



# **A Novel Textual Meta Modeling Language for the Web**

**A Thesis Submitted for the Degree  
of Master of  
Science in Computer Science**

Kasun Anupama H. Gamage

University of Colombo School of Computing


2024



## Declaration

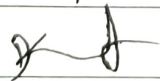
Name of the student: Kasun Anupama Halwitigala Gamage
Registration number: 2017/MCS/032
Name of the Degree Programme: Master of Science in Computer Science
Project/Thesis title: A Novel Textual Meta Modeling Language for the Web

1. The project/thesis is my original work and has not been submitted previously for a degree at this or any other University/Institute. To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.
2. I understand what plagiarism is, the various types of plagiarism, how to avoid it, what my resources are, who can help me if I am unsure about a research or plagiarism issue, as well as what the consequences are at University of Colombo School of Computing (UCSC) for plagiarism.
3. I understand that ignorance is not an excuse for plagiarism and that I am responsible for clarifying, asking questions and utilizing all available resources in order to educate myself and prevent myself from plagiarizing.
4. I am also aware of the dangers of using online plagiarism checkers and sites that offer essays for sale. I understand that if I use these resources, I am solely responsible for the consequences of my actions.
5. I assure that any work I submit with my name on it will reflect my own ideas and effort. I will properly cite all material that is not my own.
6. I understand that there is no acceptable excuse for committing plagiarism and that doing so is a violation of the Student Code of Conduct.

Signature of the Student	Date (DD/MM/YYYY)
	28/10/2024

## Certified by Supervisor(s)

This is to certify that this project/thesis is based on the work of the above-mentioned student under my/our supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

	Supervisor
Name	Prof. D. D. Kammarethi
Signature	
Date	28/10/2024

# Acknowledgement

I would like to express my sincere gratitude to the University of Colombo School of Computing for providing me with the opportunity to pursue my Master of Science degree. The knowledge and skills I have acquired during this program have been invaluable.

My deepest appreciation goes to my MSc supervisor, Prof. Damitha Karunaratna, for his guidance and support throughout my study. His insights and advice have been crucial to the completion of this thesis.

I am also grateful to my MSc project team, led by Dr. Kasun, for their assistance with the administrative intricacies and their continuous encouragement. Their support has been instrumental in navigating the complexities of this program.

Finally, I extend my heartfelt thanks to my family for their constant love, patience, and encouragement. Their support has been my greatest source of strength throughout this journey.

Kasun Anupama H. Gamage

# Abstract

Model Driven Web Engineering (MDWE) has long relied on meta-modeling languages for the structured design, development, and maintenance of web applications through high-level abstractions and formal specifications. However, the rapid evolution of web technologies has outpaced these languages, leading to shortcomings such as outdated perspectives and a lack of alignment with modern industry needs. This research investigates these issues and explores solutions through literature, focusing on meta-programming concepts.

The core of this research is the introduction of LMTH, a new Meta Modeling Language designed to standardize and streamline web application design. LMTH is a 4th Generation Meta Programming Language which is textual, less opinionated and declarative yet turning developed as a superset of TypeScript, which is well-established in the industry. The language's grammar is presented in EBNF, and it includes a "model of reality" standard library, providing a common ontology vocabulary sharable among various solution applications. LMTH supports defining solutions within data, process, UI, user, and state models in order to generate end-to-end web application code complying with semantic web standards and WCAG standards.

To demonstrate LMTH's efficacy, a transpiler based on LLVM was developed, and sample applications were generated using the MEAN stack. Evaluation covered implementation of requirements, language expressivity and developer support, addressing limitations and extensibility of the standard library, performance benchmarks and standard compliance, all yielding satisfactory results. Comparative analysis situates LMTH within the broader context of web development, highlighting its implications for enhanced developer experience, maintenance efficiency and standardization and interoperability.

This research emphasizes the adoption of semantic web concepts, compliance, and accessibility, paving the way for future-proof web development. Limitations and potential avenues for future work have been identified, laying the foundation for ongoing advancements in the field.

**Keywords:** Model Driven Web Engineering (MDWE), Meta-Modeling Languages, LMTH, Semantic Web, Web 3.0, Web Accessibility, LLVM, Web Application Design, Ontology Vocabulary, Standardization, Developer Experience, Web Engineering, Future-Proof Web Development, Programming Languages, 4GL

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background and Motivation . . . . .	1
1.2	Problem Statement . . . . .	1
1.3	Research Question . . . . .	2
1.4	Expected Outcomes . . . . .	3
1.5	Evaluation Scope . . . . .	4
1.6	Thesis Structure . . . . .	4
<b>2</b>	<b>Literature Review</b>	<b>4</b>
2.1	Foundations of the Web and Its Evolution . . . . .	4
2.2	Model Driven Web Engineering . . . . .	5
2.3	Web 3.0 and the Semantic Web . . . . .	11
2.4	Meta Programming and Compiler Design . . . . .	15
2.5	Alternatives including Artificial Intelligence(AI) based approaches . . .	16
<b>3</b>	<b>Methodology</b>	<b>18</b>
3.1	Language Design . . . . .	18
3.2	Language Specification . . . . .	21
3.3	The LMTH Ecosystem . . . . .	48
3.4	Implementation . . . . .	48
3.5	Development Environment . . . . .	55
<b>4</b>	<b>Evaluation and Results</b>	<b>57</b>
4.1	Objectives . . . . .	57
4.2	Scope . . . . .	58
4.3	Specific Evaluation Criteria . . . . .	58
4.4	Evaluation Tools and Resources . . . . .	59
4.5	Evaluation Methodology . . . . .	61
4.6	Evaluation Results . . . . .	62
4.7	Overall Analysis . . . . .	63
<b>5</b>	<b>Conclusion and Future Work</b>	<b>63</b>
5.1	Summary of Findings . . . . .	63
5.2	Implications . . . . .	66
5.3	Limitations . . . . .	67
5.4	Future Work . . . . .	69
5.5	Final Remarks . . . . .	70

## List of Figures

1	MDWE in Context . . . . .	8
2	Timeline of first generation MDWE approaches . . . . .	8
3	Semantic Web Technology Stack . . . . .	13
4	Hierarchical Overlap of LMTH, TypeScript and JavaScript Syntax and Features . . . . .	20
5	Domain model as a common basis . . . . .	22
6	The LMTH Eco System . . . . .	49
7	Rendered IDataSingle Unit . . . . .	52
8	Rendered IDataSingle Unit 2 . . . . .	52
9	Rendered IDataSingle Unit - Alternative Template . . . . .	53
10	Rendered IDataSingle Unit . . . . .	53
11	Rendered Process Unit . . . . .	54
12	Process Unit after further styling . . . . .	54
13	Default Menu Rendering . . . . .	55
14	Menu after further styling . . . . .	55
15	Menu Rendering - Alternative Template . . . . .	56
16	Sample Lighthouse Report on Scenario 2 Products Listing Page . . . .	62
17	Lighthouse Report Scenario 2 Individual Product Page . . . . .	63
18	Compliance Testing - Sports News and Scores Application . . . . .	64
19	Compliance Testing - E commerce application . . . . .	65

**List of Tables**

1	Model to Code mapping with selected tech stack . . . . .	51
2	Language Evaluation Criteria . . . . .	60



## Abbreviations

**CASE** : Computer-Aided Software Engineering

**CPU** : Central Processing Unit

**IDE** : Integrated Development Environment

**LLVM** : Low Level Virtual Machine

**NPM** : Node Package Manager

**OQL** : Object Query Language

**PC** : Personal Computer

**RDF** : Resource Description Framework

**RDFa** : Resource Description Framework in Attributes

**S-P-O** : Subject-Predicate-Object

**TAG** : Technical Architecture Group (in W3C)

**URI** : Uniform Resource Identifier

**URN** : Uniform Resource Name

**VS Code** : Visual Studio Code

**W3C** : World Wide Web Consortium

**WCAG** : Web Content Accessibility Guidelines

**WebML** : Web Modeling Language

**XML** : Extensible Markup Language

**IR** : Intermediate Representation

**MEAN** : Mongo-Express-Angular-Node

**OWASP** : Open Worldwide Application Security Project

**AJAX** : Asynchronous JavaScript and XML

**WebRTC** : Web Real-Time Communication

**API** : Application Programming Interface

**CRUD** : Create, Read, Update, Delete

# 1 Introduction

## 1.1 Background and Motivation

Model Driven Web Engineering (MDWE) has long employed meta-modeling languages as a cornerstone for designing, developing, and maintaining web applications. These languages have traditionally provided a structured approach to model web applications through high-level abstractions and formal specifications. However, the rapid evolution of web technologies and industry practices over recent years has outpaced the development and adaptation of these meta-modeling languages. This discrepancy has led to several significant shortcomings in their current application:

**1. Outdated Perspectives:** The existing meta-modeling languages are rooted in older paradigms of the web, focusing primarily on static and server-driven architectures. Modern web applications, however, have shifted towards more dynamic content with Web 3.0. Existing meta-models do not fully capture these contemporary design patterns and technologies, leading to a gap between the models and the actual implementation.

**2. Developer Familiarity:** Contemporary web developers are increasingly proficient in modern development tools, frameworks, and languages. The modeling languages used in MDWE do not align well with the development practices and tool-chains familiar to today's developers. This misalignment creates a steep learning curve and reduces the accessibility and adoption despite the fundamental benefits of these modeling languages within the developer community.

**3. Industry Needs:** The needs of the web development industry have shifted towards rapid prototyping, iterative development as well as various standard compliance requirements. Modern development practices emphasize agility, collaboration, and frequent delivery, which are not adequately supported by the existing meta-modeling languages. These languages often lack the flexibility and responsiveness required to meet current industry standards and practices.

Given these shortcomings, there is a critical need to re-evaluate and update the meta-modeling languages used in Model Driven Web Engineering. This thesis aims to identify the specific deficiencies of these languages in the context of modern web development, explore how contemporary web practices and technologies can be better integrated, and propose new paradigms that can bridge the gap between MDWE methodologies and the current state of web application development. The goal is to develop a more relevant, accessible, developer-centric and industry-aligned approach to meta-modeling in web engineering that can effectively support the development of modern web applications.

## 1.2 Problem Statement

The current meta-modeling languages used in Model Driven Web Engineering (MDWE) are failing to meet the demands of modern web application development in areas of developer experience, standard compliance and interoperability with established standards and tools.

These existing meta modeling languages are based on outdated paradigms where web content is not treated as data which results in them not being aligned with

contemporary development tools and practices, and lack the flexibility required for rapid and iterative development processes. This gap between traditional meta-modeling languages and current web technologies and industry practices results in inefficiencies and barriers to adoption, necessitating a comprehensive re-evaluation and update of these languages to better support the development of modern web applications.

Given these issues, there is need to evolve the meta-modeling languages used in MDWE. This update should address the deficiencies of current languages by incorporating modern web practices and technologies, and propose new paradigms that align with the latest industry standards. The aim is to develop a meta-modeling language that is relevant, accessible, and capable of effectively supporting the design and development of modern web applications.

### **1.3 Research Question**

How can we provide a standardized meta modeling language that allows for the efficient and effective design of web applications adhering to accepted standards and supporting model driven web development across various types of problem domains while being rooted in a more data-centric perspective of the web?

#### **1.3.1 Secondary Questions**

The following dissects the above primary problem by breaking it into focused sub-questions. Each sub-question is designed to investigate different aspects of the primary question, offering insights into causes, current solutions, innovations, practical applications, and metrics for success. Together, these questions aim to comprehensively address the primary research problem and guide the exploration of its resolution.

- What are the shortcomings of existing modeling approaches for the web, specifically in the context of model-driven engineering?
- How can these identified shortcomings be addressed with a novel language?
- What are the requirements and standards for modern web applications that are not specifically addressed by existing modeling languages?
- What design principles should be adopted for creating a meta modeling language tailored for the web?
- How can these principles be systematically applied to ensure the language overcomes existing shortcomings while facilitating the enhancements?
- What are the possible implementation strategies for this meta modeling language?
- How can these strategies be validated and tested for effectiveness?
- How to support the automatic generation of code from models developed with this novel language?

- How can the new meta modeling language inter-operate with existing web development tools and platforms?
- What challenges might arise during integration, and how can they be addressed?
- What are some practical case studies where the new meta modeling language can be applied?
- How does the new language perform in real-world scenarios compared to existing approaches?

## 1.4 Expected Outcomes

The expected outcomes and deliverables of this research are listed below. This outlines the overall scope of work in this research.

1. Design and implement a new meta modeling language as an extension of the leading Model-Driven Web Engineering (MDWE) standards, incorporating relevant advancements from other research in the field.
2. Ensure that the extended language syntax maintains support for semantic web standards at the model level, facilitating the representation of semantic concepts and relationships within web application models.
3. Ensure the expressiveness of the language to cover a broad range of aspects necessary for generating web applications while effectively integrating semantic web support, thereby enabling the representation of complex data structures and semantic annotations within the models.
4. Develop a compiler capable of translating models constructed using the extended language into fully-fledged end-to-end web application code compliant with semantic web standards specifically RDFa.
5. Validate the effectiveness and correctness of the generated web applications by producing a series of moderately complex web applications as proof of concept, demonstrating the practical application and utility of the developed language and compiler.
6. Evaluate the degree to which semantic web standards and other chosen standards are correctly implemented in the generated web applications using validators and other relevant assessment tools, ensuring compliance with established standards and best practices in semantic web engineering.
7. Formally present the research findings within an academic context through research publications, disseminating the insights gained from the study and contributing to the body of knowledge in the field of semantic web engineering and MDWE.
8. Conduct a comparative analysis to place the novel language within the broader context of web development, highlighting its implications for enhanced developer experience, maintenance efficiency standardization and interoperability.

9. Identify limitations and potential avenues for future work, laying the foundation for ongoing advancements in the field.

## **1.5 Evaluation Scope**

The scope of this research includes the following key points:

- Produce the modeling language definition, providing primitives that are extensible and broad enough to cover most of the requirements of the sample scenarios chosen.
- Code generation is limited to the MEAN stack and several limited chosen libraries.
- Standard compliance, with a primary focus on Semantic Web Compliance through RDFa.
- Optionally attempt to cover compliance for:
  - Web Content Accessibility Guidelines & W3C standard 2.0 A level
  - OWASP Application Security Verification Standard (ASVS)

## **1.6 Thesis Structure**

Chapter 1 establishes the context of the thesis by presenting the background, motivation behind the research, objectives, and scope. In Chapter 2, the study advances with a comprehensive review of relevant literature, covering topics such as model-driven web engineering, Web 3.0 and semantic web standards, and meta-programming with 4th Generation Languages. Chapter 3 outlines the design principles and specifications of the new language, followed by a detailed description of its implementation process.

The evaluation section presents the methodology and criteria used for assessment, including specific standards for compliance and the novel language's position within the broader context of meta-modeling languages through relative comparison. It also includes results from validating the language against defined criteria by generating three sample web applications, concluding with an analysis of the findings.

Finally, in Chapter 5, the thesis summarizes the contributions, discusses broader implications, acknowledges limitations, and suggests directions for future work. This includes potential enhancements and extensions for the language as well as areas for further research and development.

# **2 Literature Review**

## **2.1 Foundations of the Web and Its Evolution**

T. J. Berners-Lee (1989) has proposed the concept of a system for information management and sharing using "hypertext" to link and access documents across a decentralized network, which is the foundation of what we know today as the World Wide Web.

Key points of this proposal include the need for a more effective way to manage and share information, The use of "hypertext" as a method of linking documents through "hyperlinks," allowing users to navigate between related pieces of information easily and emphasizes decentralization. It also describes the client-server architecture and introduces the concepts of Uniform Resource Identifier (URI), Hypertext Markup Language (HTML), and Hypertext Transfer Protocol (HTTP), which are the cornerstones of the web.

Later, as the web evolved and its potential became evident, T. Berners-Lee et al. (2001) has proposed a new type of web content designed to be meaningful to computers, enabling more intelligent and efficient web interactions, known as the semantic web. This enables computers to understand and process the meaning of information on web pages, unlike the web content available at the time (web 2.0), which was designed solely for human readability. With this new form of web content, software agents can interpret and manipulate data meaningfully, facilitating tasks like cross-referencing schedules, verifying service provider details, and managing preferences.

The Semantic Web holds the promise of revolutionizing web interactions by enabling computers to process and understand web content in a meaningful way, thereby facilitating the development of more efficient and intelligent systems capable of performing complex tasks with minimal human intervention. This publication also highlights the potential role of the Resource Description Framework (RDF), which had already been published as a standard, in achieving the goals of the Semantic Web.

At the core of the Semantic Web is the idea of adding meaning and context to web content. This is achieved through semantic markup, where data is annotated with machine-readable metadata that defines its significance and relationships. One of the key mechanisms enabling this understanding is the use of hyperlinks to connect data to definitions and rules.

This represents an important paradigm shift, capturing a fundamental evolutionary step of the World Wide Web. These foundational concepts have influenced various practical applications.

## **2.2 Model Driven Web Engineering**

Model-Driven Web Engineering (MDWE), which emerged from Model-Driven Engineering (MDE), is a specialized approach for developing web applications through high-level metamodels rather than traditional coding. It has been explored within the literature in terms of various standards, application cases, as well as limitations. This subsection presents extracts of such discussions, highlighting key contributions and ongoing challenges in the field and applicability within this research.

The proposal by Ceri et al. (2000) on WebML as a modeling language for designing web sites is one of the most influential works in this field, significantly contributing to the research and development of MDWE methodologies. A significant portion of research in MDWE directly or indirectly stems from this proposal. The Essence of this proposal is summarized in the following paragraphs.

This proposal presents WebML along with several meta modeling primitives. It would allow designers to outline the essential features of a website at a high level without concerning about architectural specifics. The intuitive graphical repres-

entation would supports communication with non-technical team members and is compatible with CASE tools. WebML also includes an XML syntax for generating website implementations automatically. Core Models in WebML are as follows:

- **Structural Model:** Defines the site's data content using entities and relationships compatible with classic notations like E/R models and UML class diagrams. It includes a simplified OQL-like query language for specifying derived information.
- **Hypertext Model:** Describes hypertexts published on the site, divided into:
  - **Composition Model:** Specifies pages and their content units, such as data, multi-data, index, filter, scroller, and direct units.
  - **Navigation Model:** Defines how pages and content units are linked, distinguishing between non-contextual and contextual links based on the structure schema.
- **Presentation Model:** Describes the layout and graphic appearance of pages using an abstract XML syntax, allowing for both page-specific and generic presentation specifications.
- **Personalization Model:** Models users and groups as predefined entities, supporting the storage of personalized content and the use of high-level business rules for reacting to site-related events.

The authors further point out that for simple web applications, stages of the development process can be skipped, using defaults to produce simplified solutions and enabling the creation of a default initial site view directly from the structural schema. This concept of defaults has significant relevance to this research in simplifying model definitions and thereby improving DX, which is a primary goal of this study.

Gamage (2017) has presented a workflow focussing on the Separation of Concerns which involves 3 models for code generation namely the data model, the process model and a unified model for presentation and navigation.

Model-Driven Web Engineering (MDWE), a specialization of Model-Driven Engineering (MDE), focuses on the development of web applications using model-driven techniques. This approach involves transforming models, typically expressed in a Domain Specific Language (DSL), into fully functional web applications, covering backend data management, business logic implementation, and frontend web or Rich Internet Application (RIA) components. MDWE enables the efficient creation of large-scale enterprise applications and facilitates the evolution of data-intensive web systems. Additionally, it employs model differencing techniques to assist in the migration and ongoing development of web applications, as evidenced in the exploration by Cicchetti et al., 2011.

Schauerhuber et al., 2006 note that within the context of Web Engineering, prominent standards utilized for this purpose include WebML and its evolved form, IFML, recognized as an OMG group standard. These models are then processed by code generation tools like Web Ratio, which stands out as one of the most widely

employed tools in conjunction with WebML. Tools tailored for MDWE, such as Web Ratio and RISE Editor, offer functionalities ranging from automatic interface composition and database generation to web service generation, aiming to streamline the engineering process of information systems.

Semantic Web Technologies are increasingly being integrated with Model-Driven Web Engineering to enhance software development and enterprise application development. Pan et al., 2006; Álvarez et al., 2010 both highlight the potential of model transformation in bridging the gap between ontology engineering and traditional software engineering, with Pan et al., 2006 specifically leveraging Model-Driven Architecture (MDA) and Ontology Definition Metamodel (ODM). Kovalenko et al., 2015 further explore the differences in model features and creation processes between Semantic Web and Model-Driven Engineering, while ‘Model-driven design and development of semantic Web service applications’ 2007 propose a model-driven methodology for designing and developing semantic Web service applications. These studies collectively underscore the value of integrating Semantic Web Technologies with Model Driven Web Engineering to improve the efficiency and effectiveness of software development.

