Subject: Data modeling and source code generating and AI

1) **L1** : Islam Souissi L1.1 OK

L0 : Wassila Kali : wassila-kali@hotmail.fr OK

Loan Rosseeuw : loanrosseeuw@live.fr OK

Lea Coutellier: leacou@yahoo.com (no mail confirmation but OK)

Idriss BLOUBOU (his name is not on the excel shared file) idryssdu945@gmail.com OK

- ⇨ 4 articles on Github
- ⇨ **My task (RAS)**

2) **L1** : Maria Lolita GONCALVES MELIM L1.2 lolita.goncalves91@gmail.com (she summarized the 8 articles into 1 by mistake)

L0 : BURLON Clara : carla.burlon@icloud.com (no mail confirmation but OK)

DEVARIEUX Lucas : devarieux.l@gmail.com OK

- ⇨ Only 1 article on Github
- ⇨ **Nothing to do**

3) **L1** : Morgane Gaspar :L1.3 morganegaspar77@gmail.com (resumated all in one?/ doesn't answer mail)

L0 : Leila Zine zineleila0502@gmail.com (no mail confirmation but OK)

Isiram OUKAKI : isiramoukakipro@gmail.com OK

Johan KIBANDA : johannrfz@gmail.com OK

Shaï NAKACHE shainakache@gmail.com OK

- ⇨ Only 1 article on Github
- ⇨ **Nothing to do**

4) **L1 :** Merveille Igueme L1.4 (mail not found/ didn't work ?)
L0 : Yasmine FAKIR yasminefakirpro@outlook.fr (no mail confirmation but OK)

TARCIN Audrey audrey.tarcin@gmail.com OK

Brandon (mail not found): can't find his summaries

DIAKITE Fousseynou naziroudiakho28@gmail.com OK

- ⇨ 12 articles
- ⇨ **I will do Merveille's work+ mine**

L1.4’s work:

Summary of Yasmine FAKIR:

-The first summary explores the integration of Foundation Models and Symbolic AI to automate the detection and mitigation of code vulnerabilities, emphasizing the need for automation in creating defect-free code, especially in collaborative and cloud-based systems. It discusses the challenges in existing approaches and proposes a solution combining code generation using Large Language Models (LLMs) with scalable defect elimination methods using symbolic AI. The article introduces the concept of an "Automated Software Factory" and outlines a technical approach involving static source-code analysis and infrastructure using Abstract Syntax Trees (ASTs). It concludes by highlighting the potential of Foundation Models in mitigating the software crisis but emphasizes the importance of addressing concerns related to intellectual property, bias, accountability, and job displacement.

-The second summary presents the development of an artificial neural network model aimed at automating the generation of source code for graphical user interfaces (GUIs) using the Python programming language. It outlines a comprehensive methodology involving the creation of a natural language dataset describing Python-programmed GUIs and the training of a deep neural network, specifically employing a transformer architecture. The article emphasizes the model's efficacy in processing natural language requests related to GUI creation in Python, facilitating the automatic generation of executable source code. It also suggests enriching the dataset with more diverse GUI scenarios and expanding the model's capabilities for generating source code for web components.

-The third summary proposes the Data-Centric Service Code Generation (DCServCG) model to enhance web service-based systems through automatic code completion. It addresses the lack of machine-readable documentation and structured metadata for web services by leveraging open-source software (OSS) projects to provide developers with accurate code suggestions during the programming process. The model incorporates crucial service-based code characteristics and employs conditional text generation to address bias and sequence-overlapping issues, leading to improved model generalization. The study concludes by offering an innovative solution for automatic code completion with a focus on data-centric modeling and leveraging OSS source code.

