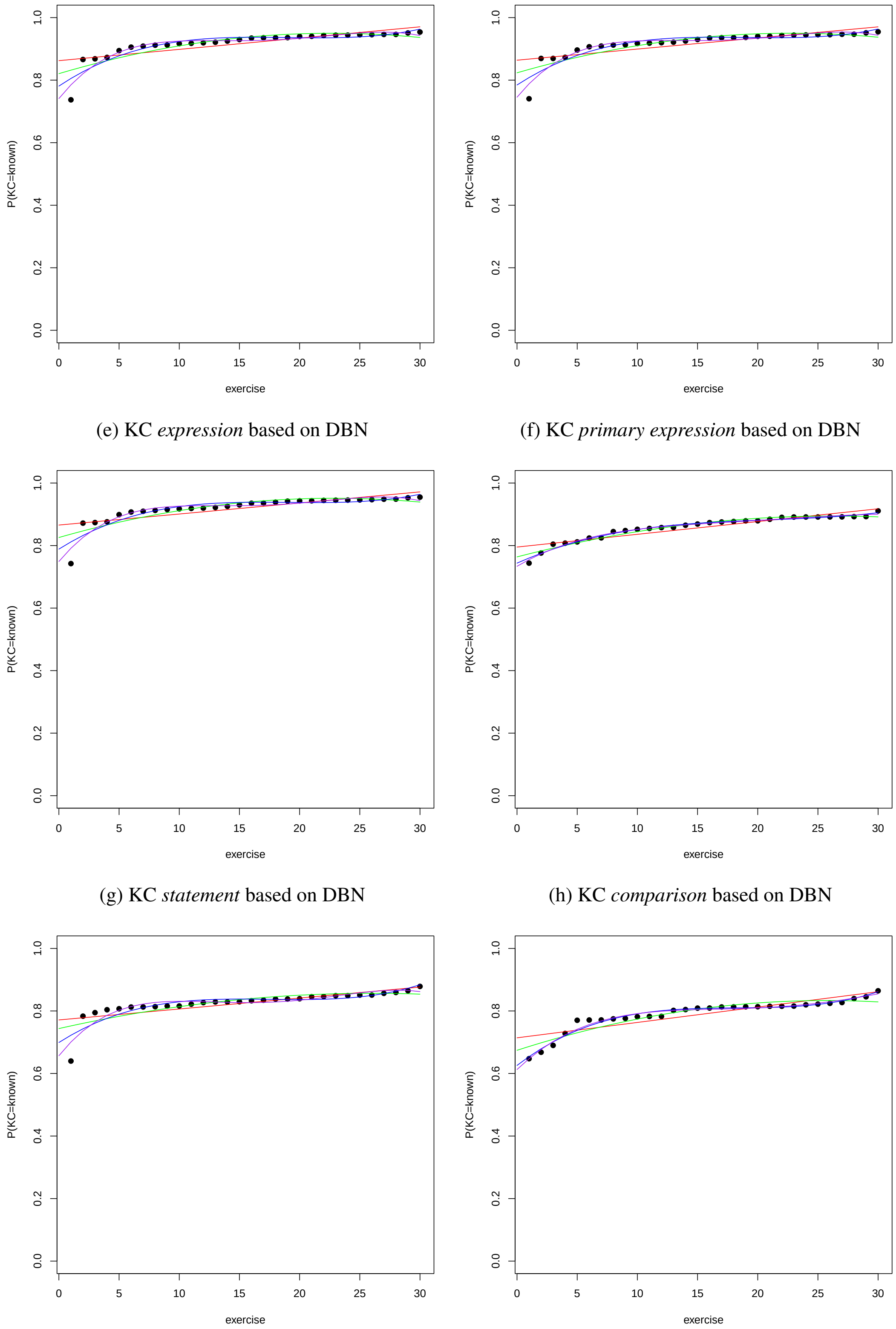
(b) KC *data type* based on DBN



(c) KC *declaration* based on DBN



(d) KC *declarator* based on DBN