A range of studies have explored the use of model-driven web engineering to develop semantic web technology compliant applications. Torres et al., 2006; ‘Model-driven design and development of semantic Web service applications’ 2007 both have proposed methodologies that integrate business processes and web engineering models to extract semantic descriptions and specify system data and functionality. Paydar and Kahani, 2015 extend this by introducing a semantic web-enabled approach for reusing functional requirements models, which involves annotating activity diagrams and measuring the similarity of use cases. Meliá and Gómez, 2006 further enhance these approaches by introducing the WebSA approach, which uses model-driven development to integrate functional and architectural aspects of web design, and provides transformations to platform-specific models. These studies collectively demonstrate the potential of model-driven web engineering in generating semantic web technology compliant web applications.

However, while some leading modeling languages and their associated code generation tools have shown potential, their practical application has been limited. Many of these proposals and explorations date back to the 2000s with limited to none further explorations and implementations following up on them, and they often focus solely on specific aspects such as web services rather than addressing the application as a whole. For instance, there has been promising research such as the WebML-RDFizer project, Semantic Rule Language based on the proposal by ‘OWL rules: A proposal and prototype implementation’ 2005 and the proposal by Der, 2012 which aimed to integrate semantic web standards into Model-Driven Web Engineering (MDWE).

### **2.2.1 Reflection, Limitations and Recommendations**

Rossi, Urbietta et al. (2016) has summarized several key findings based on a retrospective reflection on MDWE technologies from 1990 to 2016. While this paper could be considered somewhat dated in the rapidly evolving field of web engineering, its comprehensive analysis provides valuable insights. The paper discusses the challenges faced by the MDWE discipline, which had to address new issues while



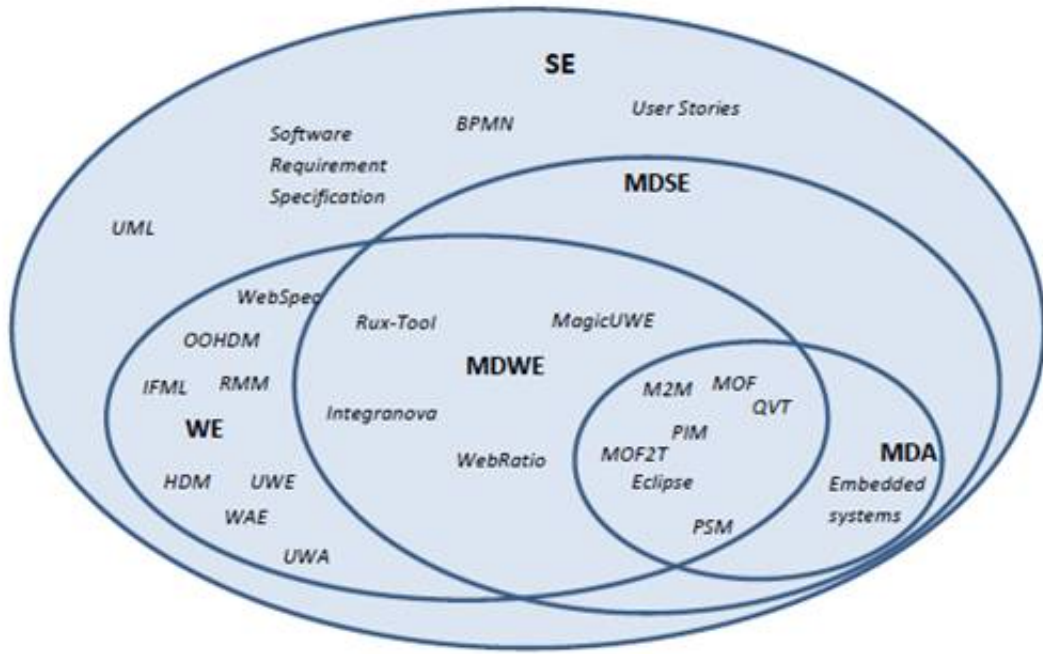


Figure 1: MDWE in Context

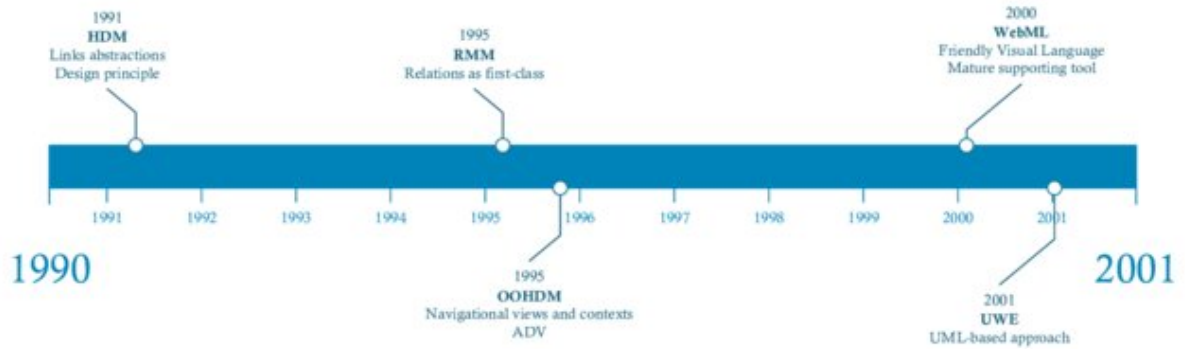


Figure 2: Timeline of first generation MDWE approaches

adapting existing solutions from software engineering and Model-Driven Software Engineering (MDSE). These challenges included aligning the Model-Driven Architecture (MDA) approach with the diverse architectural requirements of web development, the need for varied meta-models tailored to specific web application concerns, and the influence of emerging paradigms like service-oriented, social, and cloud computing on model development. The rise of multidisciplinary aspects and new stakeholder roles further complicated the adoption and development of MDWE approaches. Figures 1 and 2 in this paper by Rossi, Urbietta et al. (2016) puts leading MDWE and other related technologies in context and the timeline of first-generation MDWE approaches.

The evolution of early modeling languages into today's sophisticated approaches is also highlighted. This progression, driven by both technological needs and the broader context in which these technologies were developed, reflects the community's focus on understanding the rationale behind each step rather than merely cataloging existing methodologies and languages. Some of these advancements

were driven by new user requirements or technological innovations, such as personalization, semantic web applications, and business process support. Others aimed to enhance the software development lifecycle by improving requirements specification, separating concerns more effectively, and advancing meta-modeling and transformations.

Architectural issues in MDWE have also been a significant research focus. For example, the Web Software Architecture (WebSA) approach sought to integrate architectural concerns with functional models within the model-driven process, applying the necessary transformations to generate the running application. However, early MDWE proposals often overlooked aspects such as testing and maintenance, which were later addressed in subsequent approaches.

Specific advancements during this period included WebML's enhancement of its notation to support web services and OOWS's incorporation of web services into its design methodology. Additionally, there was a broader adoption of OMG standards to support MDA, including the Meta-Object Facility (MOF) standard for metamodels, the MOF Query/View/Transformation (QVT) language for model-to-model transformations, and the MOF Model to Text Transformation (MOFM2T) language for code generation. The existence of numerous competing methodologies created confusion among potential users, educators, and students, complicating the learning and application of MDWE techniques.

"The problem of the adoption of model-driven development (MDD) has been widely discussed in the general software domain; specifically, many studies show that the problem is more than technical and involves also cultural, organizational, and other kind of issues. While UML has already more than 20 years of life and has been practically the unique object-oriented modeling language since its birth, the myriad of Web modeling languages (briefly presented in this paper) have in some way hindered adoption. Despite the fact that WebML has been the most popular MDWE approach being used in the industry, IFML is far from being mainstream. Indeed, this standard still needs to gain a supporting community. Additionally, since the Web development activity is much more interdisciplinary than general software development, there are a huge number of developers who barely program, and of course do not model. Despite the empirical evidence presented in several works about how software quality and team productivity improve when using MDWE approaches than code-based ones, developers ignore the benefits of adopting an MDWE approach. Also, the huge variety of programming languages and frameworks make the adoption of MDWE approaches difficult since developers tend to think about modeling techniques as to an obstacle for creativity, while in fact it is the contrary. One can reasonably argue that the lack of good tool support and the absence of user communities have an important place in this problem."(Rossi, Urbietta et al., 2016)

Granada et al. (2015) presents an in-depth analysis of visual meta modeling languages and presents that one of the primary shortcomings of existing visual notations is the lack of scientific rigor in their design. As a result, proper communication among stakeholders, including developers, designers, and business analysts, can be hindered, leading to potential misinterpretations and errors. This issue is further exacerbated by the increased complexity of modern web applications, making visual modeling more challenging.

The complexity of modeling modern web applications is another significant is-

sue. As web applications have evolved, their complexity has increased, making visual modeling more challenging. Existing notations often struggle to adequately represent the intricate relationships and data flows in these applications. This complexity can overwhelm users, making it difficult to create, interpret, and maintain accurate models. Additionally, visual syntax elements in DSLs are not always standardized or consistently applied, resulting in symbol redundancy, overload, excess, and deficit. For example, in WebML, there are instances of symbol redundancy (multiple symbols for the same concept), symbol overload (one symbol for multiple concepts), and symbol excess (unnecessary symbols). This inconsistency can lead to confusion and misinterpretation of models, reducing their effectiveness.

Key recommendation of this work by Granada et al. (2015) includes integration of the Principle of Semiotic Clarity, Semantic Transparency and the Principle of Dual Coding recommends supplementing visual symbols with descriptive text to provide additional context and improve overall readability.

This captures the shortcomings of visual notations in modeling languages like WebML and further supports the premise of this research that visual notations can become overcomplicated. Although the exact recommendations of this publication are directly inapplicable to this research, the insights provide a strong foundation. Implementing these principles can lead to better communication, reduced misinterpretations, and more efficient modeling processes, ultimately supporting the development of complex web applications.

The paper "MODE: A Tool for Conceptual Modeling of Web Applications" by Bochicchio and Fiore (2005) introduces MODE, a conceptual modeling tool tailored for web applications. It addresses the need for integrated design methods that blend hypermedia's navigation features with traditional information systems' transactional capabilities. MODE helps designers manage complex web application features, from user navigation to transactional flows, through a structured approach.

Key Takeaways Relevant to a Meta Modeling Language Integration Challenge from this work includes

**Customization and Flexibility:** Importance of user profiling and contextual adaptation, essential in personalizing the experience. For a meta modeling language, flexibility in accommodating different user types, access levels, and customization preferences is crucial.

**Structured Design Approach with modeling from the following perspectives:**

**Information Design:** Defines data organization, with separate schemas for navigation and access.

**Navigation and Publishing Design:** Focuses on user navigation paths and presentation structures.

**Operational Design:** Models all user-interactive functions, extending beyond simple navigation to include complex transactional processes.

For a meta modeling language, structuring these different aspects is critical to offer a coherent and unified approach that guides the user through each phase of design and this approach shows a unified and simplified approach than the older approaches mentioned above.

Another unique insights obtainable from this work is that meta modeling can unify different design paradigms (e.g., data-centric, user-centric, transaction-centric) into a flexible textual framework. This versatility would enables designers to adjust models to suit complex web environments and evolving design methodologies.

The paper "Code Generator Development to Transform IFML (Interaction Flow Modeling Language) into a React-based User Interface" by Rohma and Azurat (2024) explores a model-driven approach to user interface (UI) development by employing Model-Driven Software Engineering (MDSE) and Software Product Line Engineering (SPLE) principles. MDSE centers on using high-level models to drive the entire development process, making abstractions like UI models pivotal. In this study, an IFML-based UI generator was developed to transform Interaction Flow Modeling Language (IFML) diagrams into React-based code. This transformation is facilitated by Acceleo within Eclipse IDE, where specific transformation rules convert platform-independent UI models into functional React components.

Key elements of IFML, such as ViewContainers (which represent website pages), ViewComponents, DataBinding, and form elements, are mapped to their React counterparts, thereby automating UI code generation. This enables developers to focus on design and functionality at an abstraction level, with the generator managing the code transformation. The UI generator also supports static page management and customization, making it versatile for diverse UI needs.

Evaluation results highlight that the tool can produce a functional website and was qualitatively rated based on six SPLE-relevant criteria. This approach points toward a promising direction in meta-modeling, web engineering, and modern frameworks by offering efficient pathways for creating adaptable, reusable UIs. Further insights suggest that MDSE and IFML have strong potential in streamlining web application development, particularly when integrated with modern JavaScript frameworks like React.

## **2.3 Web 3.0 and the Semantic Web**

Although this research does not focus on the implementation details and specific standards of semantic web technologies, it is rooted in the underlying philosophy and emphasizes ensuring compliance as specified in the scope. Therefore, the foundational concepts of the semantic web have been explored in depth.

Halpin (2006) examines the increasing prevalence of philosophical concepts such as reference, identity, and meaning on the Web. The author has explored how the architecture of the Web converges with classical philosophical problems, particularly through the Semantic Web initiative, which has sparked an "identity crisis" by using URIs for both "things" and web pages. The paper discusses solutions proposed by the W3C and analyzes the problem of reference through the lenses of Russell's direct object theory and Kripke's causal theory, proposing new URN spaces and Published Subjects.

It further posits that while a full notion of meaning, identity, and reference may be achievable, practical implementations and standards remain challenging. It evaluates the direct theory of reference and the potential benefits of content negotiation and RDF to clarify the intended "real-world" domain of URIs citing the original work on the semantic web by T. Berners-Lee et al. (2001). The paper also addresses the evolving nature of identity and representation on the Web, suggesting that the distinction between non-information resources and information resources may diminish over time.

It suggests that the Semantic Web might naturally evolve from social software as the need for open and machine-readable data increases. The notion of "semantic

holism" is briefly discussed, highlighting the challenges it poses for both machines and humans. The paper concludes that philosophical engineering, grounded in a deep understanding of logic and philosophy, could play a vital role in realizing a "Web of Meaning," despite the open problems and human limitations involved.

This concept is further explored in response to an issue raised by the semantic web community from T. Berners-Lee (2003) where it is stated that The URI specification defines URI syntax and establishes that each URI identifies a single resource. RDF documents use URIs to identify things and their relationships, where an RDF statement Subject-Predicate-Object "S-P-O" indicates that the binary relation P holds between entities S and O, all represented by URIs. The HTTP specification supports URIs with delegated ownership, publication, and retrieval of information resources which OWL provides a vocabulary for RDF documents to describe RDF properties. Also, it is highlighted that W3C Technical Architecture Group (TAG) advocates for dereferenceable URIs and the publication of relevant information. This architecture ensures that recipients of an RDF statement can dereference P to obtain human-readable or machine-readable information about the asserted relation. The architecture mandates that each URI has a single meaning, authoritatively defined by its owner, preventing misuse from altering its meaning. Using a URI in RDF commits to its ontology. The following paragraphs dive into more implementation-level technical aspects of the semantic web as presented in the literature.

"At the heart of the Semantic Web are several core technologies: RDF, SPARQL, and OWL. RDF (Resource Description Framework) provides a flexible way to represent information about resources in a graph structure using triples. SPARQL, on the other hand, is a powerful query language designed specifically for querying RDF data. Finally, OWL (Web Ontology Language) enables the creation of complex ontologies, allowing developers to define relationships and constraints within their data models." (Hebeler et al., 2009). This book offers an in-depth exploration of implementation techniques for the said technologies, which are not covered in this discussion.

The popular layer cake model of Semantic Web technologies, as illustrated by Heitmann (2007) can be seen in figure 3. This model is a W3C accepted standard and serves as a foundational reference in the understanding and development of Semantic Web applications.

The work by Heitmann (2007) proposes several architectural patterns for Semantic Web applications. These patterns include semantic viewer, semantic portal, semantic annotation, semantic repository, semantic authoring, web application environment, and desktop application environment.

The theoretical capabilities of the Semantic Web has been identified through a review of the influences, design principles, and standards related to the Semantic Web. Complementing this theoretical basis, an empirical survey was conducted on 50 applications that utilize Semantic Web technologies. This survey addressed key challenges in software engineering for the Semantic Web, providing a practical perspective on the proposed patterns.

Each pattern is described in terms of user interfaces and back-end components, with examples from the survey demonstrating the implementation of the architecture of each pattern. Additionally, a pattern language for Semantic Web applications is introduced. This language outlines how to assemble an application by

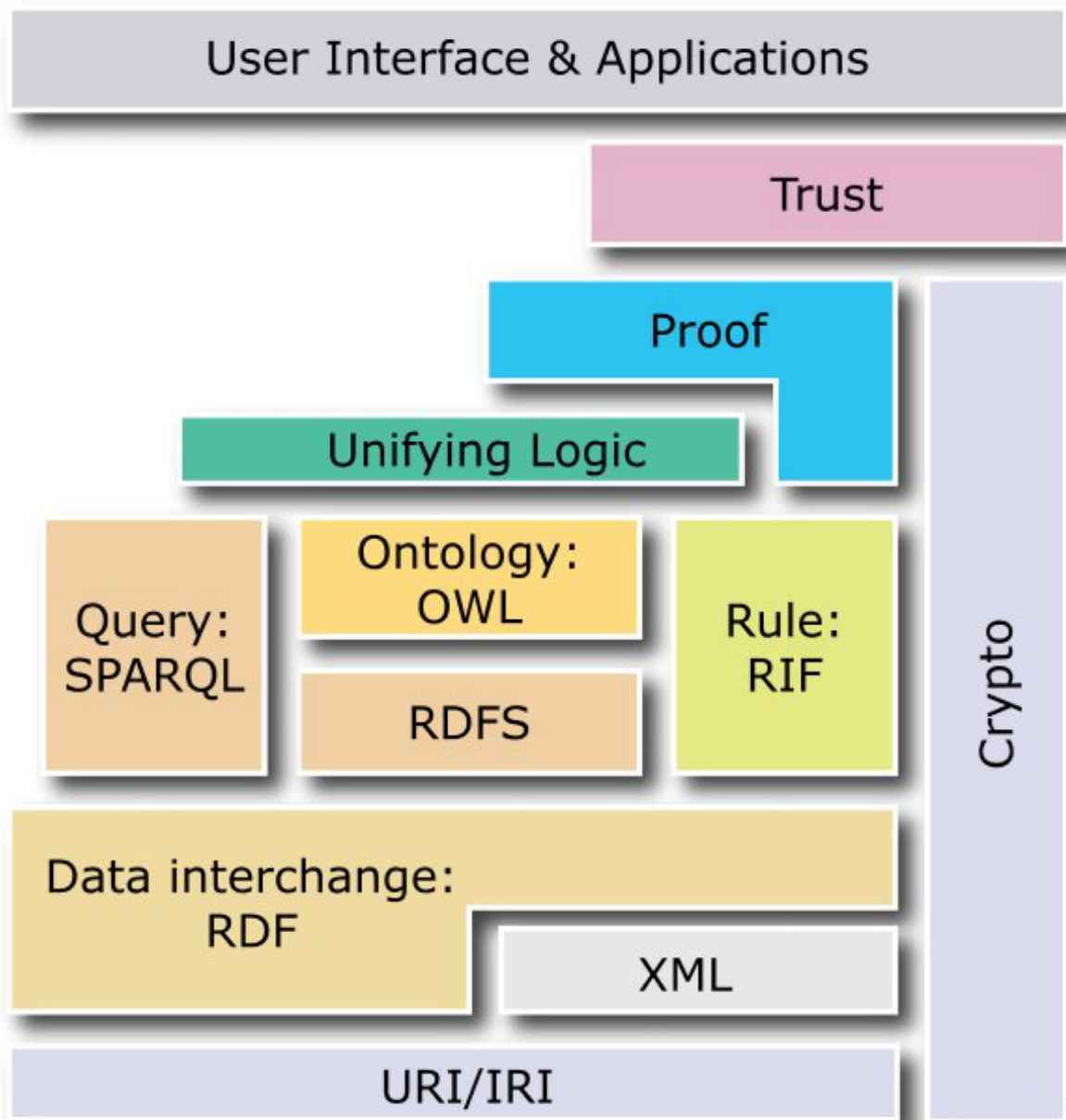


Figure 3: Semantic Web Technology Stack

combining patterns, components, and interfaces.

The various architectural patterns listed provide an excellent basis for semantic web applications, though they are not directly relevant to the scope of this research. However, the concepts of user interfaces and back-end components derived from these patterns provide a useful framework for structuring and designing robust web applications which is a goal of this research.

### **2.3.1 Semantic Web Vocabularies and Knowledge Bases**

Schema.org represents a significant advancement in enhancing web pages with machine-understandable information that can be processed by major search engines to improve search performance. The work by Patel-Schneider (2014), offers a comprehensive analysis and formal semantics for schema.org, addressing its incomplete and sometimes contradictory definitions.

Schema.org is fundamentally a collection of schemas, which webmasters use to mark up their pages in ways recognized by major search providers such as Bing, Google, Yahoo!, and Yandex. This markup is critical because it allows search engines to improve the display of search results, making schema.org a vital component of the modern web. Unlike other languages used for representing information on the web, such as RDF and OWL, schema.org provides a simpler taxonomy and set of ontological specifications organized into a generalization hierarchy.