-The fourth summary introduces CodeGeeX, a powerful multilingual code generation model pre-trained on a massive code corpus comprising 23 programming languages. Unlike its predecessors, CodeGeeX supports code explanation and translation, in addition to code generation. The article details the model architecture, training data, efficiency optimizations, fast inference techniques, and real-world extensions to popular Integrated Development Environments (IDEs). It presents the HumanEval-X benchmark for evaluating the model's performance across various tasks and concludes by highlighting CodeGeeX's robustness, efficiency, and real-world accessibility as a valuable tool for developers worldwide.

Summary of Audrey:

The first summary discusses the development and experimental validation of a self-programmed artificial intelligence system capable of generating and modifying its own source code. The AI utilizes a language model for code generation, enabling it to program sub-models and autonomously improve its performance on tasks like MNIST, CIFAR-10, and EMNIST. The system's ability to auto-modify various aspects, including model architecture and computational capacity, is detailed, along with methods used for training, reprogramming, and other algorithms. Future research directions, such as training on diverse types of computer codes and implementing self-correction mechanisms, are also outlined.

The second summary explores the coding performance of ChatGPT, a large AI language model, through a crowdsourcing data-based framework analyzing social media data related to code generation. It delves into the methodology, user feedback, data collection from platforms like Twitter and Reddit, and sentiment analysis on the generated code. The impact of ChatGPT on AI technologies, sentiment analysis results across platforms, creation of a public Python dataset, and ethical concerns regarding biases in the generated code are discussed. Specific keywords of this document include ChatGPT, crowdsourcing, sentiment analysis, and social media data.

The third summary examines various AI tools, their code generators, and their impact on programming education, addressing opportunities and challenges they present. It discusses tools like OpenAI Codex, DeepMind AlphaCode, and Amazon CodeWhisperer, emphasizing their potential to standardize coding for students and make teaching more engaging. Ethical concerns, such as plagiarism and biases in AI-generated code, are highlighted, along with challenges in programming teaching and the risk of over-reliance on AI tools.

The fourth summary focuses on the use of deep learning in language models for automatically generating code sources, comparing different architectures like AWD-LSTM, QRNN, and Transformers. It discusses methods used in the research, dataset selection, models like GPT-2, BERT, and RoBERTa, and their performance evaluation on tasks like code generation and auto-completion. The importance of human evaluation and future research directions, including exploring different training methods and creating new evaluation measures, is emphasized. Keywords of this document include deep neural network architectures, source code generator, auto-completion, and pre-trained models.

Summary of Fousseynou:

The first summary discusses the validation and reliability of AI-generated code, emphasizing the need for effective metrics to assess its efficiency, correctness, and usability. The proposed CGEMs metric model aims to evaluate AI-generated code by considering factors like maintainability, complexity, and raw metrics such as lines of code. The paper highlights the importance of AI in reducing task burdens and introduces Monte-Carlo simulation methods to validate a statistically significant number of code samples.

The second and third summaries delve into the potential of large language models (LLMs) in scientific research, particularly in software engineering tasks like code generation, data analysis, and data visualization. These summaries present empirical evidence on LLMs' usefulness in research processes and explore use cases where LLM-based tools could enhance productivity. They also discuss the release of ChatGPT and the widespread attention garnered by LLMs, underscoring their significance in aiding various research tasks.

The fourth summary addresses the security concerns associated with AI code generators, focusing on potential vulnerabilities introduced through data poisoning attacks. It discusses attackers' methods to undermine system integrity by injecting vulnerable code into training data, thus compromising the generated code's security. The proposed data poisoning attack evaluates the security of NL-to-code generators by injecting software vulnerabilities into the training data used to fine-tune AI models, highlighting the importance of assessing AI code generator security.