(e) KC *expression* based on DBN

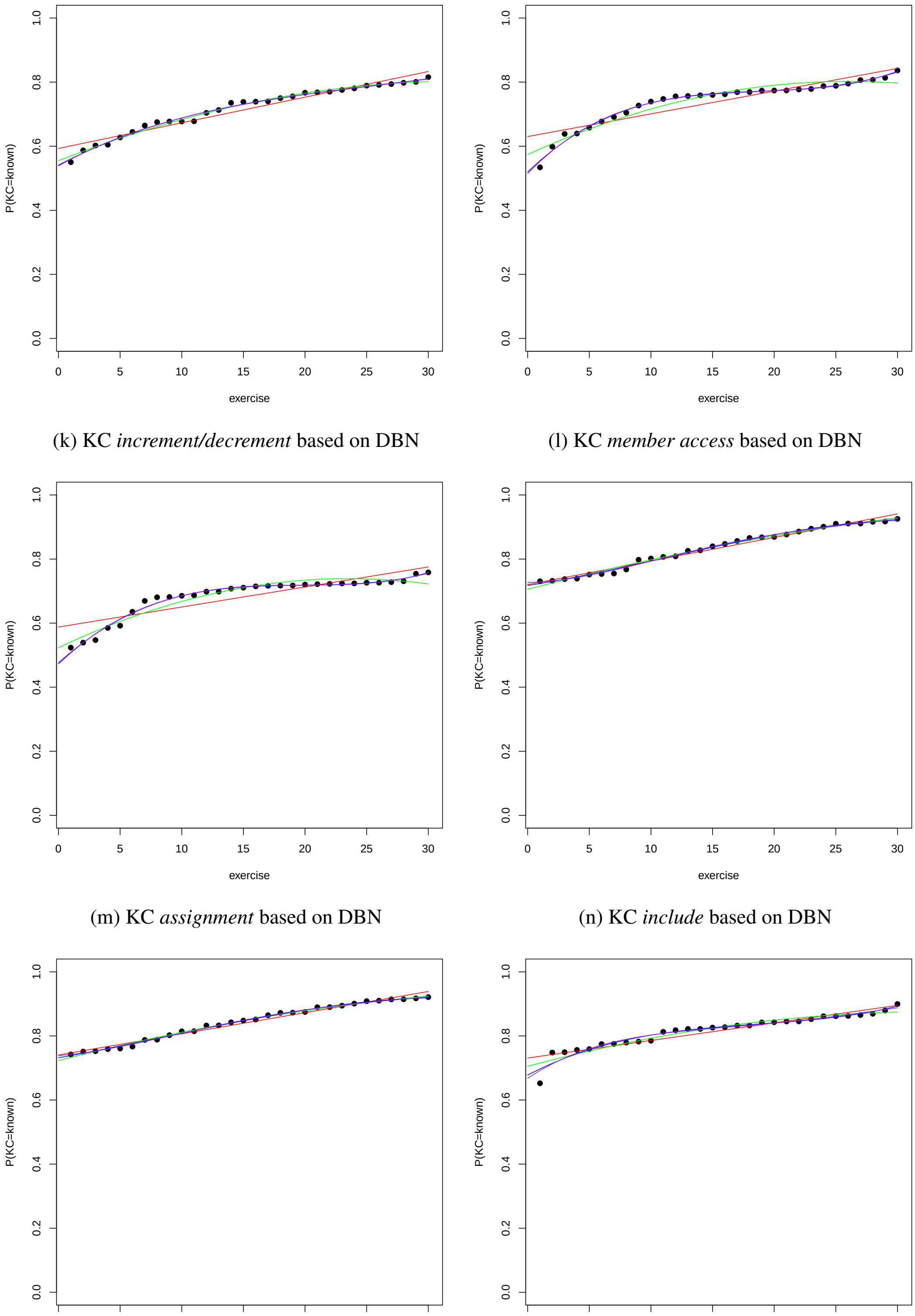
(f) KC *primary expression* based on DBN