Schemas are primarily provided as English text on various web pages, with partial mappings into RDF and OWL. However, RDF mapping uses non-RDFS properties like `schema:domainIncludes`, making it challenging to determine the full meaning of schema.org constructs.

Patel-Schneider's paper fills these gaps by offering a thorough basis for schema.org, proposing a complete and coherent version of what schema.org should be as well. It provides a pre-theoretic analysis, an abstract syntax, and a formal model-theoretic semantics for schema.org.

For this research, adhering to schema.org's framework provides an adequate basis for ontologies. It ensures RDF compliance while maintaining a cohesive and well-defined ontology vocabulary, which is crucial for the accurate representation and retrieval of e-commerce data.

In the field of e-commerce, the annotation of products and services on the web using Semantic Web technology has shown considerable promise. Further diving into specific ontologies for e-commerce, according to Hepp (2008), there has been a significant advancement in developing ontologies for various types of products and services, such as eClassOWL. However, these advancements alone are not sufficient to meet the representational requirements needed for e-commerce on the Semantic Web.

Hepp (2008) further identifies the lack of an ontology that can describe the relationships between web resources, offerings, legal entities, prices, terms and conditions, and product and service ontologies. He presents the GoodRelations ontology, which addresses this gap by providing a comprehensive framework to describe typical business scenarios involving commodity products and services. For instance, it allows for the specification that a particular website offers mobile phones of a specific make and model at a given price, that a piano maintenance service is available for pianos under a certain weight, or that a car rental service operates from

multiple branches.

This ontology aims to enhance the ability of consumers and businesses to find suitable suppliers by using detailed and structured product and service descriptions. Despite the potential of the GoodRelations ontology, this research will focus on using schema.org to maintain scope control. Schema.org provides a widely-adopted vocabulary for structured data on the internet, which is essential for the practical implementation of e-commerce applications. This ontology holds the potential for future expansion of this research through the incorporation of more specific vocabularies.

## 2.4 Meta Programming and Compiler Design

This proposal on the Low-Level Virtual Machine (LLVM) by Lattner and Vikram S. Adve (2004b) aims to standardize the complex process of translating source code into machine code. In LLVM, high-level source code is represented in a language-agnostic form known as Intermediate Representation (IR). This abstraction allows different programming languages to share tools for analysis and optimization before being converted into machine code specific to a particular architecture.

A compiler built with LLVM can be divided into three main components: the front-end, the middle-end, and the back-end. The front-end parses the source code using lexical analysis, transforming it into IR. The middle end then analyzes and optimizes this IR. Finally, the backend converts the IR into native machine code.

To build a compiler using this system with C++, the process involves several steps. First, a lexer is created to scan the raw source code and break it into a collection of tokens such as literals, identifiers, and keywords. Next, an Abstract Syntax Tree (AST) is defined to represent the structure of the code and the relationships between tokens. Each node in the AST is typically implemented as a separate class.

Subsequently, relevant LLVM primitives are imported to generate the IR. Each type in the AST is mapped to a public method called `codegen`, which returns an LLVM value object. This object represents a single assignment register, a variable that can only be assigned once by the compiler.

IR primitives, unlike assembly language, are independent of any specific machine architecture, which simplifies the process for language developers. The LLVM optimizer performs tasks such as dead code elimination. Finally, the backend module takes the optimized IR and emits object code, which is used for final machine code generation.

While the initial focus of this promising compiler architecture was primarily on machine code generation, the evolution of the architecture and its surrounding ecosystem has expanded its capabilities. It is now feasible to implement a source-to-source compiler, or more precisely, a transpiler by creating a custom backend as documented by LLVM Documentation (2024). The development of such a transpiler constitutes the objective of the proof-of-concept phase of this research.

Looking into more relatively recent research with LLVM which aligns with the goals of this research the following work is noteworthy. Zakai (2011) Emscripten, an LLVM-to-JavaScript compiler, enabling the execution of code written in languages like C, C++, and Python on the web. These studies collectively highlight LLVM's versatility in code generation and optimization across different program-



ming languages, suggesting its potential applicability for generating TypeScript code from custom languages. Nagaraj et al. (2020) has developed a compiler for Scilla, a smart contract language, targeting LLVM-IR, achieving significant performance improvements. Racordon (2021) has provided a tutorial paper on translating abstract syntax trees to machine code using LLVM, focusing on LLVM IR and its transformation into optimized machine code with a sample language, named Cocodol.

Darius (2010) in "Expectations for a Fourth Generation Language" paper notes that many 4GLs process code through an interpreter that links to compiled code. Code generators expand higher-level code into lower-level code for processing or compiling.

Expectations from 4GLs can be categorized under productivity, system performance, ease of use, and functionality. The appropriate mix of language types in a 4GL depends on current hardware, software technology, and the developer's experience. This author further suggests scoring a 4GL based on how well it meets specific expectations, which should be weighted according to the needs of the programming environment. Understanding the different types of 4GLs is essential for making informed decisions when choosing a language, balancing your needs with the available options.

Darius (2010) mentions that direct compilers for 4GLs are costly and complex, which could be a barrier to when it comes to the evolution of the proposed language, which could be taken into account in future research. However, with modern tools like LLVM, this complexity could be significantly reduced.

## **2.5 Alternatives including Artificial Intelligence(AI) based approaches**

The paper by Kaluarachchi and Wickramasinghe (2023) offers a systematic literature review on automatic website generation based on 68 sources, categorizing the approaches into three primary strategies: example-based, mock-up-driven, and artificial intelligence-driven. Each of these strategies represents a distinct methodology in achieving the automation of web design, addressing different aspects of the design and development process.

Mock-up-driven website generation revolves around transforming visual representations of a website, such as wireframes or mock-ups, into functional code. The core challenge in this strategy is pointed out as the accurate translation of high-fidelity web application designs into functional and responsive interfaces.

The example-based approach to website generation has gained popularity among non-professional designers and developers who rely on pre-designed templates to create websites. This approach is particularly appealing because it lowers the barrier to web development, making it accessible to individuals with limited technical expertise. However, the authors note that the customization options available within these templates are often limited.

Artificial intelligence (AI) has become a transformative force in web development, driving the automation of website design through intelligent algorithms. AI-driven website builders ask users a series of questions to determine their preferences regarding layout, color schemes, and content, then automatically generate a website tailored to these preferences. However, the primary challenge in this

approach is identified to be the improvement of the AI's ability to understand and interpret user preferences accurately, especially in cases where the user's requirements are complex or ambiguous. The conclusions drawn by the authors in this regard appear to be significantly influenced by the findings of Muthazhagu and B (2024).

In a subsequent work by Kaluarachchi and Wickramasinghe (2024) presents a proof of concept utilizing the AI approach for webpage generation. This process involves three main steps: GUI element detection, classification, and code generation. First, image processing techniques are used to detect atomic web GUI elements from a mock-up design artefact of a real-world website. Then, a Convolutional Neural Network (CNN) is trained to classify these extracted web GUI elements into domain-specific categories, such as headings, paragraphs, and images. Typically, a Graphical User Interface (GUI) is represented in code as a hierarchical tree, with nested elements constructing a tree structure. A recursive algorithm is proposed in this approach to construct the appropriate Document Object Model (DOM) hierarchy for a website by recursively grouping classified web GUI elements. Finally, the constructed DOM is converted into "native code".

Although the direct applicability of the above work is beyond the scope of this research, it sheds light on the potential for further automation that could be integrated into building a more extensive and comprehensive ecosystem for automated web application generation.

Moving ahead with the prospect of using AI-based codegeneration, in recent years, many, especially those who are new to the industry have expressed concerns about whether AI-driven code generation will replace developers and Model Driven Engineering. The following literature provides insights that help address these concerns and curiosity.

The analysis of AI-assisted code generation tools, specifically GitHub Copilot, Amazon CodeWhisperer, and OpenAI's ChatGPT, as presented by Yetiştiren et al. (2023) demonstrates their role in enhancing productivity by automating code generation tasks. However, these tools are not yet fully dependable for producing correct code consistently. The study reveals that ChatGPT, GitHub Copilot, and Amazon CodeWhisperer generate correct code 65.2%, 46.3%, and 31.1% of the time, respectively, indicating a significant margin for error even though the focus is on code completion.

While GitHub Copilot and Amazon CodeWhisperer have shown improvements in newer versions—18% for GitHub Copilot and 7% for Amazon CodeWhisperer—these advancements do not entirely resolve their shortcomings. Additionally, the analysis shows that the average technical debt, considering code smells, varies slightly across the tools: 8.9 minutes for ChatGPT, 9.1 minutes for GitHub Copilot, and 5.6 minutes for Amazon CodeWhisperer.

This data suggests that while these tools offer considerable productivity boosts by automating parts of the coding process, they should not be solely relied upon. Developers still need to review and refine the generated code, especially to address issues related to correctness and maintainability. Burak Yetiştiren et al. argue similarly, pointing out that although code generation tools can reduce time spent on repetitive coding tasks, manual intervention is still essential to correct errors and handle technical debt.

The analysis by Nguyen Duc et al. (2023) emphasizes the transformative role

of AI tools in software development, particularly highlighting how tools such as GitHub Copilot, OpenAI Codex, DeepCode, Amazon CodeGuru, TabNine, Kite, and IntelliCode assist developers in generating code, automating tasks, and improving productivity

Nguyen Duc et al. (2023) further notes that these AI tools excel at addressing specific problems such as code completion, bug detection, and performance optimization, thus streamlining various aspects of software development. They serve as productivity boosters by automating repetitive tasks, providing intelligent code suggestions, and detecting vulnerabilities.

Furthermore, it is crucial to note that these tools primarily solve coding-related problems as presented through the above analysis, rather than tackling more complex challenges like knowledge modeling and meta-programming, which require deeper contextual understanding and reasoning. The paper’s focus remains on generative AI’s capacity for enhancing code generation, but it also acknowledges that broader issues, such as the accurate modeling of knowledge and the integration of AI-generated code into larger systems, remain largely unaddressed.

The essence of the above works establishes that while AI-driven code generation has significantly advanced, tools such as GPT models or automated code generators, serve primarily as assistive technologies. They help automate repetitive tasks, optimize workflows, and even suggest possible solutions, but they lack the deep contextual understanding and problem-solving abilities of a skilled developer. AI-generated code still requires supervision, verification, and integration into larger systems, tasks that demand human insight and expertise.

In the realm of model-driven engineering, the situation is similar. MDE by definition focuses on high-level abstractions to automate code generation from models, increasing productivity and reducing low-level programming efforts. However, this approach complements the developer’s work rather than replaces it. Even before AI-based code generation became mainstream as explained by authors such as Schmidt (2006), automation and AI may assist many aspects of software production, yet developers and architects are still responsible for defining models, managing system architecture, and ensuring that the generated code aligns with the intended functionality. However in the distant future, this may change with the rise of Artificial General Intelligence and Artificial Super Intelligence.

Furthermore, most of these popular tools focus on addressing the problem of code generation, whereas this research tackles knowledge modeling and meta-programming challenges rather than code generation itself. For proof-of-concept (PoC) implementation, LLVM is utilized to handle the code generation aspect. This subsection addresses any curiosity readers may have regarding the distinction between code generation tools and the broader scope of this research.

## **3 Methodology**

### **3.1 Language Design**

The proposed new language is inspired by the philosophy of Web 3.0 and Semantic Web concepts in particular. It leverages many of the benefits derived from this perspective, without being solely confined by existing standards and frameworks.

Instead, it draws inspiration from them to create a more flexible and innovative approach. In terms of standards specific focus is given to RDF compliance through RDFa.

The language introduced in this paper is named LMTH: Language for Modeling Thematic Hyperstructures; a novel meta modeling language for the web.

The following sections provide a detailed overview of the nature and more specific definitions of the language.

### **3.1.1 Nature of the Language**

The following definitions describe the nature and categorization of the language

**LMTH as a Modeling Language:** LMTH is a specialized language created to describe web applications. It allows users to define and manipulate complex data models, making it ideal for applications that require detailed representation of web applications as solutions to real world problems.

**LMTH as a Meta Programming Language:** As a meta programming language, LMTH allows for the creation of programs that can generate or manipulate other programs or themselves. This capability provides a high degree of flexibility and abstraction, enabling developers to write more efficient and reusable code.

**LMTH as a Fourth-Generation Language (4GL):** LMTH could be classified as a fourth-generation language, which means it is designed to be more user-friendly and closer to human language compared to third-generation languages (3GLs) such as javascript. As a 4GL, this language offer higher-level programming constructs and simplified syntax making it easier to create complex applications with less code.

**LMTH as a Domain-Specific Language (DSL):** The term DSL is sometimes used to refer to languages addressing the needs of a narrower business domain. However, meta modeling languages such as WebML and WebDSL are identified as domain-specific languages in the literature. Therefore, it is suitable to categorize LMTH similarly.

Additionally, the following characteristics of LMTH are also noteworthy:

**Textual Language:** Unlike many meta modeling languages mentioned in the literature, which are predominantly visual, LMTH is a textual language. One of the primary reasons for proposing this language is to leverage the benefits of text-based programming, which is familiar to most programmers in their everyday work.

**Declarative:** All the rules defined in the LMTH grammar are declarative. This means that the language focuses on what the outcome should be rather than how to achieve it.

**Turing Complete:** Although the grammar defined in LMTH is declarative and does not fulfil the requirements for Turing completeness, LMTH, being a superset of TypeScript, includes rules that fulfil this. While this has no practical value for the current discussion, it may have future implications, as there is a tendency for declarative languages to evolve into the imperative paradigm over time, as observed in the field of database query languages as presented by TypeDB (2024).

**Less Opinionated:** Being based on TypeScript, LMTH allows for multiple ways to achieve the same outcome. For example, you could define a model directly as a value or assign it to a named variable/constant and then assign it to the model. A clear example of this flexibility can be seen in the user model example.

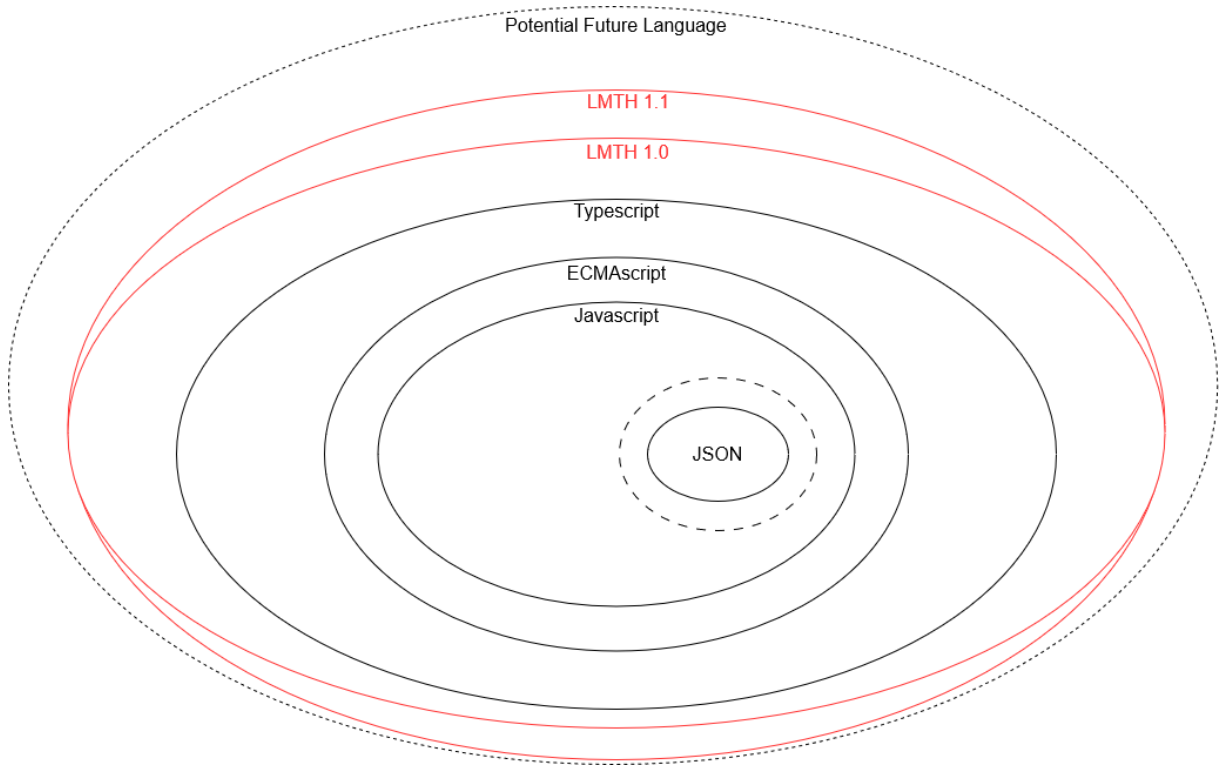


Figure 4: Hierarchical Overlap of LMTH, TypeScript and JavaScript Syntax and Features

The following section details the language definitions. Built on top of TypeScript, LMTH inherits all of TypeScript’s rules. Also note that although the rules are presented in discrete snippets for clarity, they collectively describe a single, cohesive language.

### 3.1.2 Rationale for the choice of TypeScript

Initially, this research was planned to define the language as a superset of JSON. However, the authors realized that many expressive capabilities, such as custom typing, were already available in TypeScript, eliminating the need to reinvent the wheel. This is illustrated alongside other related grammar in 6.

Defining LMTH as a superset of TypeScript, offers several advantages:

**Familiarity:** Although not a W3C standard, TypeScript is widely used in the industry and well-known by many developers, which reduces the learning curve for those adopting LMTH.

**Reduced Development Effort:** The availability of TypeScript grammar definitions for LLVM as open source significantly lowers the development effort required for the transpiler.

**IDE Support:** TypeScript has strong support from various IDEs, eliminating the need to develop new syntax highlighting plugins. However, further enhancements could be developed in the future to provide additional support in this regard.

**Modular Code Organization:** TypeScript allows developers to easily organize code into modules across multiple files, which is beneficial for maintaining complex projects with enhanced DX.

## 3.2 Language Specification

This section outlines the proposed grammar of LMTH, serving as a comprehensive guide for developers building solution web applications, as well as for compiler developers. Additionally, it functions as a reference for researchers who wish to extend or derive new versions of the language. While the content is valuable to both academics and practitioners, given that this work is a research thesis, there is a stronger emphasis on its academic and research perspective. A separate public website, dedicated exclusively to the technical aspects and future developments, is available, with links provided under Appendix A.

When defining the grammar of the language, two leading established standards are considered, as presented by Sebesta (2018) in "Concepts of Programming Languages": Backus-Naur Form (BNF) and Extended Backus-Naur Form (EBNF). EBNF notation is chosen due to its relative simplicity, whereas BNF tends to be more verbose. Additionally, it is important to note that various sources present different variants of EBNF notation. In this thesis, the conventions provided in the aforementioned text have been adopted.

Since LMTH is a superset of TypeScript, the primitives and syntax of TypeScript are not reiterated here. This document assumes the syntax of TypeScript major version 5. Given that LMTH is dependent on the syntax of only a subset of well-established TypeScript primitives, future major versions are less likely to introduce breaking changes. However, if you are referencing this document a significant time after its publication, it may be necessary to consider updates to account for any changes in future TypeScript versions.

Two versions of the novel language are presented here: LMTH 1.0 and LMTH 1.1. The accompanying default transpiler is currently capable of compiling models that are compliant with the LMTH 1.0 grammar. Additional specific rules, identified as necessary, have been introduced in LMTH 1.1; however, the transpiler is not yet able to generate code compliant with these updates. Given that LMTH is forward-compatible within the major version, models developed based on LMTH 1.0 can be extended to comply with LMTH 1.1 in the future. This versioning is based on the definitions by Preston-Werner (2013).

Meta models created with LMTH consist of two major segments. One is the common domain model, included as part of the standard library under the module named 'model-of-reality.' This component represents the problem space. The other is the solution model, which developers are responsible for creating. This represents the solution space. Each of these segments are discussed in detail in the following sub sections.

### 3.2.1 Model of Reality

It is essentially an extensible type library written in typescript belonging to various problem domains, which provides an ontology vocabulary.

The domain model represents a structured abstraction of reality, designed to represent the problem space of various scenarios accurately, capturing complex relationships and entities within a given context. Modeling such a comprehensive representation of reality is inherently challenging due to the intricate and dynamic nature of real-world systems.