⇨ **Summaries for L3:**

1) (Asma’s work): The summaries discuss various aspects of AI-generated code, large language models (LLMs), and their implications in software engineering and research.
The first summary set discusses the integration of emerging technologies across various domains, focusing on neural architectures, AI-driven code generation tools, and Large Language Models (LLMs). It highlights challenges and advancements associated with these technologies, such as translating natural language into source code, optimizing software development processes, and their impact on scientific research. It also introduces the integration of Foundation Models and Symbolic AI to automate the detection and mitigation of code vulnerabilities, emphasizing the need for automation in defect-free code creation. It proposes the concept of an "Automated Software Factory" and outlines technical approaches to address challenges in existing methods.
The second summary set focuses on various aspects of AI-generated code and LLMs, emphasizing their implications in software engineering and research. It introduces the integration of Foundation Models and Symbolic AI to automate code vulnerability detection, emphasizing automation in creating defect-free code. Additionally, it addresses security concerns associated with AI code generators, particularly data poisoning attacks.
The second and third summaries explore the potential of LLMs in scientific research, particularly in software engineering tasks such as code generation, data analysis, and data visualization. They highlight empirical evidence on LLMs' usefulness in research processes and discuss use cases where LLM-based tools could enhance productivity.
The fourth summary addresses security concerns associated with AI code generators, focusing on potential vulnerabilities introduced through data poisoning attacks. It discusses methods to undermine system integrity and proposes a data poisoning attack to evaluate the security of NL-to-code generators.
Globally, these summaries underscore the importance of automation, LLMs, and security considerations in AI-generated code and software engineering. They provide insights into emerging technologies and methodologies aimed at improving code quality, productivity, and security across diverse domains, highlighting challenges, opportunities, and advancements in the field.

2) These articles cover a wide range of topics related to software development and AI, from code generation and data management to software evolution and model-driven engineering. Each topic offers unique perspectives on how technological advances can be used to solve complex problems in these fields. Most articles address the use of models and metamodels in a variety of contexts, such as aircraft design, software evolution, model comparison and code generation. They collectively explore the integration of model-driven approaches in engineering and software development, emphasizing advancements and challenges in this domain. They advocate for the adoption of ontology-based frameworks, highlight the benefits of modeldriven software engineering (MDSE), propose automation solutions for metamodel and transformation co-evolution, and provide insights into the nuances of modeling languages. In "Ontologies in Aircraft Design », challenges arise from overlapping content, addressed through ontology-driven integration with the Oida framework, fostering coherence and collaboration. Similarly, MDSE adoption enhances software quality, but managing model configurations remains problematic. Solutions like automation operators from "Operators for Co-Evolution of Metamodels and Transformations" streamline this, reducing manual intervention and ensuring transformation integrity. Understanding model and metamodel distinctions, as discussed, aids effective communication and collaboration, enhancing model

comprehension. "On the Evolution of Lehman’s Laws" pays homage to Professor Manny Lehman, tracing his journey and discussing the relevance of his laws in modern software contexts, emphasizing adaptation to architectural shifts, demodularization, and the significance of open source and agile methodologies. The article "Different Models for Model Matching" delves into effective versioning mechanisms for managing changes in model-based artifacts within model-driven engineering (MDE). It highlights current limitations in capturing model differences and advocates for robust techniques, especially for UML diagrams and generalized metamodels. The absence of a universal solution underscores the importance of contextspecific approaches. Meanwhile, "Méthodologie de Développement Objet" explores objectoriented development methodologies, addressing code generation from specifications and the trade-off between user-friendly informal languages and more formal, less ambiguous ones. It introduces formal language concepts, grammar, and model transformations, comparing their applications in optimization and code generation. It also questions the feasibility of model-driven engineering compared to traditional software development processes. Together, these articles provide insights into the evolution, challenges, and methodologies of software development, from historical foundations to contemporary practices and future considerations. They underscore the importance of adapting methodologies to evolving contexts while addressing specific challenges inherent in software engineering. The article, "A Comparison of Model Migration Tools », explores challenges and solutions in model migration when underlying metamodels change. It emphasizes automating migration with various tools, aiming to compare and evaluate them for different scenarios. The comprehensive comparison evaluates tool performance and guides tool selection. Conversely, "Model Differences in the Eclipse Modeling Framework" addresses managing structural changes in software modeling. It introduces EMF Compare within the Eclipse Modeling framework, offering a solution to model difference challenges. "Model Matching Challenge," highlights model comparison's importance in model-driven and software engineering. It stresses the need for specialized tools and presents challenges in achieving quality results with existing algorithms. Lastly, "A Comparison of Model Migration Tools", compares AML, COPE, Ecore2Ecore, and Epsilon Flock for model migration. It aims to aid users in selecting the most suitable tool. Together, these articles shed light on the complexities of model migration, model differences, and model matching, offering insights into the tools, techniques, and challenges involved in managing model-based artifacts in software engineering contexts. These articles explores the educational landscape of AI-based code generation in programming education, stressing its transformative potential. They investigates ChatGPT's performance in code generation and users' attitudes towards AI-based programming tools, using social data for analysis. In contrast, the third article conducts a comparative study on ChatGPT 3.5's code generation across multiple languages, identifying strengths, limitations, and industry implications. Lastly, the fourth article introduces a groundbreaking self-programming AI using code-generating language models, validating its capabilities and proposing future research directions. In summary, these articles collectively highlight the transformative potential of AI-driven code generation across educational, practical, and technological domains. While presenting numerous opportunities for enhancing programming education, accelerating software development, and advancing AI research, they also underscore the need for thoughtful consideration of ethical, pedagogical, and technical challenges inherent in this rapidly evolving field.