(g) KC *statement* based on DBN

(h) KC *comparison* based on DBN

(i) KC *jump* based on DBN

(j) KC *iteration* based on DBN



(k) KC *increment/decrement* based on DBN

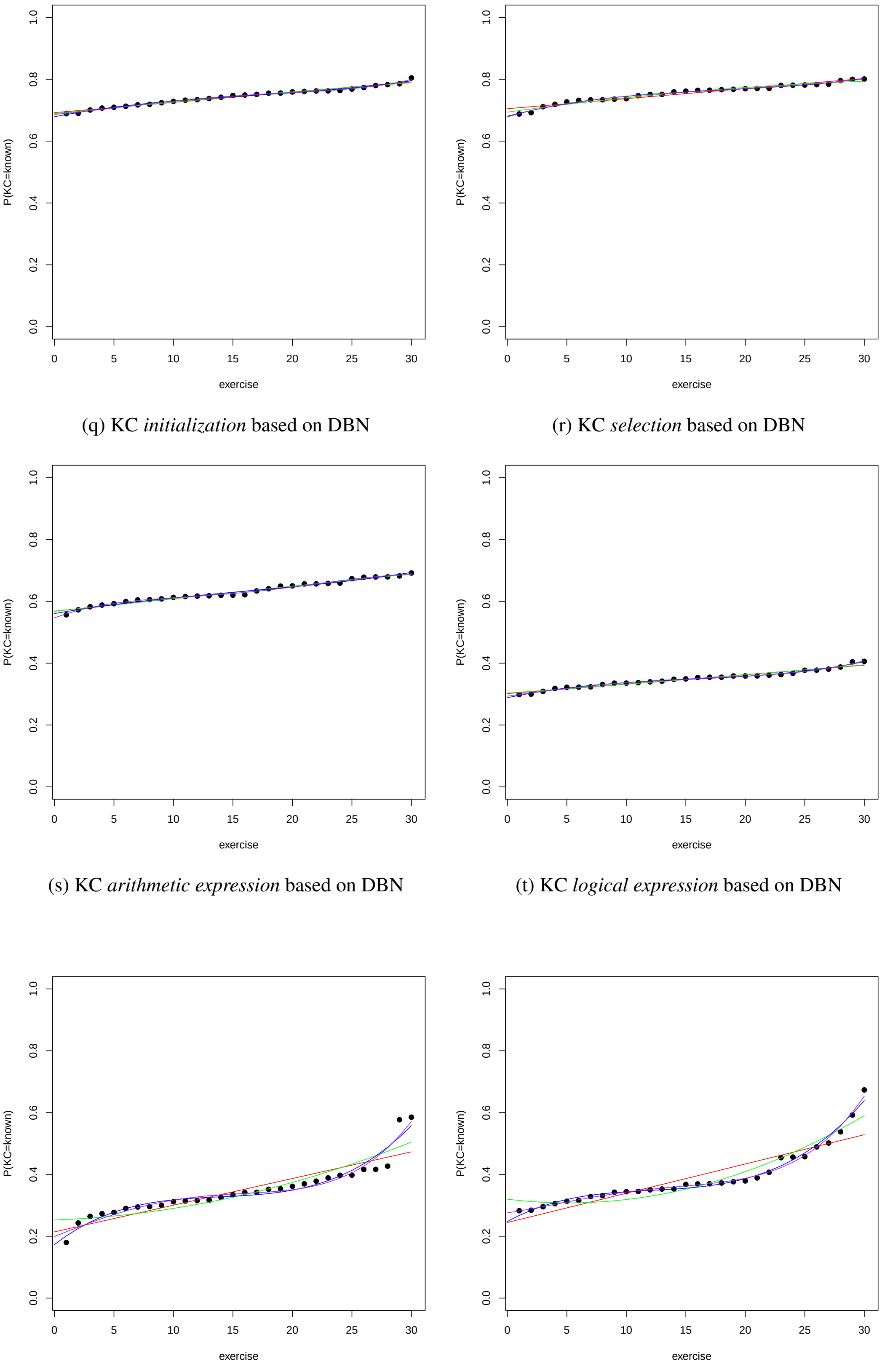
(l) KC *member access* based on DBN

(m) KC *assignment* based on DBN

(n) KC *include* based on DBN

(o) KC *preprocessor* based on DBN