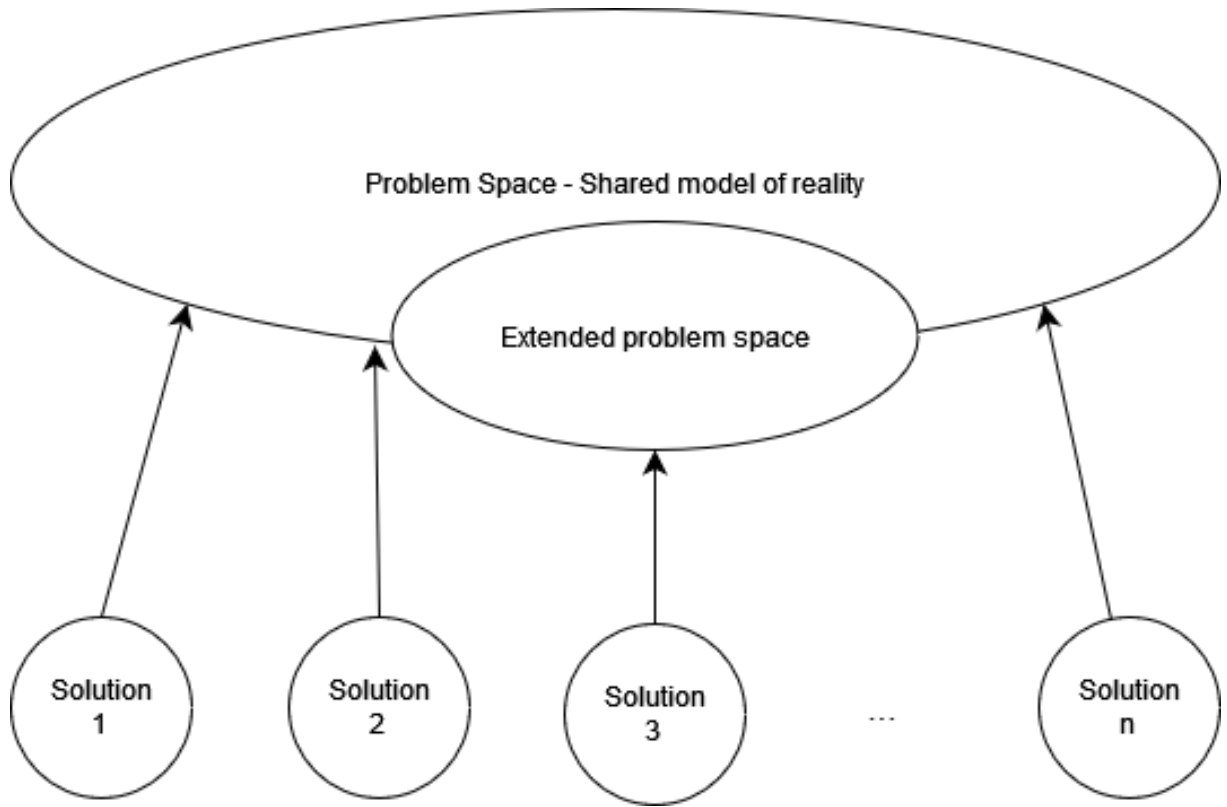


Figure 5: Domain model as a common basis

This vocabulary of ontologies provides a standardized structure applicable to a wide array of web applications. A key aspect of its efficacy lies in its adherence to Schema.org schemas. Schema.org vocabularies are widely adopted by industry leaders such as Google, Microsoft, Pinterest, and Yandex, enabling rich and extensible user experiences across various platforms.

The domain model is included as a type library within the LMTH Standard Library, in a module named 'model-of-reality'. This module encompasses a set of extensible types designed to facilitate broad applicability and customization.

Schema.org features approximately 1,000 schemas, but this model currently implements 35 schemas as a proof of concept. This initial implementation demonstrates the potential for scalability and extensibility, allowing for the incorporation of additional schemas to further enhance the model's comprehensiveness and utility as the novel language evolves. For schemas which are not concretely implemented as a type within 'model-of-reality' module a placeholder type is defined that accepts values of 'any' type. This ensures consistency and forward compatibility as the language evolves to include the definitions of more schema as types.

Attributes within this model are inheritable based on Object-Oriented Programming (OOP) principles, contributing to a vast array of relevant attributes being available for use in a cohesive standard. Developers utilizing the library can use these type directly when defining the data model or extend them to better fit their specific needs. This is demonstrated in the subsequent sections. Given that the model is based on TypeScript, it allows developers to use existing classes of the library and add additional attributes as required to suit particular scenarios. While aliasing attributes is currently possible at the UI level, it is not supported at the data modeling level.

None of the existing modeling languages provides a systematic method for distinctly separating the solution space from a shared problem space as an abstraction. While this concept is well-established within the field of knowledge engineering, model-driven web engineering lacks languages and tools that incorporate such an approach. Even theoretical proposals in the literature often focus on applying adapters or bridges rather than developing a unified model. This unified model is a critical factor in achieving compliance with Web 3.0's semantic web standards, thereby facilitating the creation of more seamless and interoperable web applications.

In this context, the model is maintained as a type library rather than being implemented through interfaces or classes. Importantly, no special new grammar rules are required, as the approach leverages TypeScript. Although the model may appear verbose due to its foundation in the schema.org library, it is important to note that not all fields will be applicable in every scenario.

In the future, the compiler can be optimized to omit unused fields without compromising the essential semantic descriptions. This optimization would reduce the size of the generated code, thereby enhancing efficiency.

**Example of a Semantic Type for Event:** 'Thing' is also shown for reference

```

1 type SemEvent = Thing & {
2   about?: Thing
3   actor?: Person
4   aggregateRating?: AggregateRating
5   attendee?: Organization | Person
6   audience?: Audience
7   composer?: Organization | Person
8   contributor?: Organization | Person
9   director?: Person
10  doorTime?: DateTime | Time
11  duration?: Duration
12  endDate?: Date | DateTime
13  eventAttendanceMode?: EventAttendanceModeEnumeration
14  eventSchedule?: Schedule
15  eventStatus?: EventStatusType
16  funder?: Organization | Person
17  funding?: Grant
18  inLanguage?: Language | Text
19  isAccessibleForFree?: Boolean
20  keywords?: DefinedTerm | Text | URL
21  location?: Place | PostalAddress | Text | VirtualLocation
22  maximumAttendeeCapacity?: Integer
23  maximumPhysicalAttendeeCapacity?: Integer
24  maximumVirtualAttendeeCapacity?: Integer
25  offers?: Demand | Offer
26  organizer?: Organization | Person
27  performer?: Organization | Person
28  previousStartDate?: Date
29  recordedIn?: CreativeWork

```



```

30   remainingAttendeeCapacity?: Integer
31   review?: Review
32   sponsor?: Organization | Person
33   startDate?: Date | DateTime
34   subEvent?: Event
35   superEvent?: Event
36   translator?: Organization | Person
37   typicalAgeRange?: Text
38   workFeatured?: CreativeWork
39   workPerformed?: CreativeWork
40 }
41
42 type Thing {
43   additionalType?: Text | URL
44   alternateName?: Text
45   description?: Text | TextObject
46   disambiguatingDescription?: Text
47   identifier?: PropertyValue | Text | URL
48   image?: ImageObject | URL
49   mainEntityOfPage?: CreativeWork | URL
50   name?: Text
51   potentialAction?: Actios
52   sameAs?: URL
53   subjectOf?: CreativeWork | Event
54   url?: URL
55 }

```

The choice to use the intersection type ('&') rather than union type ('|') is based on the fact that if there are any mandatory fields from a supertype as well as a subtype both must be enforced and the specific sports-related properties encapsulated in the subtype. For instance in the given example it guarantees that any object typed as 'SportsEvent' will have a comprehensive set of attributes, encompassing general event details like 'id', 'name', and 'startDate', as well as sports-specific details such as 'awayTeam', 'competitor', 'homeTeam', and 'sport'. This is crucial for maintaining data integrity and completeness, as it prevents the creation of objects that lack essential information.

On the other hand, using a union type would work however it would allow the creation of objects that only fulfill one of the types, potentially leading to incomplete data structures that lack critical sports-specific or event-specific properties. Therefore intersection type was chosen for subtyping within the type library.

```

1 type SportsEvent = SemEvent & {
2   awayTeam?: Person | SportsTeam
3   competitor?: Person | SportsTeam
4   homeTeam?: Person | SportsTeam
5   sport: string | URL
6 }

```

### 3.2.2 Solution Model

#### Background :

The solution model serves as the root level model for individual applications and the aliases 'application' or 'app' can also be used to define this model. This model represents the solution space. Nested within this model are sub models that cover different aspects of the application solution. These nested models are discussed in detail later in this section. Developers who intends to develop a web application as a solution to a real world problem would get started by defining this model.

**Model Definition :** There are three types of attributes: **Required**, **Optional**, and **Custom**.

#### Required Attributes:

By definition, required attributes must be initialized. For the solution model, the required attributes include:

- **name:** The name of the solution application.
- **vocabulary:** There are various standards and vocabulary providers. This research is primarily based on `schema.org`, but flexibility is provided to change this in the future.
- **language:** Multiple languages are supported here, similar to the "inLanguage" definition in `schema.org`. This attribute can be a string or a language code. Automatic translation is not considered; this attribute is mandatory to ensure descriptiveness.

#### Optional Attributes:

Optional attributes may be initialized, but are not required. These include:

- **title:** A human-friendly title for the application.
- **version:** It is recommended to use semantic versioning standards. However, this is not enforced by the grammar in versions 1.0 and 1.1.
- **author:** This could be an individual or an organization.

In addition to the explicitly defined attributes above, the following attributes, based on `schema.org/Website`, are also included in the language with appropriate code mapping templates. The data types and rules align with the `Schema.org` definitions.

Once a `solutionModel` object is created, it should be set as `solution`, `application`, or `app` in `solution.ts`. Compilation will start from this point and recursively reference the nested models, effectively building a parse tree during implementation. Each rule may optionally end with a semicolon, although this is not explicitly specified for each rule. Also note that all primitive names are case-sensitive.

#### Grammar

$\langle LMTH \rangle \quad ::= \text{SolutionModel} = \{ \text{FieldList} \}$

$\langle FieldList \rangle \quad ::= \text{Field} (, \text{Field})^*$

$\langle \textit{Field} \rangle$  ::= (GeneralField | AuthorField)  
 $\langle \textit{name} \rangle$  = name:, string;  
 $\langle \textit{vocabulary} \rangle$  = vocabulary:, string  
 $\langle \textit{language} \rangle$  = language:, (string | languageObject)  
 Rules for the optional fields  
 $\langle \textit{title} \rangle$  ::= title:, string;  
 $\langle \textit{version} \rangle$  ::= version:, (semVersionObject | string)  
 $\langle \textit{author} \rangle$  ::= author:, (organizationObject | personObject | string)  
 alias : SolutionModel == ApplicationModel, SolutionModel = App, Solution-Model

**Example :**

```

const application = {
  name: "Score Board",
  title: "Optionally overrides name – ${{pagename}}",
  version: "0.0.0",
  vocabulary : "Schema.org",
  author: {
    name : '',
    link : ''
  },
  data: [], //including static
  processes : [],
  ui : [],
  users : []
}
  
```

**Special Notes :** If custom attributes are defined, the corresponding templates must be provided for code generation; otherwise, they will be ignored during the compilation stage. This pattern recurs for nested sub-models within the application model.

It is important to inform developers who may reference schema.org or WebML that the exact models used here are not direct replicas but rather inspired by them. Therefore, it is crucial to note the differences outlined below to avoid any confusion.

A notable deviation from the schema.org ontology for 'WebSite' is the omission of the "audience" attribute, which has been relocated to the user model. Permissions are also associated with this attribute, as specific restrictions, such as age-based filtering, are more logically placed within the user model. All other models are nested within the solution model and adhere to the format of Required, Optional, and Custom attributes.

These fields define the details necessary for generating the web application. Every other model required for this purpose is nested within the solution model and is part of the overall grammar. The following subsection discusses the sub-models proposed in this version of the language and potential extensions for future iterations.

### 3.2.3 Sub Models of the Solution Model

This section explores the sub-models that must be defined within the solution model, which are then utilized to generate code for the web application solution. Each sub-model addresses a distinct concern and may have interdependencies with other models, which can be specified according to the applicable rules. These sub models are :

- Data Model
- Process Model
- UI Model
- User Model
- State Model

In LMTH, certain models found in other alternatives are incorporated, though in a modified form within the models described above. This includes

- Business Logic Model,
- Presentation Model
- Interaction Model
- Navigation Model
- Hypertext Model

However, the following models which are mentioned within literature on MDWE are excluded from the scope of this thesis and could be considered for future research:

- Internationalization and Localization Models
- Personalization Model
- Deployment Model
- Adaptation Model
- Context Model
- Security Model
- Integration Model

### 3.2.4 Data Model

#### Background

The data model is defined as an array of classes. It is crucial to emphasize that this array consists of classes rather than objects; adding objects to the array will result in a compilation error. Classes defined within the model-of-reality, custom classes based on real-world model-of-reality as boilerplate, achieved through the `implements` keyword as well as independent classes can be used in this data model.

For ensuring standard compliance, including semantic web standard compliance, and to fully leverage the capabilities of the Linked Model of the Human Thought (LMTH), it is strongly recommended that each class implements a corresponding type from the model of reality library. Any classes that are not based on the types in the domain model are still valid and do not result in any error but do not ensure full standard compliance at the code generation stage. Using the type library of the domain model guarantees adherence to established standards, promoting interoperability and maximizing the utility and effectiveness of the data model within various applications and systems as intended.

**Free Hanging Classes:** This term refers to classes that do not implement any class within the model of reality. Nominally, this can be avoided by implementing the 'Thing' class, which is the most generic root-level class of the entire library. However, doing so may not add significant real value.

Since the default compiler for LMTH generates Angular framework code for the frontend, the classes are directly mapped to classes in the frontend application. The backend application generated by the default compiler is based on Express.js, where these classes are mapped to Mongoose schemas. If the backend code were generated in a framework that supports TypeScript, such as NestJS, this approach would be even more elegant. NestJS offers a robust, modular architecture that aligns well with TypeScript, providing enhanced type safety and developer productivity. Adopting NestJS for the backend would streamline the development process but also enhance maintainability and scalability, offering a cohesive and efficient development experience across the entire stack. Researchers and contributors with experience in NestJS or other similar typescript based backend frameworks are encouraged to look into this implementation.

Another meaningful and valuable future enhancement to this data modeling approach would be the development of a code generator capable of producing LMTH-compatible data models directly from class diagrams or entity-relationship (ER) diagrams.

Currently, there exists a plugin that facilitates the generation of TypeScript classes, which can be adapted to fit the data model in an LMTH solution with minimal effort. Use of such tools in this context is encouraged and it would significantly reduce manual coding efforts, minimize errors, and promote adherence to LMTH standards, making it an invaluable asset for developers and data architects working within this framework. However efforts should be given to ensure compatibility with the standardized types provided.

There are various proposals in the literature, some of which suggest using ER diagrams in combination with WebML, while others recommend class diagrams. These methods could be employed to automate database generation, with the data model being a property of the solution model.

The data model is typically defined as classes, facilitating a natural representation that is familiar to developers. This approach can even encompass an entire class diagram, which can be mapped accordingly.

Although not a one-to-one mapping, some parallels can be drawn where applicable. Class diagram to LHTH converters may be useful for those who wish to utilize class diagrams in this context.

In this research, the authors experimented with several representations, including custom JSON definitions for each and unique representations. However, these experiences led to a shift toward a TypeScript-based grammar, allowing for the use of existing classes and types without the need for redefinition.

### **Grammar**

```
 $\langle data\_model \rangle \quad ::= \text{data} : [ \langle classes \rangle ]$   
 $\langle classes \rangle \quad ::= \langle class \rangle , \langle class \rangle$   
 $\langle class \rangle \quad ::= \langle identifier \rangle$ 
```

### **Examples**

```
class SpecializedOrder extends Order{  
  extra : string  
}  
  
const application = {  
  data: [Product, Order, SpecializedOrder],  
  ...  
}
```

## **3.2.5 Process Model**

### **Background**

The process model within LMTH defines a set of business processes that can take zero or more inputs and produce zero or one output of specific data types. This model intentionally excludes fundamental operations such as CRUD (Create, Read, Update, Delete) operations, derived data, mapped data, and session-related set and get operations. This exclusion is a key feature of LMTH's simplicity, distinguishing it from its predecessors, which required detailed modeling of each of these operations.

The process model in this context represents the operations or logic component of the solution web application, offering a more abstract view of the various operations involved. However, it's important to note that the detailed inner workings or specific logic of each process are not generated within the scope of this research. Instead, the process model serves as a framework that can be mapped to controllers and function or method stubs, guided by the input and output types required for both backend and frontend execution, or a combination of both, depending on the execution type. This mapping is carried out using predefined templates.

A BPMN-based code generator could potentially be adapted for this purpose. Additionally, it's important to note that the current approach is used to generate

process views that communicate with a REST API using AJAX. It generates a stub for an Angular service on the frontend and corresponding endpoints within the Express.js application on the backend. The data types are consistently maintained from end to end. Additionally, it is possible to exclude the backend endpoint and the corresponding stub by marking the process as a frontend process, where applicable. Alternatives such as WebSockets, WebRTC, webhooks, and similar technologies are not supported at this time. However, the language could be extended in the future to represent these options and update the compiler accordingly, though this would require substantial effort. It's also worth mentioning that these features are not included in LMTH 1.1, highlighting the significant amount of work involved in such an extension. This extension could be an independent research on its own.

Model Definition :

The process representation includes the following key elements:

- **Input:** This represents the data required for the process as an array of data types, which could include classes defined in the data model, such as a Person or Product.
- **Output:** The process produces a single data type as output. This output is returned from the controller and can be mapped to an Process type unit, which is explained in the subsequent section. Similar to the input, this could include classes defined in the data model, like a Person or Product.
- **Precondition Function (preconditionF):** A function that serves as a precondition check before executing the process.
- **Cacheable:** This indicates whether the data should be cached. Currently, if this is set to true, only 10% of interactions will trigger an API call. This behavior could be customized in the future by defining custom caching rules.
- **Controller:** A string representing the categorization of the process. For example, in the current implementation, if the value is set to "Billing", the generated function will be added to a controller named "BillingController".

Grammar (LMTH 1.0) :

```

<process_definition> ::= process = { <process_body> }

<process_body>      ::= executionType : <execution_type>
                        | in : [ <input_list> ]
                        | output : <data_type>
                        | preconditionF = <function>
                        | cacheable : <boolean>
                        | controller : <string>

<execution_type>    ::= 'frontend' | 'backend'

<input_list>        ::= { <io_type> , <io_type>* }

<io_type>           ::= <class_type> | <data_type>

```

Grammar (LMTH 1.1) : Primary differences here is the addition of caching rules function and named parameters.

```

<process_body>      ::= executionType : <execution_type>
                      | in : [ <named_param> ]
                      | output : <data_type>
                      | preconditionF = <function>
                      | cacheable : <boolean>
                      | cachingRules = <function>

<named_param>      ::= { name : <string> , type <io_type> }

```

Example :

```

1 const checkoutProcess = {
2   executionType: 'backend',
3   in: [ Product[], User],
4   output: Invoice
5   preconditionF: function(cart: Cart, user: User): boolean { return
6     true },
7   cacheable: true
8 }

```

### 3.2.6 UI Model - Presentation

In LMTH, there is a sub-model called the UI model. This includes the elements of presentation model and the navigation model from WebML, and it is also strongly inspired by IFML. This section focuses on how the presentation aspects of the solution web application is represented within the meta-model. Following section will focus on navigation aspects.

It is beneficial to consider presentation and navigation as a unified models as they share the same 'concern' as suggested by (Gamage, 2017). An integral part of this model is the concept of a unit. The entire UI model is comprised of various units placed inside views and pages. The concepts of views and pages, which are concerned with the navigation part of this model, are discussed in detail in the next chapter. LMTH proposes five types of units, namely:

- IData Units (three variants)
- Process Units
- Service Units
- CMS Units
- Link Units
- Menu Units

Following grammar rules define the structure of a common data unit. All the data units are extension of UIUnit and therefore shares has the same.



```

<Unit> ::= {, <UnitType>, { <UnitAttrib> }, }

<UnitType> ::= IDataSingle
              | IDataMulti
              | IDataIndex
              | Process
              | Link
              | Menu
              | Service
              | CMS
              | Static

<UnitAttrib> ::= <TemplateList>
                | <ClassList>
                | <StyleList>
                | <ContainerElementAttrib>

<ContainerElementAttrib> ::= containerElement : <String> @v1.1

<TemplateLabel> ::= templateLabel : <String>

<ClassList> ::= classes : <StringArray>

<StyleList> ::= styles : <StringArray>

```

**UnitType** is a mandatory attribute for all units and would determine the manner in which it is rendered on the frontend application.

Each unit is mapped to a template of an Angular component by default in the compiler. It is essential to establish suitable mappings when integrating with new frameworks or handling variations.

The template utilized can be overridden by specifying the optional `templateLabel` attribute. A concrete example of this process is provided in the Implementation chapter.

The `classList` attribute, which is optional, specifies a set of classes to be applied to the unit.

