ABSTRACTS:

**Ontologies in Aircraft Design**:

The development of a new aircraft involves various teams working simultaneously on different aspects of the design. These teams create models that may overlap in content. When changes are made to one model without updating others, inconsistencies arise through the process. Integrating model parts is challenging due to the diverse modeling approaches used by different teams.

Currently, two approaches are used to address this challenge: separating concerns between models and using standardized data formats for exchanging model parts. However, during conceptual aircraft design, clear separation of models is difficult, leading to overlapping content.

To counter this overlapping tendencies, standardized data formats are used, but they lack domain-specific concepts, making it hard to identify overlaps and conflicts between models. This results in inefficient and error-prone maintenance of consistency.

This dissertation proposes a semiautomated ontology-based approach for model integration in aircraft design. It introduces the Oida framework, which uses a domain-specific reference ontology to consolidate overlapping content and exchange selected non-overlapping content between models. The framework allows model owners to map their elements to the reference ontology, evaluates these mappings, identifies matching elements, and supports conflict resolution and model part importation.

The effectiveness of the Oida framework was evaluated through experiments using real sample models from different aircraft design tools. Aircraft designers found the framework easy to use and considered the results generated by it to be correct.

This approach not only automates aircraft model integration but also improves collaboration during the interdisciplinary process of aircraft design.

**Model-Driven Software Engineering (MDSE)**:

Software plays a crucial role in our daily lives, from playing music to managing emails, to running a video game and even assisting in driving cars. However, traditional software engineering struggles to keep up with the increasing demand for high-quality software. To address this, Model-Driven Software Engineering (MDSE) is gaining popularity as a new paradigm (=rule).

MDSE aim is to overcome the limitations of traditional software engineering by raising the level of abstraction. Instead of writing code directly, MDSE advocates using models and model transformations to produce software. These models are typically represented graphically, making it easier to understand the relationships between different parts of the software and reducing the likelihood of errors during development.

Formal model transformations allow for the generation of various artifacts from models at different stages of software production. For instance, these artifacts can be used for model checking or simulation, enabling early testing and reducing the chances of defects in the final product.

Despite its potential benefits, MDSE still lacks mature methods and techniques, particularly in model configuration management (MCM). This is where my research comes in. I focused on developing generic methods and techniques to support model configuration management, with a specific emphasis on model evolution and co-evolution.

To support model evolution, I developed methods for representing, calculating, and visualizing state-based model differences. Unlike previous approaches that dealt with these aspects separately, my research integrated them, resulting in a generic representation format for model differences that can be used across different types of models.

Model co-evolution, the process of adapting models as their metamodels evolve, was addressed through a three-step solution. First, I introduced a special metamodel representation. Then, I reused existing methods for model evolution to represent and calculate metamodel differences. Finally, I defined an algorithm to adapt models based on the calculated metamodel differences.

To validate these approaches, I developed tools for comparing models, visualizing differences, and supporting model co-evolution. Additionally, I conducted experiments to compare these tools with existing industrial tools, providing further validation for my approach.

**Operators for Co-Evolution of Metamodels and Transformations**:

In Model Driven Software Development (MDSD), maintaining consistency between models, metamodels, and transformations as they evolve presents challenges. This paper focuses on co-evolution, specifically the relationship between metamodels and transformations.

Changes to metamodels can disrupt existing transformations, requiring manual alignment. To address this, the paper proposes a novel approach using operators to facilitate the adaptation of transformation descriptions to metamodel changes.

These operators enable automatic or semi-automatic co-evolution of transformations alongside metamodel changes. They include functionalities such as renaming elements, adding or deleting classes/attributes/relations, and restructuring inheritance hierarchies.

Implemented in Java and tested on representative metamodels and transformations, these operators streamline the co-evolution process, preserving syntactic correctness and reducing manual intervention on software engineering (=SE=creation of software).

Despite the complexity of MDSD projects, this approach offers promising results in maintaining the integrity and coherence of transformations in evolving metamodels.

**Modelling Language :**

**Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement**

- A model represents something specific, like a particular system, object, or concept.

- It's like a blueprint or a description of how something works or looks.

- For example, a model of a car would include details about its design, features, and specifications.