(p) KC *function call* based on DBN



(q) KC *initialization* based on DBN

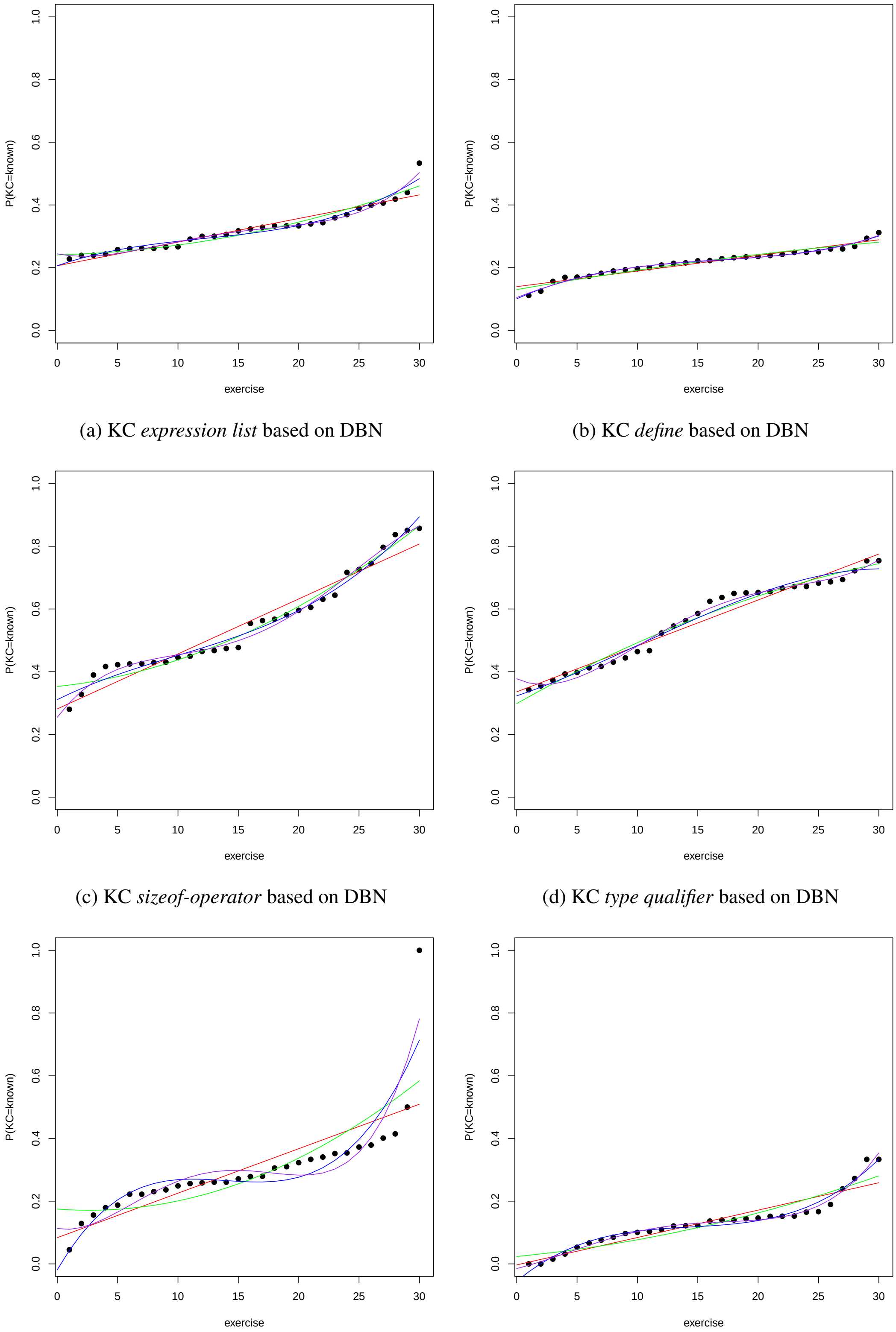
(r) KC *selection* based on DBN

(s) KC *arithmetic expression* based on DBN

(t) KC *logical expression* based on DBN

(u) KC *label* based on DBN

(v) KC *type cast* based on DBN



(a) KC *expression list* based on DBN

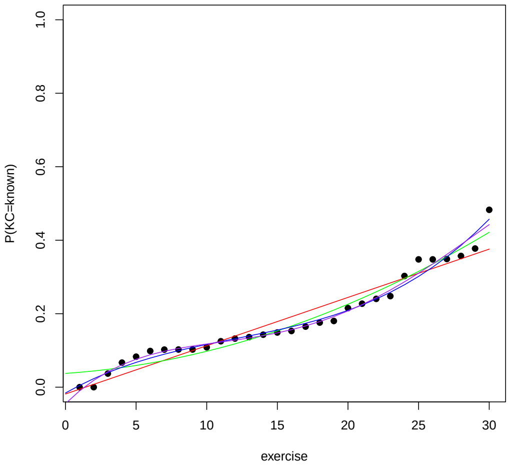