The `styleList` attribute, also optional, allows the application of specific styles or a collection of styles to the unit. At the implementation level, this is typically achieved by adding additional classes. However, presenting it as an alternative attribute offers greater clarity.

In version 1.1, the optional `containerElement` attribute was introduced, permitting the override of the default container type element of the component. By default, this element is a `<div>` or any element specified in the template for unit generation. This attribute enables the use of any semantic element type, such as `<header>`, `<footer>`, `<nav>`, `<article>`, `<section>`, `<aside>`, `<details>`, `<figure>`, `<mark>` or `<summary>`.

## **IData Units**

### **Background**

This would be the most common type of units on most web applications. In WebML

we find Data Units, Multi Data Units and Index Units as publish units and in IFML we find «detail» and «list» units.>

In the context of LMTH, Interactive Data Units (IDUs) are categorized into three distinct types: IData, IMultiData, and IIndex Units. These categories differ primarily in their rendering methods and the content they display. Importantly, IDUs integrate functions that, in WebML, would typically require multiple units. For example, various operation units in WebML, such as those handling edit and delete functionalities, have been streamlined into attributes of the IDUs.

User interactions within the system can trigger AJAX requests to the backend, with the results subsequently displayed in a different unit on the frontend. This process is straightforward, adhering to a simple request-response model without the need for explicit data flow modeling. The system's design is technology-agnostic, with the default compiler automatically generating a corresponding function within the Angular service and an endpoint on the Express backend. This backend code facilitates querying or executing data operations on a MongoDB database. Additionally, the data unit can be configured to render directly on the frontend, bypassing the need for an AJAX request to the backend.

The IDUs in LMTH draw strong inspiration from Publish Units in WebML, particularly in their conceptual approach, though they have been named differently to avoid confusion for those who are familiar with the terminology used in those other meta modeling languages. This model serves to define the elements with which the user interacts, highlighting opportunities for semantic web compliance and accessibility compliance. These aspects are vital in aligning with semantic web standards. The following section outlines the grammar that governs this model.

### **Grammar (LMTH 1.0)**

$\langle conditions \rangle$	$::= \langle string \rangle$
$\langle model \rangle$	$::= \langle string \rangle$
$\langle oid \rangle$	$::= (\langle string \rangle \mid \langle function \rangle)$
$\langle param \rangle$	$::= \langle string \rangle$
$\langle fields \rangle$	$:= [ \text{dataFieldList} ]$
$\langle classes \rangle$	$:= [ \text{stringList} ]$
$\langle dataFieldList \rangle$	$:= ((\langle fieldName \rangle \mid \text{labeledField}) (, ((\langle fieldName \rangle \mid \text{labeledField}))^*$
$\langle labeledField \rangle$	$:= \{ \text{name} : \text{fieldName} \ \text{label} : \text{boolean} \}$
$\langle fieldName \rangle$	$:= \text{string}$
$\langle editMode \rangle$	$:= \text{boolean}$
$\langle preloadBehaviour \rangle$	$:= \text{string}$
$\langle preloadVolume \rangle$	$:= \text{number}$

$\langle \text{editable} \rangle$	$:= \text{boolean}$
$\langle \text{deletable} \rangle$	$:= \text{boolean}$
$\langle \text{deleteModel} \rangle$	$:= \text{boolean}$
$\langle \text{searchable} \rangle$	$:= \text{boolean}$
$\langle \text{searchableFields} \rangle$	$:= [ \text{stringArray} ]$
$\langle \text{sortable} \rangle$	$:= \text{boolean}$
$\langle \text{sortableFields} \rangle$	$:= [ \text{stringArray} ]$
$\langle \text{filterable} \rangle$	$:= \text{boolean}$
$\langle \text{filterableFields} \rangle$	$:= [ \text{stringArray} ]$
$\langle \text{paginationMode} \rangle$	$:= \text{paginationModeChoice}$
$\langle \text{limit} \rangle$	$:= \text{number}$
$\langle \text{links} \rangle$	$:= [ \text{linkUnitArray} ]$
$\langle \text{typeChoice} \rangle$	$::= \text{IDataSingle} \mid \text{IDataMulti} \mid \text{IDataIndex}$
$\langle \text{dataSource} \rangle$	$::= \text{DB} \mid \text{API} \mid \text{SESSION}$
$\langle \text{paginationModeChoice} \rangle$	$::= \text{paginated} \mid \text{scrollable} \mid \text{none}$
$\langle \text{linkUnitArray} \rangle$	$::= \text{linkUnit} (, \text{linkUnit})^*$

Following are the extended grammar rules for future implementation. With these new grammar rules the oid, params and conditions can be dynamically calculated based on a function which allows for further processing of user input or use additional logic in determining the data to be displayed. Additional data could be passed to the data source such as configuration settings or personalization settings. Custom definitions of preload animation and delete modal through a custom function is also facilitated through this. Also custom templates can be defined at the field level.

#### **Grammar (LMTH 1.1)**

$\langle \text{oid} \rangle$	$::= (\text{string} \mid \text{function})$
$\langle \text{conditions} \rangle$	$::= (\text{string} \mid \text{function})$
$\langle \text{param} \rangle$	$::= (\text{string} \mid \text{function})$
$\langle \text{preloadBehaviour} \rangle$	$:= (\text{string} \mid \text{function})$

```

⟨dataSource⟩      ::= { type : 'string' }

⟨deleteModel⟩     := (string | function)

⟨dataSource⟩      ::= { ⟨keyValuePair⟩ , ⟨keyValuePair⟩ }

⟨labeledField⟩    ::= { nameField ( , showLbl)? ( , template)? ( , mappingFunction)? }

⟨nameField⟩       ::= "name" : string

⟨showLbl⟩         ::= "label" : boolean

⟨template⟩        ::= "template" : string

⟨mappingFunction⟩ ::= function

⟨keyValuePair⟩    ::= ⟨key⟩ : ⟨value⟩

⟨key⟩             ::= 'string'

⟨value⟩           ::= 'string' | 'number' | 'boolean' | 'null'

```

linkUnit ::= /\* Define as per the structure of LinkUnit \*/

### Example

The following LMTH code snippet shows the definition of a IData Unit displaying a single item.

```

1 {
2   type: IDataSingle
3   model: "Product"
4   param : prodId
5   dataSource: {
6     type: DB
7   }
8   fields : ['image', 'name', 'vegan', 'description', 'volume', 'size', 'weight', { name : price, label : false }]
9   classes: [ "tst" ]
10  editMode : false
11  preloadBehaviour : "img/ProductSpinner.png"
12 }

```

The following LMTH code snippet shows the definition of a IData Unit displaying a a list of item.

```

1 {
2   type: IDataSingle
3   model: "Product"
4   param : prodId
5   dataSource: {
6     type: DB

```

```

7     }
8     fields : ['image', 'name', 'vegan', 'description', 'volume', 'size
          ', 'weight', { name : price, label : false }]
9     classes: [ "tst" ]
10    editMode : false
11    preloadBehaviour : "img/ProductSpinner.png"
12 }

```

The following LMTH code snippet shows the definition of a IData Unit displaying a list of items with contextual link leading to a IDataSingle type unit. This is known as an index unit.

```

1 {
2     type: IDataSingle
3     model: "Product"
4     param : prodId
5     dataSource: {
6         type: DB
7     }
8     fields : ['image', 'name', 'vegan', 'description', 'volume', 'size
          ', 'weight', { name : price, label : false }]
9     classes: [ "tst" ]
10    editMode : false
11    preloadBehaviour : "img/ProductSpinner.png"
12 }

```

## Process Units

These units adhere to the same structure as `IDataUnits`, with a key distinction in their implementation. Instead of directing requests to a backend endpoint connected to a database, these units send requests to an endpoint associated with a process. While the underlying grammar rules remain consistent with those of `IDataUnits`, the primary difference lies in the absence of an attribute for specifying a data source. This divergence reflects the unique focus of these units on process-oriented operations rather than data retrieval or manipulation from a database.

Whether or not these units call the backend depends on the process definition within the model, rather than needing to be specified at the UI unit level. While this distinction is noteworthy, it simplifies the modeling process by eliminating the need to define backend interactions at the level of individual UI units. This approach highlights a key difference in how these units function compared to `IDataUnits`, emphasizing the process-driven nature of their operations.

Examples :

Process unit involving and end to end process can be seen below :

```
1 {  
2   type: Process  
3   process: "CartQtyUpdate"  
4  
5   classes: [ "counter-btn" ]  
6   preloadBehaviour : "SmallBox.png"  
7 }
```

Process unit involving front-end process can be seen below :

```
{  
  type: Process  
  process : "Checkout"  
  fields : [ 'success', 'total' ]  
}
```

## Link Units

Inspiration is taken from Lithium PHP definition of Link based on the works by RAD (2024) in addition to WebML and IFML representations in this research. However, code generation with Lithium or other PHP frameworks is not possible at the moment, although a new code generator could be developed in the future. This representation was chosen because it is concise and covers the necessary aspects for modeling a link.

Grammar :

```
<LinkUnit> ::= <lid>  
             | <ltitle>  
             | <ltext>  
             | <lurl>  
             | <canonical>  
             | <options>  
             | <target>  
             | <rel>  
             | <generateCode>
```

	$\langle context \rangle$
	$\langle icon \rangle$
$\langle context \rangle$	$:= \langle StringArray \rangle$
$\langle icon \rangle$	$:= \text{string}$
$\langle lid \rangle$	$:= \text{string}$
$\langle ltitle \rangle$	$:= \text{string}$
$\langle ltext \rangle$	$:= \text{string}$
$\langle lurl \rangle$	$:= (\text{string} \mid \text{object} \mid \text{null})$
$\langle canonical \rangle$	$:= \text{boolean}$
$\langle options \rangle$	$:= (\{ \langle linkOptionsList \rangle \})?$
$\langle target \rangle$	$:= \text{string?}$
$\langle rel \rangle$	$:= \text{string?}$
$\langle generateCode \rangle$	$:= \text{function}$
$\langle linkOptionsList \rangle$	$::= \langle linkOption \rangle (, \langle linkOption \rangle)^*$
$\langle linkOption \rangle$	$::= \langle key \rangle := \text{any}$

#### 1.1 grammar

enable carrying additional data along with the link.

$\langle context \rangle$	$:= \langle ObjectArray \rangle$
---------------------------	----------------------------------

### Menu Units

simply a collection of link units and submenus. submenus can recursively go in any number of times. However templates should allow sensible rendering of those in each recursive sub menu.

Predefined links.

$\langle menu \rangle$	$::= \{ \langle mTitle \rangle := \text{string} , \langle links \rangle := [ \langle menuItemList \rangle ] \}$
$\langle menuItemList \rangle$	$::= \langle menuItem \rangle (, \langle menuItem \rangle)^*$
$\langle menuItem \rangle$	$::= \langle linkUnit \rangle \mid \langle menu \rangle$

### CMS Units

A Headless CMS can be connected in a similar manner to data units, maintaining a one-way data flow. While this integration is beneficial, especially for enhanced configuration flexibility, it remains advantageous to keep the CMS separate from the core data units. This separation allows for more refined and adaptable configurations, while still extending the functionality of data units during implementation.

Grammar :

$\langle CMSUnit \rangle$	$::= \{ \langle elementList \rangle \}$
$\langle elementList \rangle$	$::= \langle element \rangle (, \langle element \rangle)^*$
$\langle element \rangle$	$::= \langle source \rangle$ $  \langle contentType \rangle$ $  \langle script \rangle$ $  \langle config \rangle$ $  \langle model \rangle$ $  \langle params \rangle$
$\langle source \rangle$	$:= \text{string}$
$\langle contentType \rangle$	$:= \text{string}$
$\langle script \rangle$	$:= \text{string}$
$\langle config \rangle$	$:= \text{string}$
$\langle model \rangle$	$:= \text{string}$
$\langle params \rangle$	$:= [ \langle paramList \rangle ]$
$\langle paramList \rangle$	$::= \langle param \rangle (, \langle param \rangle)^*$
$\langle param \rangle$	$::= \langle paramKey \rangle \Rightarrow \langle paramValue \rangle$
$\langle paramKey \rangle$	$::= \text{string}$
$\langle paramValue \rangle$	$::= \text{string}$

### 3.2.7 UI Model - Navigation

Inspired by WebML's and IFML's approach, there are Views, then nested in views. Containers called areas maybe inside pages for further organization.

### 3.2.8 Views

A SiteView or simply Views in LMTH is a structured representation of a specific section or mode of a solution web application. It defines how various components, pages, and common elements are organized and presented to the user.

#### Definition

**name :** This field defines the name of the site view. The name field determines the route or path of the site view. For example, if the name is "admin", the default route of the view will be "site\_url/admin" (all lower case for URI standard compliance). If the name is guest or null, it will be set as the root view.

**layout:** This field specifies the layout template that will be used for this site view. This is a pre-defined layout structure that dictates how the content will be arranged on the page. The name "Fluid3" suggests a flexible, responsive design with potentially three primary sections or columns.



pages: This field is an array that lists the pages included in this site view. Each entry in the pages array represents a different page or component that is part of this site view. These pages typically correspond to various sections or functionalities of the web application.

**Grammar:**

```

<siteview>          ::= { <siteview_fields> }

<siteview_fields>   ::= <siteview_field> (, <siteview_field>)*

<siteview_field>    ::= <name_field>
                       | <layout_field>
                       | <pages_field>
                       | <units_field>

<name_field>        ::= name: <string>

<layout_field>      ::= layout: <string>

<pages_field>       ::= pages: [ <page_list> ]

<page_list>         ::= <page> (, <page>)*

<page>              ::= <object_ref>

```

**Examples :** A sample site view can be seen below :

```

1  const guest : SiteView = {
2      name: "Guest"
3      layout : "Fluid3"
4      pages : [ homePage, articleView, player, match, tournament,
5                venue, about ]
6      units: [ header, footer, menu ]
    }

```

### 3.2.9 Pages

In LMTH, pages are individual web pages that can be configured to create a comprehensive and interactive website. The configuration of pages involves defining various attributes, themes, content units, and layout areas to deliver a structured and cohesive user experience. Below are detailed explanations of different aspects of pages in LMTH:

**name:** This specifies the name of the page. In this case, it is the "Home" page.  
**theme:**

**theme :** This denotes the theme applied to the page. In the example below "RoyalPurple" is the name of the theme, which would define the visual style and aesthetics of the page. This is achieved by simply setting the container class.

**home :** A flag indicating whether this page is the home page of the website. Setting this to true designates this page as the primary landing page.

landing: This flag indicates whether the page serves as a landing page. Landing pages are typically the main entry points for users and may have specific content or structure optimized for first impressions or navigation.

units: An array of unit configurations. Units are reusable components or widgets that constitute the content and functionality of the page. Each item in the array represents a different unit such as staticLogo, popularArticles, recentMatches, mainMenu, readMoreLink, sponsorList, and monthEventCalendar. These units can include static content, dynamic content, navigation elements, etc.

grammar : |

$\langle page \rangle$	$::= \{ \langle pagefields \rangle \}$
$\langle pagefields \rangle$	$::= \langle pagefield\_list \rangle$
$\langle pagefield\_list \rangle$	$::= \langle pagefield \rangle (, \langle pagefield \rangle)^*$
$\langle pagefield \rangle$	$::= \langle name\_field \rangle$ $  \langle theme\_field \rangle$ $  \langle home\_field \rangle$ $  \langle landing\_field \rangle$ $  \langle units\_field \rangle$ $  \langle areas\_field \rangle$
$\langle name\_field \rangle$	$::= name: \langle string \rangle$
$\langle theme\_field \rangle$	$::= theme: \langle string \rangle$
$\langle home\_field \rangle$	$::= home: \langle boolean \rangle$
$\langle landing\_field \rangle$	$::= landing: \langle boolean \rangle$
$\langle units\_field \rangle$	$::= units: [ \langle unit\_list \rangle ]$
$\langle areas\_field \rangle$	$::= areas: [ \langle area\_list \rangle ]$
$\langle unit\_list \rangle$	$::= \langle unit \rangle (, \langle unit \rangle)^*$
$\langle area\_list \rangle$	$::= \langle area \rangle (, \langle area \rangle)^*$
$\langle area \rangle$	$::= \{ \langle area\_fields \rangle \}$
$\langle area\_field \rangle$	$::= \langle area\_name\_field \rangle$ $  \langle area\_units\_field \rangle$
$\langle area\_name\_field \rangle$	$::= name: \langle string \rangle$
$\langle area\_units\_field \rangle$	$::= units: [ \langle unit\_list \rangle ]$
$\langle unit \rangle$	$::= \langle object\_ref \rangle$

**Examples :** Following code shows a sample page with several units. The units will be rendered in the given order

```
1    const homePage: Page = {
2        name: "Home"
3        theme : "RoyalPurple"
4        home: true
5        landing: true
6        units: [staticLogo, popularArticles, recentMatches, mainMenu,
7                readMoreLink, sponsorList, monthEventCalendar]
8        areas : [{
9            name : "Match Schedule"
10           units : [staticHeader, upcomingEvents, featuredMatch]
11        }]
12    }
13
14    /*The Units can be defined as follows. More on defining UI units
15       can be found above in the respective section above.*/
16
17    const popularArticles: UIUnit = {
18        type: "IDataMultiUnit",
19        model: "Article",
20
21        conditions: "{ interactionStatistic: { $max: 5 } }",
22
23        dataSource: {
24            type: "DB"
25        },
26
27        fields: ["datePublished", "title", "image", "articleBody"],
28
29        classes: ["article-list", "popular-articles"],
30
31        preloadBehaviour: "img/LoadingArticles.png",
32        preloadVolume: 5,
33        paginationMode: "none",
34        limit: 5
35    }
```

### 3.2.10 Areas

An optional array of area configurations set within pages. Each area is a defined section of the page that groups related units together. Areas help organize the layout and structure of the page.

**name:** The name of the specific area within the page. In this case, "Match Schedule" indicates an area dedicated to displaying match-related information.

**units:** An array of unit configurations specific to the area. Each unit serves a specific function within the area, contributing to the overall purpose of the area.

The following enhancement is proposed for **LMTH 1.1** to customize the rendered container type from div to another semantic type such as aside or footer.

```

<area_field>          ::= <area_name_field>
                        | <area_units_field>
                        | <container_type>

<area_units_field>    ::= units: [ <unit_list> ]

<container_type>      ::= string

```

Examples for areas can be seen in the sub-section above on 'pages'.

### 3.2.11 User Model

#### Precedence Rules :

LMTH proposes the following precedence order for permissions. By default, all permissions are restricted, meaning no access is granted unless explicitly allowed. Here is a detailed explanation of the permission check precedence:

##### Default Denial:

By default, users do not have access to any data, processes, UI units, or site views unless explicitly allowed. This ensures a secure baseline where access must be explicitly granted. Class Reference Bypass:

If a reference to a class (representing data in the data model), a process, or a site view is added, then no permission check is needed.

This means that certain essential or critical references bypass the normal permission checks entirely, granting access as necessary.

##### Allowed Field Check:

If an item (data, process, UI unit, or site view) is in the allowedData, allowedProcesses, allowedUnits, or allowedSiteviews field of the user role, access is granted. This positive list explicitly states what a user can access, overriding the default denial. If it is already allowed for guests, then no permission check is needed.

##### Denied Field Check:

If an item is not in the allowed fields but is found in the deniedData, deniedProcesses, deniedUnits, or deniedSiteviews fields of the user role, access is explicitly denied. This negative list takes precedence over allowed fields to ensure that certain critical or sensitive items remain inaccessible even if other permissions might suggest otherwise.

##### Guest Role Exception:

If the user role is guest, even if an item is in the allowed fields, access is denied if it is also listed in the denied fields. This ensures that guests have the most restrictive access, adhering strictly to the denied fields despite any allowed permissions.