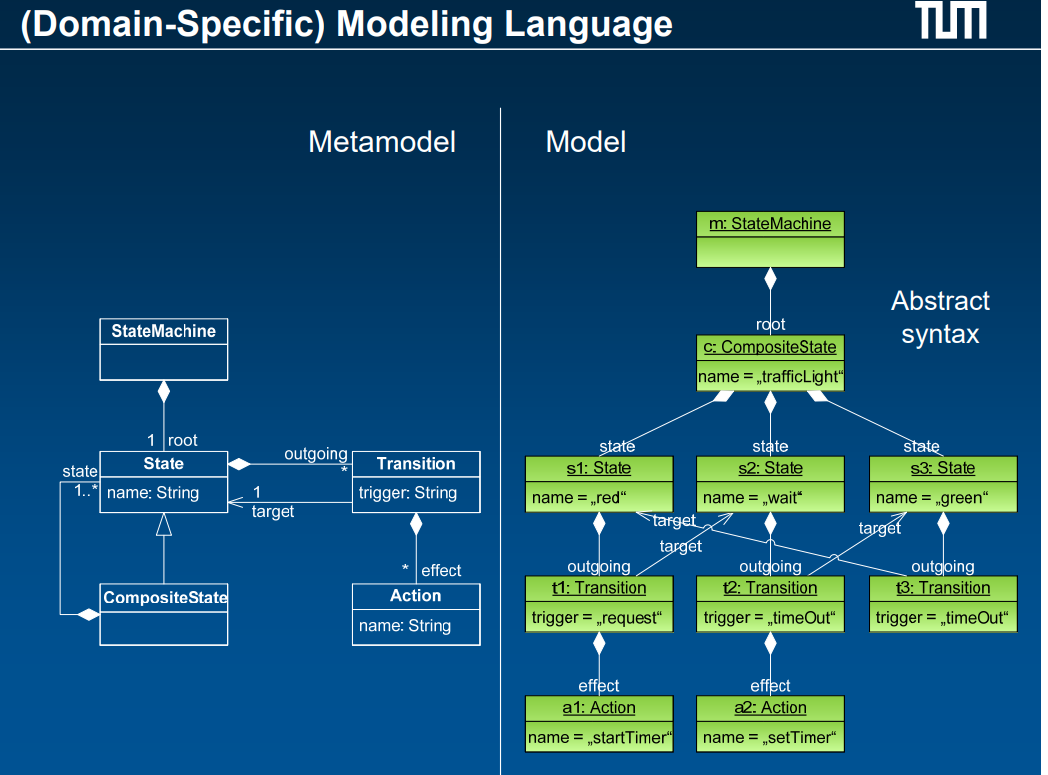
- A metamodel, on the other hand, describes the structure and rules that govern models.

- It defines the types of elements that can exist in a model and how they relate to each other.

- Essentially, it's a model of models.

- For instance, a metamodel for cars might specify that a car model must include elements like "engine," "wheels," and "doors," and it might define how these elements can be connected.

We can say that a model is like a specific instance, such as a particular car, while a metamodel is like the template or rules for creating and understanding models of cars.

****

Concrete syntax, refers to the actual representation of a model using symbols, text, or diagrams that are understandable to humans.

It's how the model is visually or textually expressed, often in a specific language or notation.

Concrete syntax is what you see and work with directly when creating or viewing a model.

For example, in a UML class diagram, the boxes, lines, and symbols used to represent classes, attributes, and relationships constitute the concrete syntax.

On the other hand, the abstract syntax, on the other hand, focuses on the underlying structure and meaning of the model, independent of its visual or textual representation.

It defines the logical elements and relationships within the model without specifying how they are visually represented.

Abstract syntax is more concerned with the semantics or meaning of the model elements.

Using the same example, in a UML class diagram, the abstract syntax would describe the concepts of classes, attributes, and associations and how they are related to each other in a formal, language-independent manner.

In essence, concrete syntax deals with how a model looks or is presented visually or textually, while abstract syntax deals with the logical structure and meaning of the model elements.

DEFINITIONS:

Metamodel: A metamodel is a model that defines the structure and constraints of other models within a certain domain. It describes the elements, relationships, and rules that govern those models. For example, in a modeling language for cars, a metamodel might define that a car model should have attributes like "make," "model," and "year."

Model Transformation: Model transformation involves converting or modifying one model into another model. It's like translating information from one representation to another. For instance, transforming a model of a car's design into a model of its manufacturing specifications.

Co-evolution: Co-evolution refers to the simultaneous evolution or change of interconnected elements. In the context of models and metamodels, it means that when one evolves, the other may need to change accordingly to maintain compatibility. For example, as a metamodel defining a car's features evolves to include electric cars, existing car models might need to be updated to align with these changes.

Operators: In the context of model-driven development, operators are functions or tools used to manipulate models and metamodels. These operations could involve tasks like merging, splitting, or transforming models. For instance, an operator might be used to merge two different models of a car's design into a single comprehensive model.

Model Driven Software Development (MDSD): MDSD is an approach to software development where models are used as primary tools throughout the development process. Instead of writing code directly, developers create models that capture various aspects of the software, which are then transformed into executable code. For example, in MDSD, a model representing the system's architecture could be automatically transformed into code implementing that architecture. It’s like a blueprint.

Ontology: An ontology is a formal representation of knowledge within a specific domain, including concepts, relationships between those concepts, and constraints on how they can be used. It's like a structured vocabulary or taxonomy for organizing information. For instance, an ontology for animals might define classes like "mammals," "birds," and "reptiles," along with their properties and relationships.

Paradigm: A paradigm as a way of thinking or doing things that everyone in a certain area agrees on. It's like the rules of the game or the usual way of doing stuff in a particular field.