(b) KC *define* based on DBN

(c) KC *sizeof-operator* based on DBN

(d) KC *type qualifier* based on DBN

(e) KC *storage class* based on DBN

(f) KC *bitwise operator* based on DBN



(g) KC *conditional operator* based on DBN

Figure E.5.: Learning curves of syntactic KCs

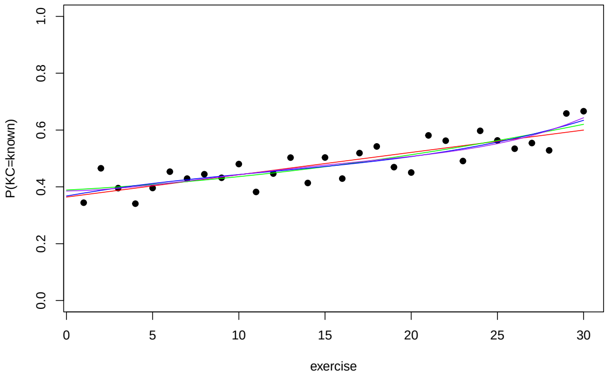
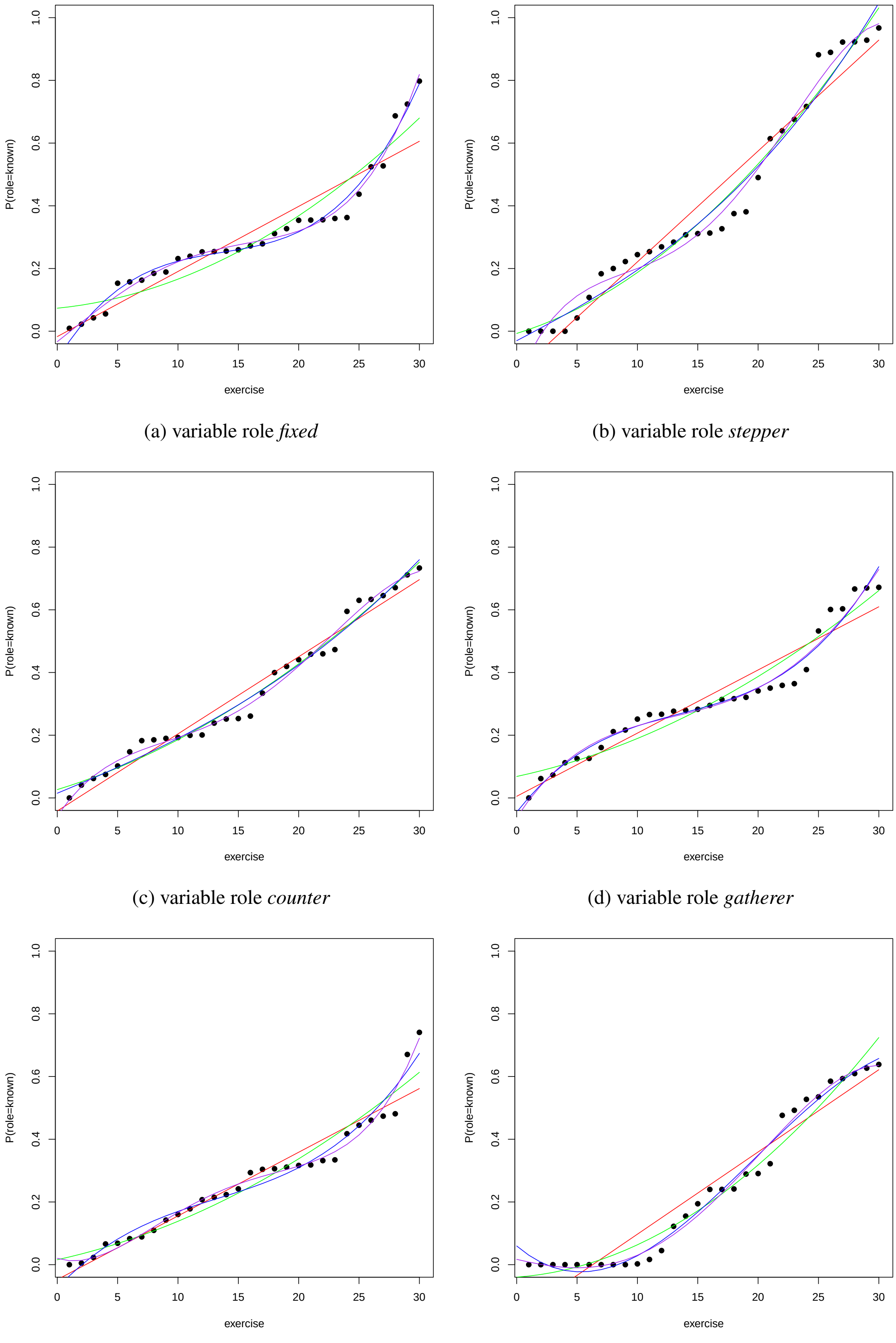