##### Grammar

```

<user_model>          ::= { <fields> }

<user_fields>         ::= <field> (, <user_field>)*

```

$\langle \text{user\_field} \rangle ::= \langle \text{name\_field} \rangle$   
 $\quad \quad \quad | \langle \text{auth\_field} \rangle$   
 $\quad \quad \quad | \langle \text{allowedData\_field} \rangle$   
 $\quad \quad \quad | \langle \text{deniedData\_field} \rangle$   
 $\quad \quad \quad | \langle \text{allowedProcesses\_field} \rangle$   
 $\quad \quad \quad | \langle \text{deniedProcesses\_field} \rangle$   
 $\quad \quad \quad | \langle \text{allowedUnits\_field} \rangle$   
 $\quad \quad \quad | \langle \text{deniedUnits\_field} \rangle$   
 $\quad \quad \quad | \langle \text{allowedSiteviews\_field} \rangle$   
 $\quad \quad \quad | \langle \text{deniedSiteviews\_field} \rangle$

$\langle \text{name\_field} \rangle ::= \text{name} : \langle \text{string} \rangle$

$\langle \text{auth\_field} \rangle ::= \text{auth} : \langle \text{boolean} \rangle$

$\langle \text{allowedData\_field} \rangle ::= \text{allowedData} : [ \langle \text{data\_list} \rangle ]$

$\langle \text{deniedData\_field} \rangle ::= \text{deniedData} : \langle \text{identifier} \rangle$

$\langle \text{allowedProcesses\_field} \rangle ::= \text{allowedProcesses} : [ \langle \text{process\_list} \rangle ]$

$\langle \text{deniedProcesses\_field} \rangle ::= \text{deniedProcesses} : \langle \text{identifier} \rangle$

$\langle \text{allowedUnits\_field} \rangle ::= \text{allowedUnits} : [ \langle \text{unit\_list} \rangle ]$

$\langle \text{deniedUnits\_field} \rangle ::= \text{deniedUnits} : \langle \text{identifier} \rangle$

$\langle \text{allowedSiteviews\_field} \rangle ::= \text{allowedSiteviews} : [ \langle \text{siteview\_list} \rangle ]$

$\langle \text{deniedSiteviews\_field} \rangle ::= \text{deniedSiteviews} : \langle \text{identifier} \rangle$

$\langle \text{data\_list} \rangle ::= \langle \text{data} \rangle (, \langle \text{data} \rangle)^*$

$\langle \text{data} \rangle ::= \{ \langle \text{data\_fields} \rangle \}$

$\langle \text{data\_fields} \rangle ::= \langle \text{data\_name\_field} \rangle , \langle \text{permission\_field} \rangle$

$\langle \text{data\_name\_field} \rangle ::= \text{name} : \langle \text{string} \rangle$

$\langle \text{permission\_field} \rangle ::= \text{permission} : \{ \langle \text{permission\_fields} \rangle \}$

$\langle \text{permission\_fields} \rangle ::= \langle \text{permission\_field\_item} \rangle (, \langle \text{permission\_field\_item} \rangle)^*$

$\langle \text{permission\_field\_item} \rangle ::= \text{C} : \langle \text{boolean} \rangle$   
 $\quad \quad \quad | \text{R} : \langle \text{boolean} \rangle$   
 $\quad \quad \quad | \text{U} : \langle \text{boolean} \rangle$   
 $\quad \quad \quad | \text{D} : \langle \text{boolean} \rangle$   
 $\quad \quad \quad | \text{L} : \langle \text{boolean} \rangle$

$\langle \text{process\_list} \rangle ::= \langle \text{process} \rangle (, \langle \text{process} \rangle)^*$

$\langle unit\_list \rangle \quad ::= \langle unit \rangle (, \langle unit \rangle)^*$   
 $\langle siteview\_list \rangle \quad ::= \langle siteview \rangle (, \langle siteview \rangle)^*$   
 $\langle process \rangle \quad ::= \langle string \rangle$   
 $\langle unit \rangle \quad ::= \langle string \rangle$   
 $\langle siteview \rangle \quad ::= \langle string \rangle$

Examples :

```

1  const adminDataPerm = [
2      {
3          name: 'product',
4          permission: { C: true, R: true, U: true, D: true, L: true }
5      },
6      {
7          name: 'order',
8          permission: { C: true, R: true, U: true, D: false, L: true }
9      }
10 ]
11
12 const userModel = [
13     {
14         name: 'admin',
15         auth: true,
16         allowedData: adminDataPerm,
17         deniedData: [],
18         allowedProcesses: processes,
19         deniedProcesses: [],
20         allowedUnits: uiUnits,
21         deniedUnits: [],
22         allowedSiteviews: siteViews,
23         deniedSiteviews: []
24     },
25     {
26         name: 'guest',
27         auth: false,
28         allowedData: [
29             {
30                 name: 'product',
31                 permission: { C: false, R: true, U: false, D: false, L
32                     : true }
33             },
34             deniedData: dataPermissions,
35             allowedProcesses: [checkout],
36             deniedProcesses: processes,
37             allowedUnits: [],

```

```

38     deniedUnits: uiUnits,
39     allowedSiteviews: [],
40     deniedSiteviews: siteViews
41 }
42 ]

```

## Permissions

There could be other requirements such as dynamic generation and assignment of roles to users. That is beyond the scope of this research.

### 3.2.12 State Model

The state model in this meta modeling language aims to abstract and encapsulate the state management concerns of a web application. This model can be applied to the entire application, individual views or pages, and specific UI units or components, thereby providing a structured approach to state management. The focus is entirely on the front end, reflecting the application's UI state and behavior without diving into backend logic or data persistence mechanisms.

#### Definitions

**name:** The name of the state model, in the example case below, "cart".

**initialState:** An array of objects that describe the initial state variables, the action types that affect them, and the handlers that modify the state.

**reducers:** An array of objects that define the action types and their corresponding creators for the state model.

**actions:** An array of objects that define the names and functions for creating actions to interact with the state.

**grammar :**

```

⟨statemodel⟩      ::= { ⟨statemodel_fields⟩ }

⟨statemodel_fields⟩ ::= ⟨field⟩ (, ⟨field⟩)*

⟨field⟩           ::= ⟨name_field⟩
                    |  ⟨initialState_field⟩
                    |  ⟨reducers_field⟩
                    |  ⟨actions_field⟩

⟨name_field⟩      ::= name: ⟨string⟩

⟨initialState_field⟩ ::= initialState: [ ⟨initialState_list⟩ ]

⟨reducers_field⟩  ::= reducers: [ ⟨string_list⟩ ]

⟨actions_field⟩   ::= actions: [ ⟨string_list⟩ ]

⟨initialState_list⟩ ::= ⟨initialState_item⟩ (, ⟨initialState_item⟩)*

⟨initialState_item⟩ ::= { ⟨initialState_fields⟩ }

⟨initialState_fields⟩ ::= ⟨initialState_name_field⟩ , ⟨initialState_actionType_field⟩

```

$\langle initialState\_name\_field \rangle ::= name: \langle string \rangle$

$\langle initialState\_actionType\_field \rangle ::= actionType: \langle string \rangle$

example :

```
1 const shoppingCartStateModel: StateModel = {
2   name: "cart",
3   initialState: [
4     {
5       name: "items",
6       actionType: "ADD_ITEM"
7     },
8     {
9       name: "items",
10      actionType: "REMOVE_ITEM"
11    },
12    {
13      name: "items",
14      actionType: "UPDATE_QUANTITY"
15    },
16    {
17      name: "total",
18      actionType: "ADD_ITEM"
19    },
20    {
21      name: "total",
22      actionType: "REMOVE_ITEM"
23    },
24    {
25      name: "total",
26      actionType: "UPDATE_QUANTITY"
27    },
28  ],
29  reducers: ["ADD_ITEM", "REMOVE_ITEM", "UPDATE_QUANTITY"],
30  actions: ["ADD_ITEM", "REMOVE_ITEM", "UPDATE_QUANTITY"]
31 }
```

Note for extension : In LMTH 1.2, the concept of state binding is to be introduced to facilitate the mapping of code generation, ensuring that states are accurately bound to the relevant sub-models. This approach is crucial for maintaining the integrity of the application structure. However, it is important to note that this binding process is not applicable to the application model, as there exists only one global state.

Application Level: At the application level, there is a single global state that governs the overall behavior and configuration of the application. This state is not bound to any specific sub-model but rather operates across the entire application.

Page Level: At the page level, individual pages may have their own states, which are distinct from the global state. These states are specific to each page and are used to manage the behavior and presentation of that particular page.



Unit Level: At the unit level, states such as *Currency* may be defined, which can be reused across multiple units. Each unit, however, maintains its own local state, allowing for customized behavior within the context of that unit. This local state may interact with the broader application state but remains independently managed.

Implementation: In implementation, these states are simply component states, where each component creates and manages its own state. The state can be altered as a result of binding to a data unit. When a state change occurs, it may be communicated with the data units or processes, leading to different UI outcomes. For instance, certain components may be shown or hidden, or styled differently, based on the state changes. This dynamic interaction between states and data units is a common practice, enabling a more responsive and adaptable user interface.

### 3.3 The LMTH Ecosystem

Figure 6 illustrates the various components of the LMTH language and its encompassing ecosystem, which facilitate future extensions. A key insight from the reflection on MDWE by Rossi, Urbietta et al. (2016), as discussed in the literature review, is that building a community around MDWE tools and technologies is essential for their adoption and evolution. The MDWE tools and technologies that were not prominently successful in being adopted by the industry have lacked this crucial aspect of community building.

### 3.4 Implementation

#### 3.4.1 The Transpiler

Creating a code generator that translates a custom language into TypeScript code using LLVM involves several steps.

Based on the literature and standard practices the following typical compiler development steps are followed.

Lexical Analysis: Write a lexer to convert the input source model into tokens. Each token represents a basic element in the language definition.

Syntax Analysis: Write a parser to analyze the sequence of tokens and construct an Abstract Syntax Tree (AST). The AST represents the hierarchical structure of the source code.

Semantic Analysis: Traverse the AST to perform semantic checks (like type checking) and gather additional information required for code generation. This can be achieved through semantic analysis libraries. However, omitted in this work.

Intermediate Representation (IR): Convert the AST into an intermediate representation that is easier to manipulate for code generation. LLVM IR is used for this purpose. This is the critical step leading up to code generation.

Code Generation: In the process of code generation, the intermediate representation is traversed to produce TypeScript code according to predefined custom rules. These rules facilitate the translation of constructs from the custom language into corresponding TypeScript constructs. The implementation relies on templates, which are discussed in greater detail in the subsequent subsection.

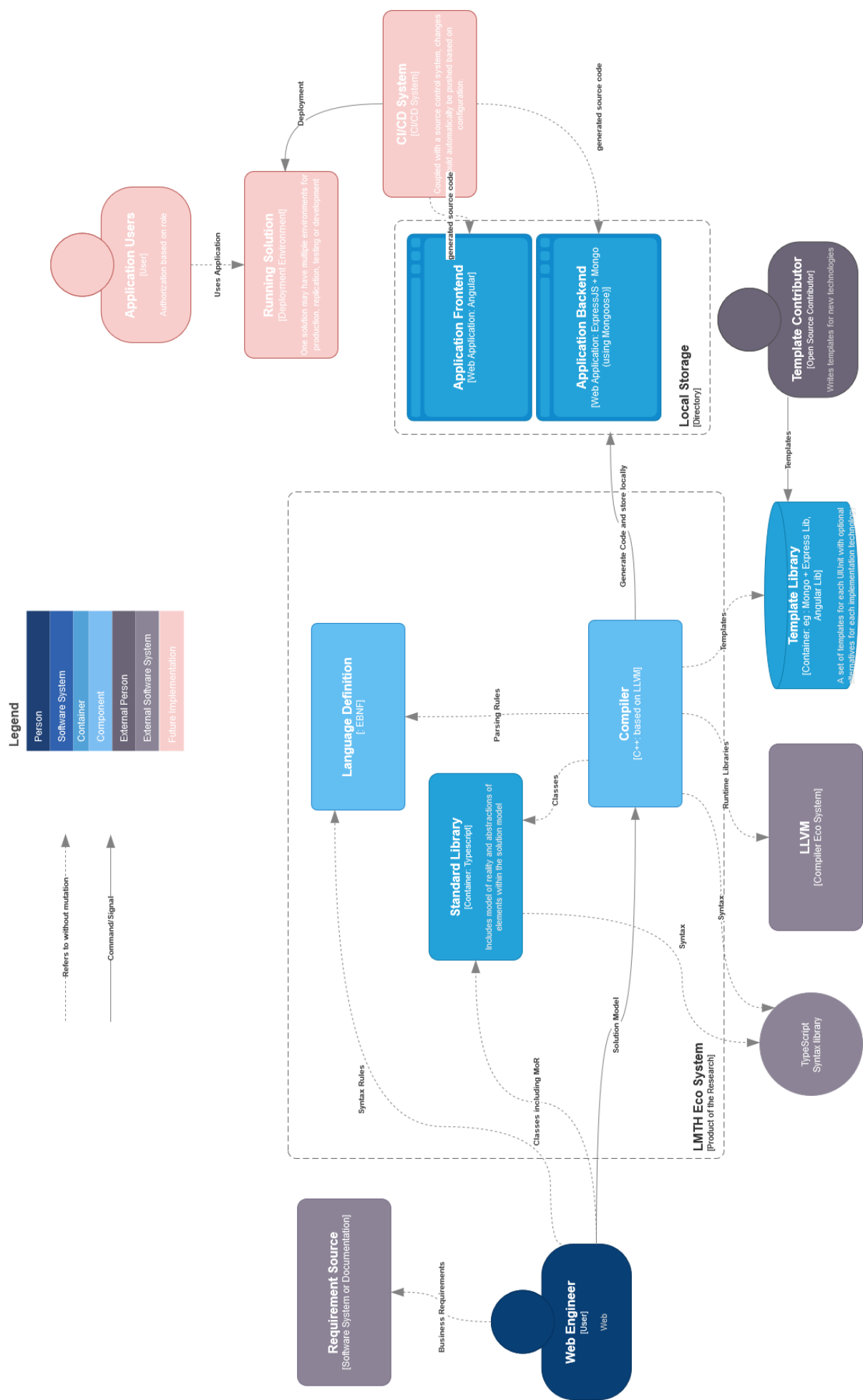


Figure 6: The LMTH Eco System

Although this phase is the most time-consuming, the availability of several open-source TypeScript libraries, with only minor modifications required, made the task more manageable and allowed for simplification of the overall process. This also reveals that the choice of TypeScript as a foundational grammar was a prudent decision.

Using LLVM for Optimizations (Optional): Although LLVM is primarily used for compiling to machine code, you can leverage LLVM's optimization passes on your intermediate representation before generating TypeScript code.

### 3.4.2 Code Generation Templates and Mapping

In this research work the generated code is based on the following technologies, frameworks and libraries:

- Angular 18
- NGPrime Component Library
- Next Js
- Auth0
- Redux
- MongoDB
- Mongoose
- Expressjs

The selection of these particular technologies was not prescriptive but rather tailored to the scope of this research demonstration, primarily influenced by the author's familiarity and prior exposure. This choice was made to effectively illustrate the concepts under consideration, though other technologies could have been equally valid depending on the context and objectives. The following table lists the specific mappings from the model to the resultant code using these technologies.

Based on the aforementioned mapping, several components generated corresponding to various UI units, are illustrated below. It is important to note that these components are also compliant with RDFa standards and WCAG guidelines, which is a critical aspect of this research. Further details on this compliance are provided in the evaluation chapter. These units are based on the example code presented above under each unit the language definition section above.

The figure 7 show a raw generated component based on IDataSingle type unit. In 9 an interesting observation can be made. It is generated based on the same templates however maintains the RDFa compliance based on the different RDF attributes for 'weight' and 'volume' for the two separate products. 9 is generated with a custom template where instead of displaying 'vegan : true', a leaf icon is shown and the price is associated with a currency pipe in angular. 10 shows the rendering of IDataulti based on the same data from a mongo database.

11 shows a process unit that is generated based on default templates. However, a limitation here is that it is generated separately although associated with the

Table 1: Model to Code mapping with selected tech stack

	<b>Frontend</b>	<b>Backend</b>	<b>Database</b>
<b>Data Model</b>	Classes in Angular model	ORM mapping with Mongoose and Express endpoints. NestJS and GraphQL would particularly suit this for future implementation.	Achieved with Mongoose snippets within Express. In the case of a relational database, this would require its own templates.
<b>Process Model</b>	Angular Service calling the backend	Express Endpoint, Controller with method stub	N/A
<b>Process Model (Frontend)</b>	Angular Service with method stub	N/A	N/A
<b>UI Model - IDataSingle</b>	Call the relevant endpoint via service	N/A	N/A
<b>UI Model - IDataMulti</b>	Call the relevant endpoint via service	N/A	N/A
<b>UI Model - IDataIndex</b>	Link to alternative link, parameter capturing on second unit	N/A	N/A
<b>UI Model - : Link - Contextual</b>	Link to alternative link, parameter capturing on second unit	N/A	N/A
<b>UI Model - : Link - Non Contextual</b>	Anchors	N/A	N/A

## Product



Name: Dummy Brand Thermal Keratin Smooth weightless Fortifying Heat Protection Spray

Description: The heat is on! Curling wands, flat irons, blow dryers, mermaid wavers-heat styling causes damage, breakage, split ends and dry hair. Stand up to heat styling with our Keratin Smooth Heat Protection Spray to prep and protect your strands up to 450°F. Our heat protecting formula is designed to enhance hair's flexibility and help it hold its shape, all while fighting heat damage. This lightweight spray also gives your hair softness and shine, so you can style away.

Volume: 500ml

Vegan: true

7700

Figure 7: Rendered IDataSingle Unit

## Dummy Brand Thermal Keratin Smooth weightless Fortifying Heat Protection Spray



**LKR 7,700.00**




The heat is on! Curling wands, flat irons, blow dryers, mermaid wavers-heat styling causes damage, breakage, split ends and dry hair. Stand up to heat styling with our Keratin Smooth Heat Protection Spray to prep and protect your strands up to 450°F. Our heat protecting formula is designed to enhance hair's flexibility and help it hold its shape, all while fighting heat damage. This lightweight spray also gives your hair softness and shine, so you can style away.

Volume: 500ml

Figure 8: Rendered IDataSingle Unit 2

**Dummy Brand2 Refreshingly Clear Daily Exfoliator**



*Your*  
**Text Here**

**LKR 5,900.00**

Developed with dermatologists, Dummy Brand2 Refreshingly Clear Daily Exfoliator with pink grapefruit and vitamin C is an uplifting daily facial scrub. The oil-free formula with natural exfoliators deeply cleanses to unclog pores and help prevent imperfections, without over-drying skin. The daily face scrub refreshes and purifies for a clearer, radiant complexion. With pink grapefruit & vitamin C For blemish-prone skin Developed with dermatologists Massage onto wet face once a day and rinse thoroughly.

Weight: 100g

Figure 9: Rendered IDataSingle Unit - Alternative Template






Name ↑↓	Image	Price ↑↓
<input type="text" value="Search"/>		
Dummy Brand Silky Smooth Argan Oil Serum		4800
Dummy Brand Ultra Repair Hair Mask with Biotin		9500
Dummy Brand Thermal Keratin Smooth weightless Fortifying Heat Protection Spray		7700
Dummy Brand Volumizing Root Lift Spray		6200
Dummy Brand Intensive Scalp Revitalizer		11500
<div>             &lt;&lt; &lt; 1 &gt; &gt;&gt;             <input type="text" value="30"/> </div>		