3) These articles explore various aspects of software engineering and artificial intelligence, with a particular focus on Intelligent Driver Model (IDM) model-driven engineering, which is a theory used in traffic engineering to model driver behavior in traffic. It aims to describe how drivers adjust their speed according to various factors such as distance from other vehicles, current speed and desired speed. This model is often used in traffic simulation and the design

of traffic control systems. IDM is often formatted with megadata and AI for vector-borne disease prediction. The first article proposes a catalog of refactorings for model-to-model (M2M) transformations in IDM, highlighting the importance of maintainability in these operations. Indeed, direct communication between two devices without humans requires constant maintenance to automate processes and improve them over time. Despite this, the lack of automation remains a major problem, even when predefined programs are applied. The component model for model transformations, offering a reusable approach to their conceptualization and pipeline implementation. The article discusses the limitations of meta-models, which may require additional manual work. The method for adapting OCL constraints is therefore important. OCL, or Object Constraint Language, is a formal specification language used mainly in the field of model-driven engineering (MDE). It is used to define constraints and rules on models created using modeling languages such as UML (Unified Modeling Language). OCL can be used to specify conditions and invariants that must be respected by models to guarantee their consistency and conformity with system specifications. It is used to express static and dynamic constraints on model objects, as well as to define operations on these objects. OCL is often used in conjunction with other modeling languages to enrich the specification of software systems and facilitate the analysis and verification of their properties. They evolve from their meta-model within the IDM framework, aiming to maintain compliance and consistency of constraints throughout the evolution process. Examining the current state of AI in industries and proposing a "data ecosystem" framework to overcome data management and AI adoption challenges.
Included are: - Data collection. - Data integration. - Predictive modeling (disease risk prediction). - Machine learning. - Early warning system. - Decision support tool. The introduction of a DSL(designed to facilitate the extraction of different types of models from existing code bases, helping to understand the structure, behavior and dependencies of the software system.) would therefore be important, but is the subject of much debate. The functionalities and capabilities of the DSL, provides examples or case studies illustrating its application in real projects, and evaluates its efficiency and user-friendliness. Finally, the last article explores the performance and scalability of EMF as a model comparison framework, highlighting the factors that influence its effectiveness in different scenarios.
Here's a list of some scenarios:
- Model size (the more elements, the longer the processing time).
- Algorithm efficiency. - Resource constraints (memory, CPU). - Optimization techniques. - Tool integration. - User configuration and tuning. In summary, these articles provide an in-depth overview of various areas of research and practice in software engineering and AI, covering topics ranging from model maintenance to vector disease prediction, data management and model comparison.