Figure E.6.: Learning curve of the problem solving ability



(a) variable role *fixed*

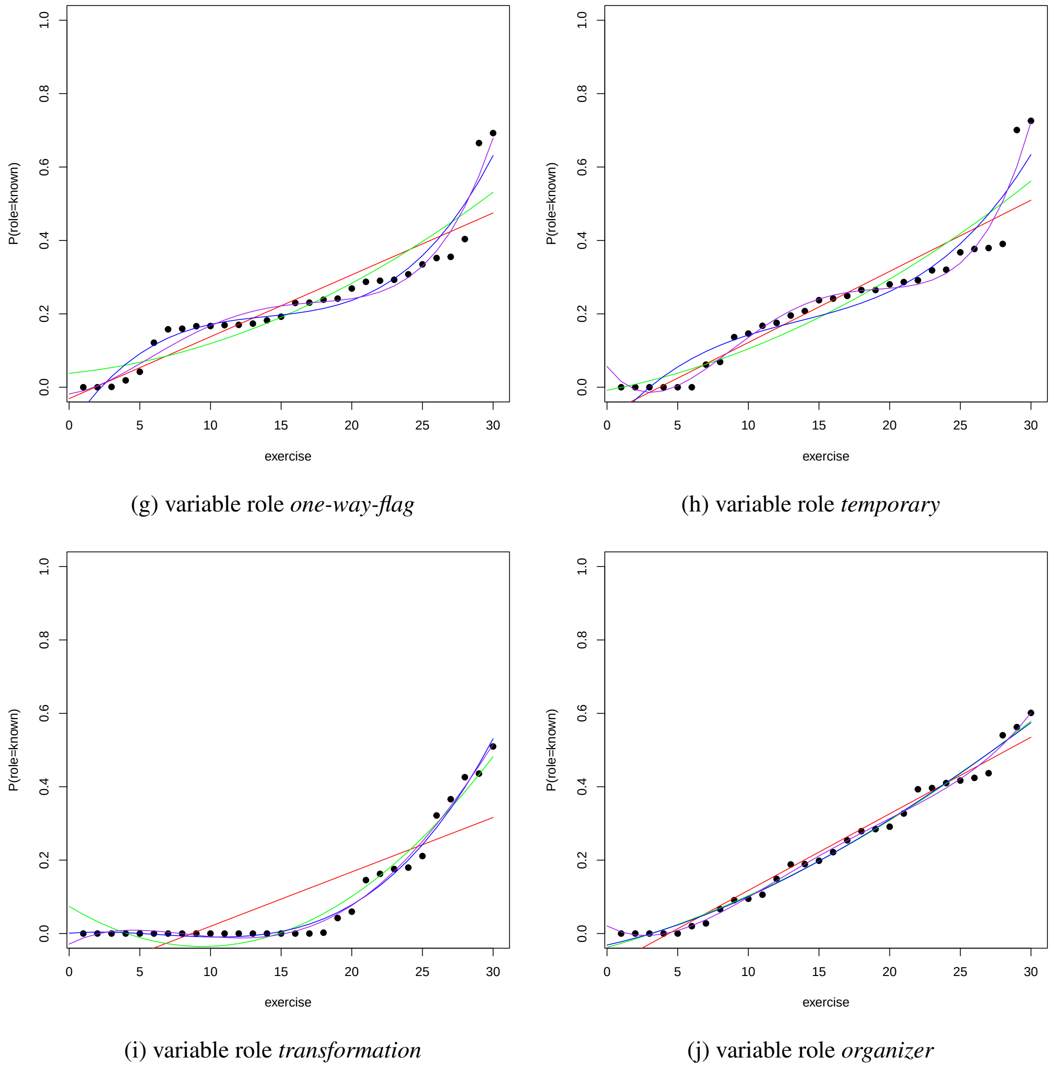
(b) variable role *stepper*

(c) variable role *counter*

(d) variable role *gatherer*

(e) variable role *MRH*

(f) variable role *MWH*



(g) variable role *one-way-flag*

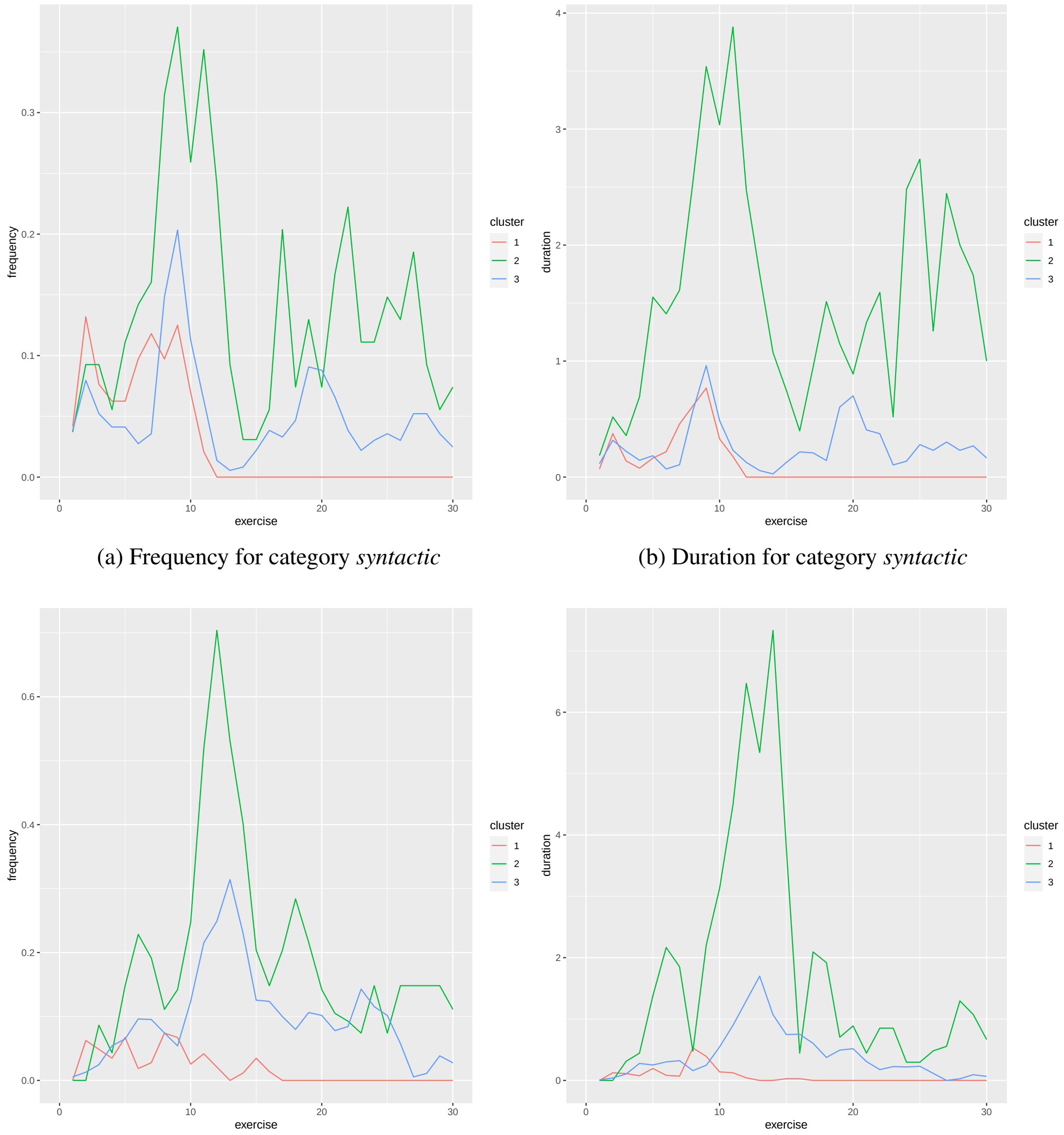
(h) variable role *temporary*