Figure 10: Rendered IDataSingle Unit

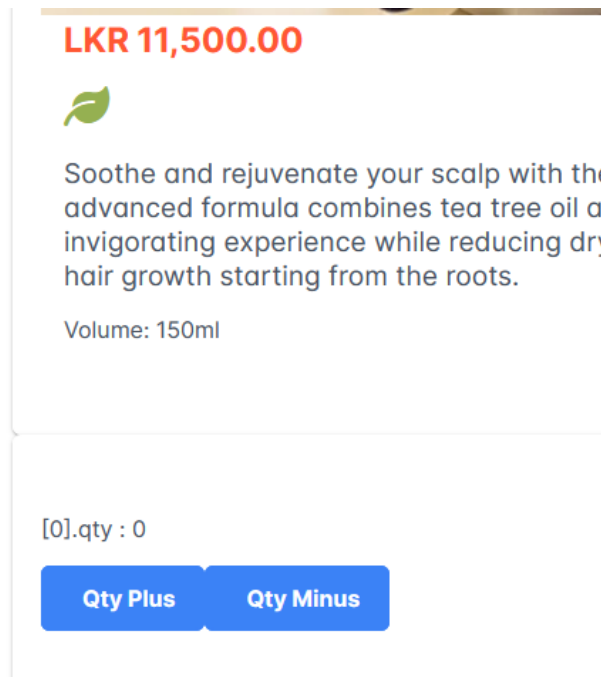


Figure 11: Rendered Process Unit

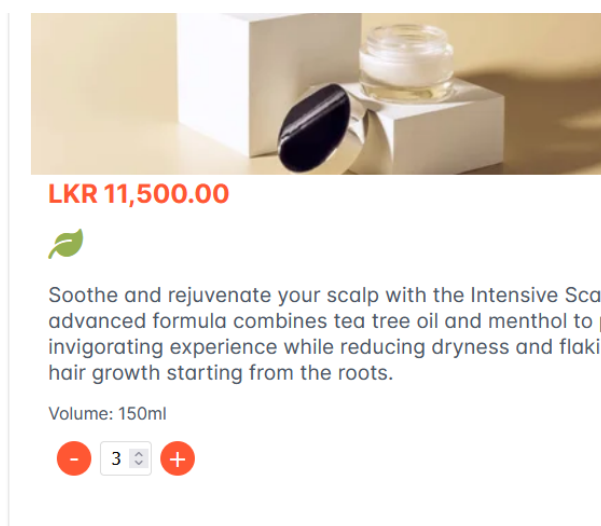


Figure 12: Process Unit after further styling

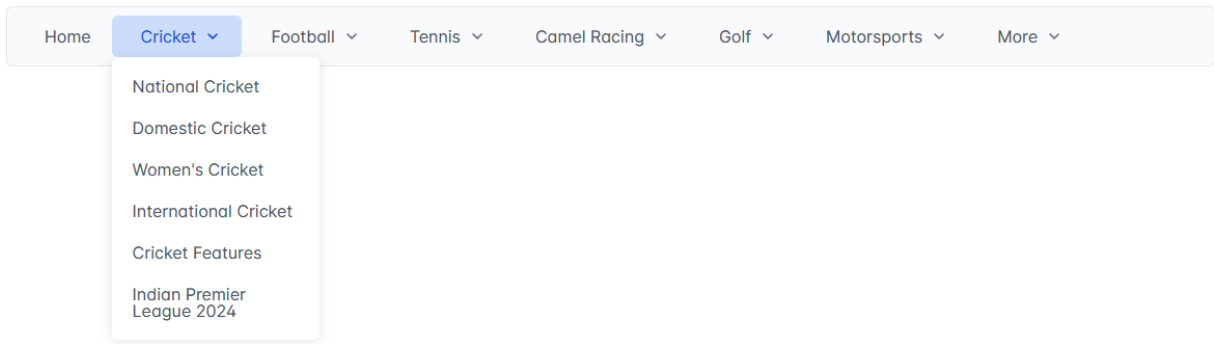


Figure 13: Default Menu Rendering

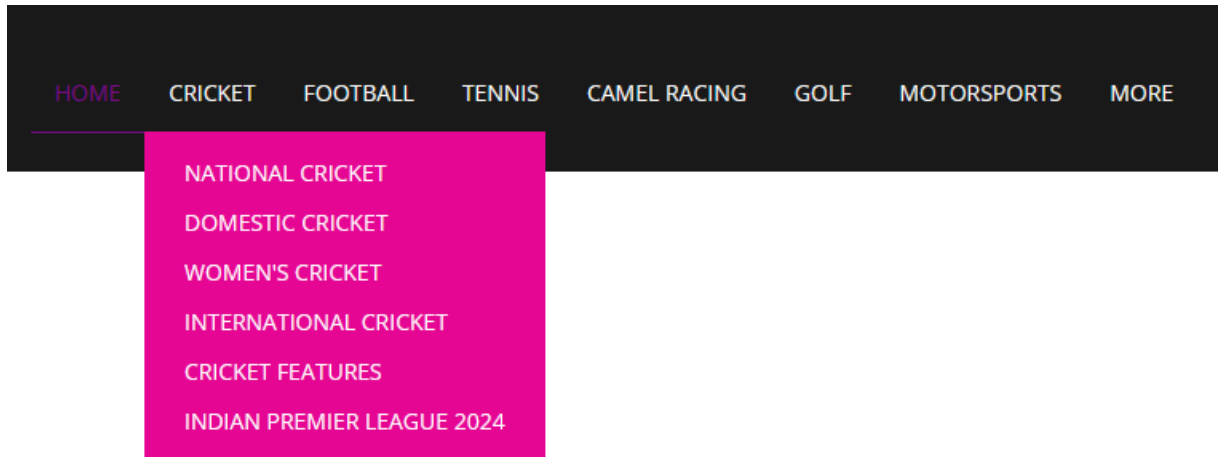


Figure 14: Menu after further styling

IDataSingle Unit above. With additional styling applied post code generation this could be overcome as shown in figure 12.

13 shows a menu generated based on a MenuUnit, and 14 shows the same menu with post-code generation styling. 15 displays a vertical menu generated using an alternative code generation template for menus within the same solution application. This demonstrates a key strength of this proposal and of model-driven engineering in general, as it allows flexibility beyond a set of preset designs.

### 3.4.3 Sample Models

Three sample models have been created based on the real needs of projects for evaluating the capabilities and performance of the novel language and the accompanying transpiler. These models can be downloaded via the links mentioned in Appendix A. The models cover all LMTH models and unit types. Information pertaining to client-specific details has been omitted from these samples. An extract of the models can be seen below.

## 3.5 Development Environment

### 3.5.1 Tools and Extensions Used

For the development of the compiler, along with the associated templates, standard library, and sample models, VS Code was utilized.



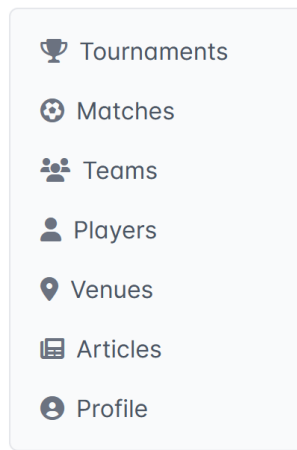


Figure 15: Menu Rendering - Alternative Template

Developers who intend to create custom solution applications using the new meta models, which are TypeScript compatible, can employ the same tools. Additionally, developers who wish to author new templates or extend the standard library can utilize the same resources. The following extensions were applied in this development process and are recommended for use:

- TypeScript Language Basics (Built-in extension): This extension is included with any new version of VS Code by default and provides basic TypeScript language support.
- Node.js: Node.js is necessary to run JavaScript/TypeScript code outside of the browser. It is essential for tasks such as running scripts, building projects, and using package managers primarily npm.
- TypeScript: TypeScript should be installed globally project. Although local installation for each project is possible, it's convenient in testing out generated solutions if it is installed globally. It is used to write and compile TypeScript code.
- ESLint (Optional but Recommended): ESLint helps in identifying and fixing problems in your TypeScript code. The ESLint extension for VS Code can be installed and configured for TypeScript.

In addition to the previously mentioned tools the following tools and extensions were employed, for compiler development. These extensions would be beneficial for developers looking to further develop the compiler, enhance it or even create their own LMTH compiler:

- CMake Tools : CMake is a cross-platform build system that allows developers to define the build process in a compiler-independent manner. It is the main primary system used by LLVM and therefore essential for managing the build process of the compiler project.
- Ninja : Ninja is another build system which is lightweight and prioritizes speed. It is also a dependency of LLVM and is required for managing the build process of the compiler.

- C++ extension for VS code by Microsoft: This extension provides essential support for C++ development within Visual Studio Code. It includes features such as IntelliSense (code completion), debugging, and code navigation, which are critical for writing and maintaining the C++ code that forms the backbone of the compiler.
- LLVM extension for VS code: This extension enhances the development experience by providing syntax highlighting and other features specific to LLVM. It aids developers in writing and understanding LLVM-specific code, ensuring that the code adheres to LLVM conventions and standards, which supports the seamless integration and functionality of the compiler with the LLVM infrastructure.

### 3.5.2 System Specification

It is recommended to have high RAM, 32GB or above. While this is not a system requirement of any tools used, it is practically useful because of the following concerns:

- Code Generation: Generating code, especially for large and complex projects, can be memory-intensive.
- Loading Templates: Working with extensive libraries and templates can consume significant memory.
- Recursive Processes: Recursive algorithms and processes used in compiler development can quickly consume available memory, potentially leading to slower performance or crashes if sufficient RAM is not available.

The specific environment on which testing was done and specific challenges faced in this regard can be found under the evaluation chapter.

## 4 Evaluation and Results

### 4.1 Objectives

This chapter focuses on presenting the comprehensive assessment strategy, the results and the analysis of LMTH and the accompanying compiler. This evaluation ensures that the language and compiler functioned as intended, offered usability benefits, and performed efficiently and scalably and presents to what extent those goals have been achieved along with the shortcomings encountered. Additionally, the evaluation ascertains the language's compliance with relevant selected standards and its comparative standing against existing alternatives for modeling web applications. The specific objectives of this evaluation are as follows:

**Functionality Assessment** : To verify that the new meta modeling language and its compiler meet all specified functional requirements and perform the tasks it was designed to accomplish.

**Usability Evaluation(DX)** : To assess the ease of use and learning curve associated with the new language. This includes evaluating the intuitiveness of the

language's syntax and the efficiency of the compiler's error messaging and debugging tools as well as the completeness and distribution of documentation.

**Scalability Analysis** : To determine the scalability of the language and compiler, ensuring they can handle large meta models and efficiently without significant degradation in performance.

**Performance Testing** : To measure the performance of the new language and compiler, including compilation speed, overall execution time and resource utilization.

**Standard Compliance** : To ensure that the new meta modeling language adheres to relevant industry standards and protocols, which is a key aspect of its novelty and potential for widespread adoption. Specific standards listed in subsequent sections.

**Comparative Analysis** : To compare the new meta modeling language against existing alternatives in terms of language quality. This will help identify the unique strengths and potential areas for improvement of the new language.

## 4.2 Scope

There are three versions of the language defined in this research. The evaluation will focus on version 1.0 for functionality, usability, and standard compliance, as it is the only version that compiles at the moment. Scalability and comparative analysis of language quality will involve both version 1.0 and version 1.1 as these aspects are based on qualitative analysis of the language syntax.

Three real projects based on real client requirements are chosen for the evaluation. By selecting real projects, any biases and forced controls in demonstrating the usage of the language are eliminated. The project that will be used for evaluation are:

1. A sports score and news web application based in UAE
2. An e-commerce store for cosmetic products

These tests would cover a wide variety of requirements, demonstrating the broad applicability of the language. Limitations discovered, if any, will be categorized as follows:

**Critical Limitations** : If certain requirements are permanently unattainable, they will be documented. This means if you use this language, you will never be able to accomplish these tasks.

**Mitigable Limitations** : Any requirements not directly met, which require manual effort, will be documented. This includes limitations in the language's expressivity, templates, or other aspects.

## 4.3 Specific Evaluation Criteria

### 4.3.1 Standard Compliance

Ensuring Web 3.0 Semantic Web standard compliance is essential as it is a key novel contribution of the new language.

- **RDFa Standard Compliance** : ensuring the language adheres to Web 3.0 Semantic web standards for RDF.
- **Schema.org Schema Compliance**: Ensuring the RDFa schema is compliant with Schema.org schemas, facilitating structured data in a cohesive manner across web applications.

#### 4.3.2 Optional Compliance:

- **WCAG 2.0 Level A Compliance**: The Web Content Accessibility Guidelines (WCAG) 2.0 are a set of recommendations for making web content more accessible to people with disabilities. Level A is the minimum level of compliance, addressing the most basic web accessibility features. Example case: Ensuring that all images of beauty products have descriptive alt text so that visually impaired users can understand the content using screen readers.
- **OWASP Application Security Verification Standard (ASVS)**: The OWASP Application Security Verification Standard (ASVS) is a framework for testing the security of web applications. It provides a basis for assessing application security controls and identifying areas for improvement.

Example case: Running security tests to ensure that the store's checkout process is secure, preventing vulnerabilities such as SQL injection or cross-site scripting (XSS) attacks that could compromise user data.

#### 4.3.3 Performance

While performance evaluation is not the primary focus, as the main goal is to establish standards and demonstrate functionality, documenting performance metrics is valuable for future improvements and optimization efforts. The time taken for compiling each of the three applications will be the measured metric. Additionally, the details of the environment in which the compilation was run, including hardware specifications, operating system, and compiler version, will be recorded and reported.

#### 4.3.4 Comparative analysis

This would be a qualitative comparison based on definitions from literature and selected instances from the sample applications being compared with their respective representation in other meta modeling languages.

This is not an exhaustive analysis of each rule but rather an attempt to place the novel language in a comparative context within the body of knowledge. The following criteria presented by Sebesta (2018) are used as a guideline in the comparison.

This comparison will include equivalent solutions such as WebML and WebDSL as well as narrower solutions such as GRPC, Ballerina, BPMN and IFML.

### 4.4 Evaluation Tools and Resources

All evaluation tests were conducted on a PC with the following specifications, which is a fairly standard and readily available setup:

Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Table 2: Language Evaluation Criteria

Processor: 11th Gen Intel Core i7-11700 @ 2.50GHz

RAM: 32.0 GB (31.7 GB usable)

System Type: 64-bit operating system, x64-based processor

Operating System: Windows 11

Browsers: Google Chrome + Wave Plugin and OpenLink Structured Data Sniffer Plugin, Mozilla Firefox, Microsoft Edge

**Custom Parser with LLVM:** A custom parser created with LLVM is used for parsing the three meta models for each scenario. This serves as a part of the functionality testing.

**Custom Code Generator with LLVM:** A custom code generator created with LLVM is used for generating the final solution source code based on the intermediate code from the parser.

**Browser Based Testing:** The resulting source code, with necessary dependencies added, is placed on a local node server to ensure it works in the browser. This setup allows verification of whether each business requirement is met. Ideally, end-to-end tests should be written; however, due to time restrictions and the fact the application does not evolve within duration of the study, requirements will be visually verified by running the application.

#### 4.4.1 Validation Tools

In order to verify the standard compliance of the generated code, the following validators were identified as suitable.

1. OpenLink Structured Data Sniffer by OpenLink Software (2024): This tool is particularly useful for identifying rich data marked with RDFa, making it ideal for this evaluation. Its availability as a browser plugin is especially advantageous as it allows for the testing of local applications.
2. Lighthouse: Utilized to evaluate various performance metrics and ensure overall quality of the web application including WCAG 2.0 compliance.
3. WAVE: Used for checking WCAG 2.0 compliance, ensuring accessibility standards are met.

Additionally, the following tools are also relevant for testing the chosen compliance requirements but have certain limitations.

1. Rich Results Test and Schema Markup Validator by Google for Developers (2024) are particularly useful in testing RDF Compliance. This tool allows copy pasting as well as testing compliance of hosted web applications. Although these tools cannot be applied within this work, it could be useful in testing further work.
2. W3C RDF Document Validator : Used to check Web 3.0 Semantic web standard compliance. However, it has a limitation in that it only allows copy-pasting code, making it impractical for validating Angular frontend code.
3. Schema.org Free Validator: However, it also has a limitation in that it only allows copy-pasting code, making it impractical for validating Angular frontend code.
4. OWASP ZAP: Employed for compliance with the OWASP Application Security Verification Standard (ASVS) ensuring the security of the application. OWASP compliance which was selected as a potential optional criterion in the scope was not covered within the research.

## **4.5 Evaluation Methodology**

**Functionality Assessment :** This will involve testing a series of predefined use cases with models for each of the selected sample scenarios that the language is expected to handle. The aim is to ensure that the language correctly models and processes information according to its design specifications.

**Usability Evaluation from developer perspective :** Metrics such as time taken to complete specific tasks and the number of errors encountered will be recorded and analyzed. If time permits conduct user studies and surveys with developers to gather qualitative data on their experiences using the language and compiler.

**Performance Testing :** Perform benchmarking tests using a variety tasks of increasing complexity. Compare the results with established performance benchmarks for similar tasks performed using other meta modeling languages.

**Scalability Analysis :** Test the language with progressively larger and more complex models to observe how the compiler performs under varying loads. Considering that templates are injected recursively the performance level should be adequate to generate the sample applications within a reasonable time. Record and analyze performance metrics to identify any potential bottlenecks or limitations. Extensibility aspects of the language should be evaluated as well.

**Standard Compliance :** Conduct a thorough review of the language's design and implementation against established standards in the field of meta modeling. This will include both automated and manual verification processes to ensure full compliance using the tools identified above.

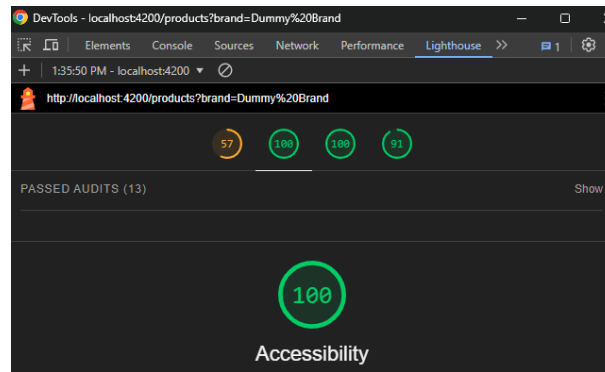


Figure 16: Sample Lighthouse Report on Scenario 2 Products Listing Page

**Comparative Analysis :** Identify key alternative meta modeling languages and perform a comparative study. This will involve implementing the same set of tasks in both the new language and the alternatives, followed by a detailed analysis of the results across various metrics.

## 4.6 Evaluation Results

**Implementation of Requirements** The following scenarios were tested.

Scenario 1: Sports News and Scores Application

Scenario 2: Cosmetics Store with Next.JS (pivoted and switched to angular)

Scenario 2.1: Cosmetics Store

**Requirement Coverage** A significant portion of the customer requirements were covered in the generated code while maintaining compliance. Some post-code generation modifications were required; however, these were mostly related to styling. It could be concluded that this is a good coverage.

**Language Expressivity and Developer Support** The language provides comprehensive solutions for complex web applications. Where limitations were encountered, new rules have been introduced in LMTH 1.1. Overall, it could be considered a highly expressive language.

**Performance Benchmarks and Future Improvements** The average compile time for Scenario 1, based on an average of five compilations, was **43 minutes and 29 seconds**, while for Scenario 2, the average was **27 minutes and 9 seconds**. This compile time could be considered acceptable for generating an entire application. However, even after a minor change, the compile time remains lengthy, indicating that optimization is necessary—a consideration that was omitted in this work.

Given the standard setup with 32GB RAM documented in the Evaluation Resources sub-section, the compiler encountered an "LLVM Error: Out of Memory" error several times but was able to execute on subsequent runs. This reveals the need for compiler optimization if developers are expected to use this regularly, as repeated errors may lead to frustration.

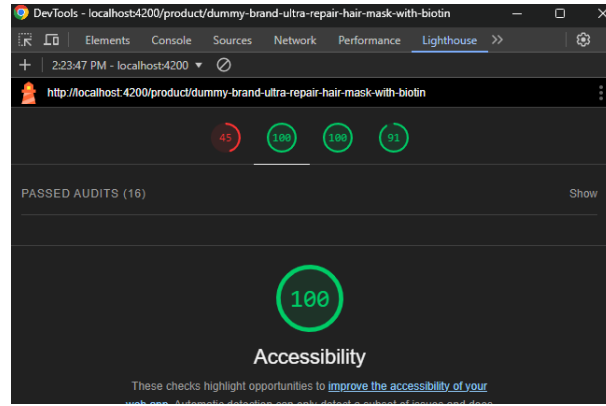


Figure 17: Lighthouse Report Scenario 2 Individual Product Page

**Standard Compliance** Figure 16 and 17 shows sample lighthouse reports and the scenario-wise tables summarize overall standard compliance of the generated code.

**Comparative Analysis** LMTH has a greater number of primitives compared to its visual alternative, with these primitives being more specialized yet orthogonal at a similar level. However, its power lies in the simplicity of overall expression and developer familiarity.

**Empirical Analysis** Empirical assessment of this novel language’s performance across diverse user profiles and environments, including factors such as learnability, was beyond the scope of this research due to time constraints. Currently, professionals and industry experts are evaluating the language, and their feedback will be collected using the survey form provided in Appendix C. This feedback will be analyzed and publicized as a separate study to inform further development and refinement of the language.

## 4.7 Overall Analysis

Based on the aforementioned methodology, comprehensive results would be obtained that provide a clear assessment of the new meta modeling language. The analysis would indicate the overall success or failure of the research, whether the language has met its design goals and standard compliance to the specified levels based on the selected tools and place the novel language in the context of model driven engineering paradigm. The limitations that arose do not indicate any fundamental flaws with the proposed language. They are primarily implementation-specific issues, such as those related to the compiler and templates.