(i) variable role *transformation*

(j) variable role *organizer*

Figure E.7.: Learning curves of variable roles

E.5. Clustering Results

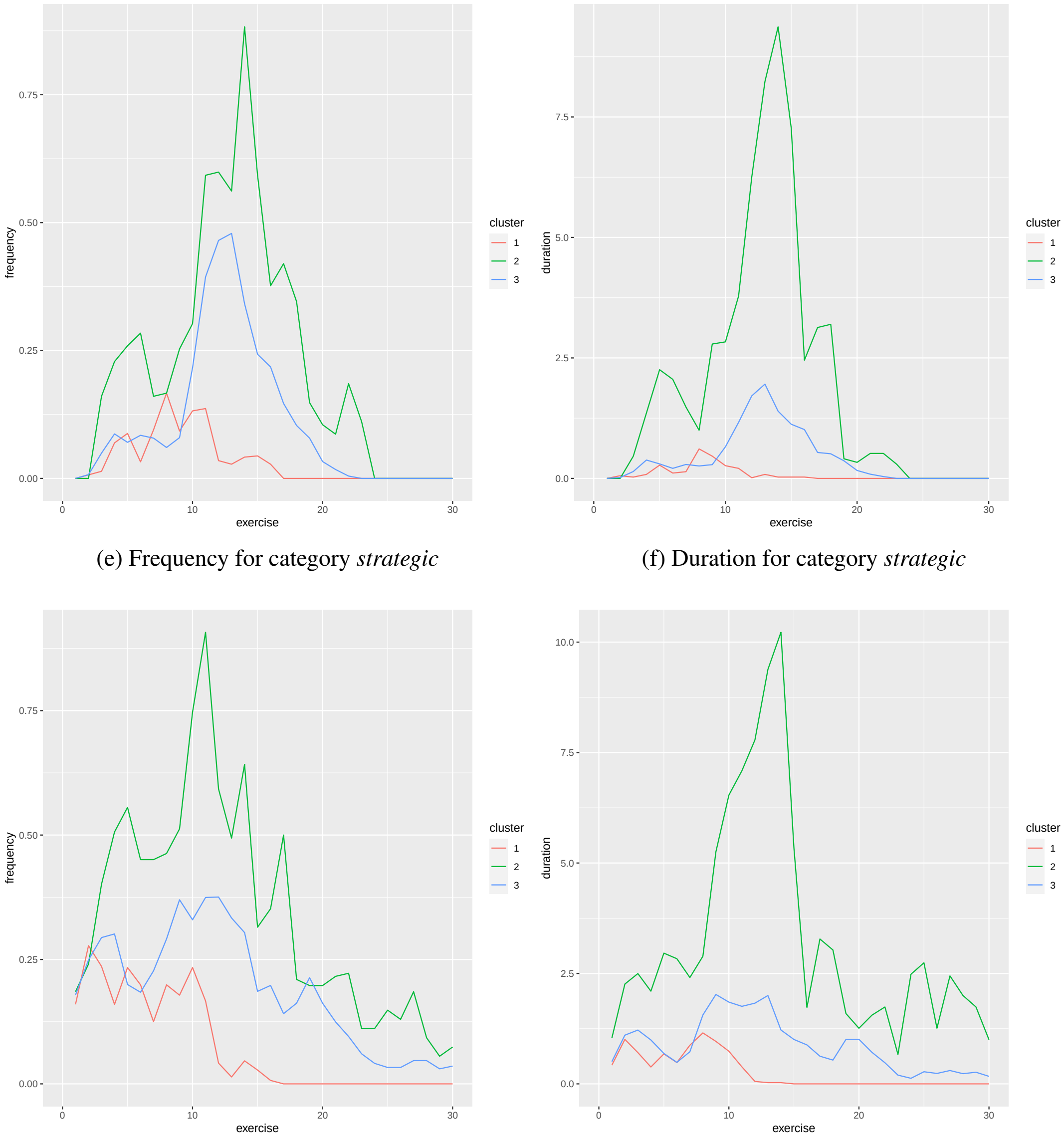


(a) Frequency for category *syntactic*

(b) Duration for category *syntactic*

(c) Frequency for category *conceptual*

(d) Duration for category *conceptual*

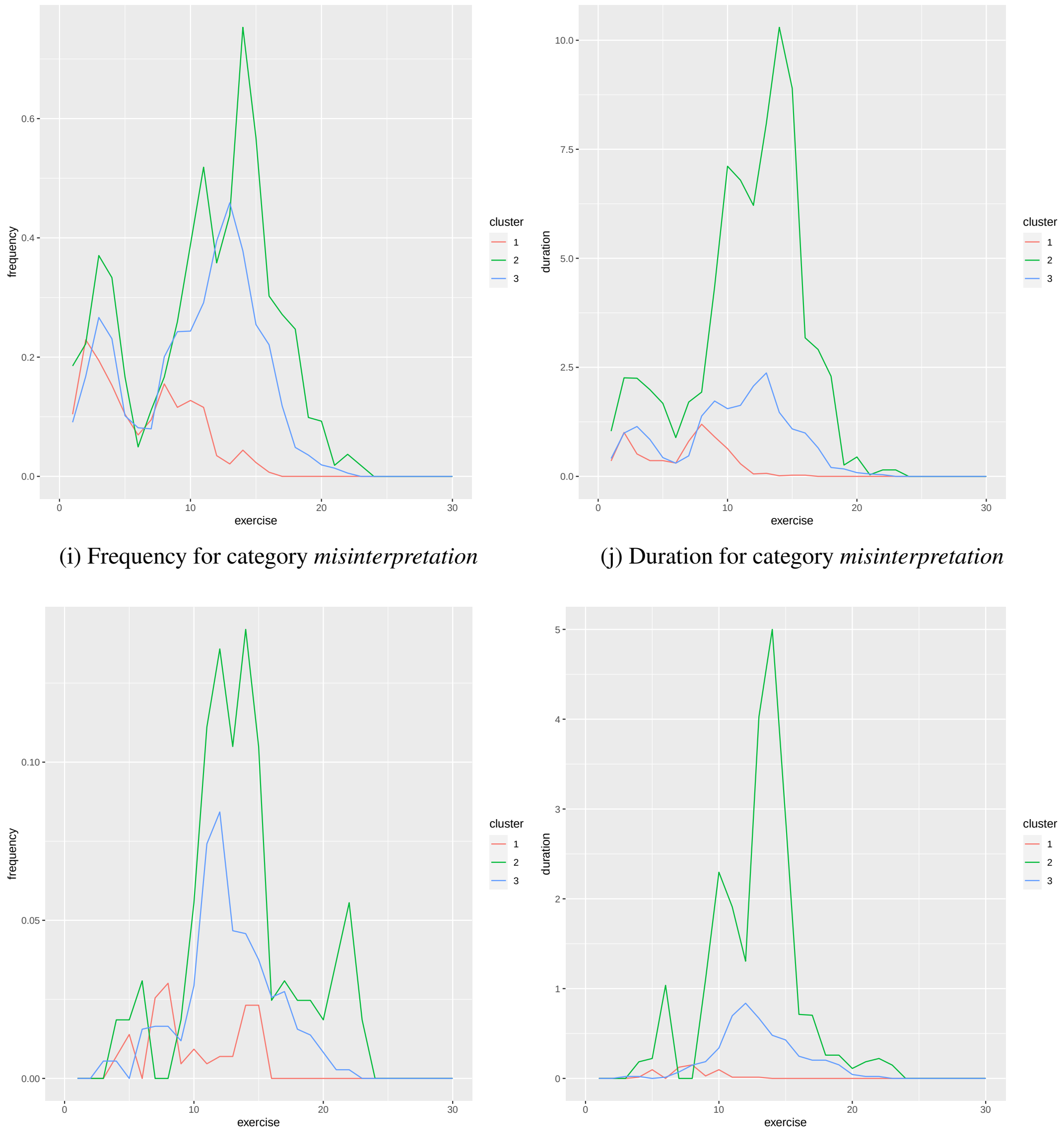


(e) Frequency for category *strategic*

(f) Duration for category *strategic*

(g) Frequency for category *sloppiness*

(h) Duration for category *sloppiness*



(i) Frequency for category *misinterpretation*

(j) Duration for category *misinterpretation*

(k) Frequency for category *domain*

(l) Duration for category *domain*

Figure E.8.: Clustering of error patterns for different categories