# 5 Conclusion and Future Work

## 5.1 Summary of Findings

The literature and practical applications reveal a gap in model-driven web engineering in terms of adaptation to the Web 3.0 paradigm, enhanced developer experi-



View	Page	UI Units	Notes	Rich Data Compliance	Lighthouse			WAVE		
					Accessibility	Performance	Best Practices	SEO	Errors	Alerts
Guest	View Layout (Master Page)	4		OnlineStore	100	53	95	100	0	12
	Home	16		Articles	100	55	88	100	0	15
	Tournament List	3		SportsTeam	100	78	100	100	0	2
	Tournament	4		SportsTeam	100	64	100	100	0	6
	Match List	3		SportsTeam	100	68	100	100	0	2
	Match	4		SportsTeam	100	74	100	100	0	6
	Team List	3		SportsTeam	100	66	100	100	0	7
	Team	2		SportsTeam	100	56	100	100	0	3
	Player List	3		SportsTeam	100	76	100	100	0	2
	Player	2		SportsTeam	100	47	78	100	0	1
	Venue List	3		Article	100	48	100	100	0	0
	Venue	1		Article	100	89	100	100	0	2
	Articles	5		Article	99	89	100	100	0	2
	Article	5		Article	99	47	100	100	0	4
	View Layout (Master Page)				100	68	100		0	5
	Admin - Tournament				100	68	100		0	6
Admin Panel	Admin - Match				100	69	100		0	7
	Admin - Team				100	69	100		0	1
	Admin - Player				100	69	100		0	1
	Admin - Venue				100	70	100		0	1
	Admin - Article				99	90	78		0	4
	Admin Authenticate			N/A	79	82	79		1	19
	Profile	Auth0 based profile page generated			100					

Figure 18: Compliance Testing - Sports News and Scores Application

View	Page	UI Units	Notes	Rich Data Comp	Lighthouse				WAVE			
					Accessibility		Performance		Best Practices	SEO	Errors	Alerts
	View Layout (Master Page)	5			100	92	100	100	95	0	17	
Guest	Product List	3	parameter based filtering also works	Product	90	57	100	100	91	0	2	
	Product	3		Product	100	45	100	100	91	0	2	
	Cart	2		Product	100	46				0	4	
	Checkout			-	100	77				0	5	
	Profile		Auth0 based profile page generated		100	55				0	2	
	Authenticate			N/A	79	82	79	92	2	9		
	Contact	1		Organization						0	1	
Admin Panel												excluding masterpage alerts
	Admin - Product	5	image upload. didnt' work off the box. had to add	Product							0	3
	Admin - Orders										0	1
	Admin Authenticate			N/A	79	82	79			1	9	
Admin Panel												
	Profile		Auth0 based profile page generated							0	1	
responsive. Lighthouse taken into account. No additional tests done with this.												
Authentication done. Authorization by unit level not done. The templates were not updated. This is a security flaw. However can be mitigated by updating the templates accordingly												

Figure 19: Compliance Testing - E commerce application

ence, and addressing the complexities of modern web applications. While numerous standards and tools for model-driven web engineering are well-established and mature, this research leverages these foundations and integrates core semantic web concepts to produce a novel and promising approach for representing complex web applications as meta models.

This research demonstrates that, when combined with a suitable transpiler, these meta models can generate web application code that is agnostic of specific backend and frontend frameworks and libraries, addressing the ever-evolving nature of web technologies. The default compiler, although still improvable, successfully generates code within acceptable parameters, serving as a proof of concept for the approach.

Primarily, due to the approach in representing the web application solutions, maintaining cohesive identities with the help of ontology vocabularies specifically Schema.org allows for the seamless generation of rich data, which is also compliant with WCAG accessibility guidelines. This establishes the potential for further extension and enhancement of this approach, paving the way for more accessible, efficient, and future-proof web application development.

## **5.2 Implications**

The implications of this work for the field of programming languages are profound, particularly in how it can shape development practices and industry standards. By introducing a language that stays on par with existing meta-modeling languages in terms of characteristics such as simplicity, orthogonality, data types, syntax design, support for abstraction and expressivity while also comparatively improving in several of these aspects leading to better developer-friendliness, this research hopes to set a new standard for model-driven web engineering.

### **5.2.1 Impacts on Development Practices**

1. **Enhanced Developer Experience:** The focus on improving developer experience means that the language and its associated tools are easier to learn and use, reducing the learning curve for new developers and increasing productivity for experienced ones. This would lead to faster development cycles and more consistent code quality in the long run.
2. **Efficiency in Maintenance:** Organizations, especially those with stringent compliance requirements, stand to benefit significantly. The ability to generate code that adheres to standards such as the WCAG accessibility standards means that less manual intervention is required post-code generation. This efficiency becomes crucial when standards change or new technologies emerge. Instead of rewriting large portions of code, developers only need to update the meta model, which simplifies the maintenance process, minimizes technical debt as new changes are introduced and reduces the potential for errors.
3. **Standardization and Interoperability:** As the language matures and gains traction, it could drive the adoption of new industry standards, promoting greater interoperability between different tools and frameworks. This can lead to

a more unified approach to web application development, where developers and organizations can rely on consistent methodologies and best practices.

### **5.2.2 Impacts on Industry Standards**

1. **Adoption of Semantic Web Concepts:** By integrating semantic web concepts at its core, this work encourages the adoption of these principles across the industry. This could lead to more intelligent and interconnected web applications, enhancing the overall user experience and enabling more advanced functionalities such as automated reasoning and data integration.
2. **Compliance and Accessibility:** The built-in support for WCAG accessibility guidelines and strong compliance features means that this language can help organizations meet regulatory requirements more easily. As compliance becomes increasingly important, especially in sectors like education, finance, healthcare and government, the adoption of this language could set a new standard for accessibility and legal compliance in web development.
3. **Future-Proof Web Development:** The flexibility to generate code that is agnostic to backend and frontend frameworks means that applications developed using this language are more resilient to changes in the technological landscape. This future-proofing aspect is critical as it allows organizations to adapt quickly to new trends and technologies without the need for extensive rewrites, thereby protecting their investments in software development.

Overall, the implications of this research suggest a shift towards more efficient, compliant, and adaptable web development practices. By addressing the current gaps in model-driven web engineering and offering a more developer-friendly approach, this work has the potential to significantly influence both the practice and standards of web application development in the future.

## **5.3 Limitations**

Despite the promising advancements and potential implications of this research, several limitations have been identified that need to be addressed to enhance its applicability and robustness further.

1. **Dynamic Menu Generation:** While not required in the sample scenarios tested in this work, dynamic menu generation is a requirement for many web applications. The defined meta models do not adequately capture this feature, limiting its use in applications where menus need to adapt based on user interactions or context.
2. **Dynamic Roles:** LMTH currently works with a predefined set of user roles, which restricts its flexibility. This limitation arises because of the tightly coupled implementation that does not accommodate dynamically created roles, making it less adaptable in environments with frequently changing user role requirements.

3. **Technology Coupling:** Certain parts of the implementation are too closely coupled with specific technologies. For instance, MongoDB queries are directly written within the models, filter parameters are based on Angular style and state management functions are closely aligned with Redux. This tight coupling necessitates updates to the language rules or existing implementations to allow for greater abstraction and adaptability to different technologies. This is a rather relatively simpler limitation to overcome with the expansion of the syntax to accommodate one more level of abstraction where such coupling is identified.
4. **Post Code Generation Work:** Although the goal is to minimize work after code generation, achieving fully generated styles and complete functionality remains complex. The current language syntax and templates must be updated to prevent developers having to introduce breaking changes. Until this is resolved, the advantage of minimal maintenance effort cannot be fully realized, even though it is theoretically possible.
5. **Manual Dependency Management:** The generated code, with any modern web development frameworks and libraries, requires numerous dependencies. While the generated code includes a package.json file with relevant dependency list added, developers must manually install these dependencies with a package manager. Any issues arising from dependencies must be handled separately, adding to the manual workload.
6. **Version Dependence:** Although the approach is overall technology-agnostic and future-proof, library or framework version updates necessitate corresponding updates to the templates to stay current. This version dependence can introduce additional maintenance overhead as well as compatibility conflicts.
7. **Limited Interoperability:** The current system is compatible with class diagrams and ER diagram-based code generation tools for data models. However, it lacks compatibility potential standards, such as with BPMN to process model mapping, which would be beneficial for broader application scenarios.
8. **Data Integration Limitations:** The current implementation supports data integration through a database, REST API calls and session-based data handling. However, modern data integration often involves technologies such as message queues, WebRTC, WebSockets or webhooks, which are not currently supported.
9. **Event-Triggered Actions:** Modeling event-triggered actions, such as user interactions, timers, or system events, is not supported. These actions need to be manually added, limiting the automation and completeness of the generated code.
10. **Omitted Models:** To keep the research scope manageable, certain models such as personalization, internationalization, and deployment models were deliberately omitted. Including these models would enhance the applicability and utility of the language for a broader range of scenarios.

11. **Support for Architectural Design:** The proposed language focuses on developer-centric features rather than architectural design. For architectural design tasks, visual modeling languages like WebML, IFML and UML remain more suitable. These alternative methods are better suited for defining system structure and behavior, which is outside the scope of the proposed language.
12. **No Legacy Support:** LMTH is primarily useful for developing new web applications rather than for maintaining or upgrading existing ones. It lacks reverse engineering capabilities, meaning it cannot extract models from existing web applications for analysis or modification. This limitation makes it less suitable for projects that require understanding and updating legacy systems, as developers cannot use the language to generate models from existing code-bases.

## 5.4 Future Work

### Language Enhancements

- **Expressivity:** Enhance the language to address current limitations, such as the ability to handle dynamic menus and roles.
- **Interoperability:** Improve interoperability with additional standards by developing converters, particularly for BPMN (Business Process Model and Notation).
- **Design Patterns & Architectures:** Expand the language to include various design patterns and architectures, beyond the currently supported MVC (Model-View-Controller) framework.
- **Web API Support:** Extend support for Web APIs. Currently, session APIs are supported; this should be expanded.
- **Ontology Library:** Broaden the ontology library by incorporating additional ontologies.

### Compiler and Code Generation

- **Performance Optimization:** Enhance performance through optimization features available in LLVM and explore alternative approaches.
- **Framework and Library Support:** Add support for more front-end and back-end frameworks, libraries, and DBMSs, with the inclusion of more code template libraries.
- **Automated Dependency Management:** Implement automated dependency management for improved efficiency.

### Community Building

- **Networking & Forums:** Create provisions for networking channels and forums to foster community building.

- **Community Feedback:** Actively seek community feedback for continuous improvement.
- **Open Source Collaboration:** Encourage community involvement in further developing the codebases as open-source projects.

## 5.5 Final Remarks

The work presented in this thesis builds upon the long lineage of theoretical and empirical research on Model Driven Web Engineering, Meta Programming and Semantic Web Technologies. These contributions are profound in that they not only address existing gaps in language and tool development but also introduce new paradigms for state binding and abstraction in web development.

Reflecting on the overall project, it is clear that the proposed approaches and extensions have the potential to make a lasting impact on the ever-evolving field of web development. This work contributes to the ongoing development of languages and tools, aiming to empower developers and enrich the broader community.

It is our hope that these advancements will catalyze further innovation and collaboration, ultimately leading to a more robust and interconnected web ecosystem built on a 'web of data'. The contributions made here are intended not just to solve immediate challenges, but to pave the way for future developments that will continue to shape the web development landscape.

## **Appendix A : Additional Resources and Links**

This thesis is written from an academic perspective, with a focus on researchers and those who intend to study and extend the LMTH language. For a more developer-centric perspective, focusing on building application solutions, please visit:

- LMTH Developer Resources: <https://www.semwei.org/lmth>

### **Source Code**

The source code for the LLVM-based compiler for LMTH can be obtained via the following links:

- LMTH Compiler on SemWEI: <https://www.semwei.org/lmth-compiler>
- LMTH Compiler on GitHub: <https://github.com/SemWEI/lmth-compiler>

Instructions for setting up the compiler locally can be found at these repositories.

### **Sample Models**

Sample models are available at:

- LMTH Sample Models on GitHub: <https://github.com/SemWEI/models>

### **Core Library**

The core library for the 'model-of-reality' based on schema.org is available at:

- Model-of-Reality Library on GitHub: <https://github.com/SemWEI/reality>



## **Appendix B : Disclaimer**

The technologies, frameworks, and libraries mentioned in this research work, including but not limited to Angular, NGPrime Component Library, Auth0, Redux, MongoDB, Mongoose, NextJS and Expressjs are utilized solely for testing and demonstration purposes. The use of these technologies is not an endorsement or recommendation of any particular product, service, or practice.

The authors of this research work do not claim any ownership or proprietary rights over the mentioned technologies and have not extracted or copied any proprietary content from them. Any issues, problems, or defects discussed or identified in this research work are solely for illustrative purposes and do not necessarily reflect the actual performance or reliability of the mentioned technologies.

The authors disclaim any responsibility or liability for any harm, loss, or damage arising from the use, reference to, or reliance on the technologies mentioned herein. This disclaimer extends to any direct, indirect, incidental, consequential, or punitive damages arising out of or related to the use or performance of the technologies discussed in this research work.

By reading or utilizing this research work, you acknowledge and agree to the terms of this disclaimer. If you have any concerns or require further information about the technologies mentioned, it is recommended that you seek professional advice or consult the official documentation of the respective technologies.

For specific open-source licenses and related information, please refer to the source code repositories mentioned in Appendix A.

## Appendix C : Sample Survey Form

### Expert Feedback on a Novel Meta Modeling Language for the Web

Thank you for participating in this survey. Please provide feedback based on your experience with trying LMTH. Your insights will help shape the development of LMTH. The survey should take approximately 10-15 minutes to complete.

#### Part 1: Participant Profile

1. What is your area of expertise? (Select all that apply)
  - Web Development
  - Software Engineering
  - Systems Design
  - Data Modeling
  - Front-end Development
  - Back-end Development
  - Other (Please specify): \_\_\_\_\_
2. How many years of experience do you have in web or software development?
  - Less than 5 years
  - 5-10 years
  - More than 10 years
3. What is your familiarity with meta modeling/4th Generation languages?
  - Never used
  - Aware of the concept but never used
  - Have used them occasionally
  - Frequently use them in my work

#### Part 2: Feedback on Meta Modeling Aspects

4. How intuitive is the syntax of this new meta modeling language?  
(1 = Not intuitive at all, 5 = Very intuitive) \_\_\_\_\_
5. How well does the language support common web development tasks?  
(1 = Not well at all, 5 = Very well) \_\_\_\_\_
6. Do you find LMTH to be flexible enough to extend or customize for different web architectures?
  - Yes
  - No

- Unsure

Please elaborate your answer \_\_\_\_\_

7. Does the language provide sufficient abstraction while maintaining control over lower-level details?

- Yes
- No

If not, please suggest improvements:

8. How does this language compare to existing modeling languages you've used?  
(1 = *Much worse*, 5 = *Much better*)

\_\_\_\_\_

Why? \_\_\_\_\_

### Part 3: Performance and Efficiency

9. How would you rate the performance of the meta modeling language in terms of:

- Modeling speed and ease of use  
(1 = *Very slow/difficult*, 5 = *Very fast/easy*) \_\_\_\_\_
- Rendering or compiling models into "native code"  
newline (1 = *Very slow/inefficient*, 5 = *Very fast/efficient*) \_\_\_\_\_

### Part 4: Usability and Documentation

10. How clear and comprehensive is the language's documentation?

(1 = *Very unclear*, 5 = *Very clear*) \_\_\_\_\_

Suggestions for improvement \_\_\_\_\_

11. What additional features, if any, would you suggest to improve the usability of this language?

- More tutorials
- More examples
- A more detailed reference guide
- Interactive online playground
- Other (Please specify): \_\_\_\_\_

12. On a scale of 1-5, how likely are you to adopt this meta modeling language in your work?

(1 = *Not likely at all*, 5 = *Extremely likely*) \_\_\_\_\_

Why or why not? \_\_\_\_\_

## **Part 5: Open Feedback**

13. What are the primary strengths of this meta modeling language for web technologies?

---

14. What are the most significant weaknesses or challenges with this language?

---

15. Do you see any specific areas where this language could provide a unique advantage compared to existing alternatives?

---

16. Any additional comments or feedback?

---

## References

- [1] Timothy J Berners-Lee (1989). *Information management: a proposal*. March 1989 version includes the annotations from Mike Sendall, Tim Berners-Lee's supervisor. URL: <https://cds.cern.ch/record/369245>
- [2] Tim Berners-Lee et al. (May 2001). 'The Semantic Web : A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities'. In: *Scientific American*, pp. 35–43
- [3] Robert W. Sebesta (2018). *Concepts of Programming Languages*. 12th. Boston: Pearson. ISBN: 978-0134997186
- [4] Harry Halpin (May 2006). 'Identity, Reference, and Meaning on the Web'. In: *Proc. WWW 2006 Workshop on Identity, Reference, and the Web*. Edinburgh, United Kingdom. URL: <http://www.ibiblio.org/hhalpin/irw2006/hhalpin.pdf>
- [5] Tim Berners-Lee (July 2003). *Re: URIs, Addressability, and the use of HTTP GET and POST*. Online. URL: <https://lists.w3.org/Archives/Public/www-tag/2003Jul/0022.html>
- [6] John Hebel et al. (2009). *Semantic Web Programming*. Indianapolis, IN: Wiley Publishing, Inc. ISBN: 978-0-470-41801-7. URL: <https://www.wiley.com/en-us/Semantic+Web+Programming-p-9780470418017>
- [7] Stefano Ceri et al. (2000). 'Web Modeling Language (WebML): a modeling language for designing Web sites'. In: *Computer Networks* 33.1, pp. 137–157. ISSN: 1389-1286. DOI: [https://doi.org/10.1016/S1389-1286\(00\)00040-2](https://doi.org/10.1016/S1389-1286(00)00040-2). URL: <https://www.sciencedirect.com/science/article/pii/S1389128600000402>
- [8] Benjamin Heitmann (Dec. 2007). 'Transitioning Web Application Frameworks Towards the Semantic Web'. Supervised by Prof. Dr. Stefan Decker and Dr. Ir. Eyal Oren; 2nd supervisor: Prof. Dr. Rudi Studer. PhD Thesis. Galway, Ireland: National University of Ireland, Galway. URL: [https://www.researchgate.net/publication/237021317\\_Transitioning\\_web\\_application\\_frameworks\\_towards\\_the\\_Semantic\\_Web](https://www.researchgate.net/publication/237021317_Transitioning_web_application_frameworks_towards_the_Semantic_Web) (visited on 06/08/2024)
- [9] Google for Developers (2024). *Schema Markup Testing Tool*. URL: <https://developers.google.com/search/docs/appearance/structured-data>
- [10] OpenLink Software (2024). *OpenLink Structured Data Sniffer*. URL: <https://chromewebstore.google.com/detail/openlink-structured-data/egdaiaihbdoiibopledjahjaihbmjhdj>
- [11] TypeDB (2024). *Imperative Querying*. URL: <https://typedb.com/fundamentals/imperative-querying>
- [12] David Granada et al. (Jan. 2015). 'Analysing the cognitive effectiveness of the WebML visual notation'. In: *Software Systems Modeling* 16. DOI: 10.1007/s10270-014-0447-8

- [13] Martin Hepp (2008). 'GoodRelations: An Ontology for Describing Products and Services Offers on the Web'. In: *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2008)*. E-Business and Web Science Research Group, Bundeswehr University Munich, Germany. Acitrezza, Italy: Springer, pp. 329–346
- [14] Peter F. Patel-Schneider (2014). 'Analyzing Schema.org'. In: *The Semantic Web – ISWC 2014*. Ed. by Peter Mika et al. Cham: Springer International Publishing, pp. 261–276. ISBN: 978-3-319-11964-9
- [15] 'WebML Modeling in UML' (Feb. 2007). In: *Software, IET*. URL: [https://www.researchgate.net/publication/237331381\\_WebML\\_Modeling\\_in\\_UML](https://www.researchgate.net/publication/237331381_WebML_Modeling_in_UML)
- [16] 'Model-driven design and development of semantic Web service applications' (Nov. 2007). In: *ACM Transactions on Internet Technology* 8 (1), p. 3. ISSN: 1533-5399. DOI: 10.1145/1294148.1294151
- [17] Weijun Sun et al. (2009). 'A Model-Driven Reverse Engineering Approach for Semantic Web Services Composition'. In: *IEEE*, pp. 101–105. ISBN: 978-0-7695-3570-8. DOI: 10.1109/WCSE.2009.403
- [18] Madhushi Bandara and Fethi A. Rabhi (Apr. 2020). 'Semantic modeling for engineering data analytics solutions'. In: *Semantic Web* 11 (3). Ed. by Oscar Corcho, pp. 525–547. ISSN: 22104968. DOI: 10.3233/SW-190352. URL: <https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SW-190352>
- [19] GEYLANI KARDAS et al. (June 2009). 'MODEL DRIVEN DEVELOPMENT OF SEMANTIC WEB ENABLED MULTI-AGENT SYSTEMS'. in: *International Journal of Cooperative Information Systems* 18 (02), pp. 261–308. ISSN: 0218-8430. DOI: 10.1142/S0218843009002014
- [20] Manuel Álvarez Álvarez et al. (2010). 'Bridging together Semantic Web and Model-Driven Engineering'. In: pp. 601–604. DOI: 10.1007/978-3-642-14883-5\_76
- [21] Olga Kovalenko et al. (Sept. 2015). 'Modeling AutomationML: Semantic Web technologies vs. Model-Driven Engineering'. In: *IEEE*, pp. 1–4. ISBN: 978-1-4673-7929-8. DOI: 10.1109/ETFA.2015.7301643
- [22] R. Groenmo and M.C. Jaeger (2005). 'Model-driven semantic Web service composition'. In: *IEEE*, 8 pp. ISBN: 0-7695-2465-6. DOI: 10.1109/APSEC.2005.81
- [23] Claus Pahl (Aug. 2007). 'Semantic model-driven architecting of service-based software systems'. In: *Information and Software Technology* 49 (8), pp. 838–850. ISSN: 09505849. DOI: 10.1016/j.infsof.2006.09.007
- [24] Jin Song Dong (2004). 'Software modeling techniques and the semantic Web'. In: *IEEE Comput. Soc*, pp. 724–725. ISBN: 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317506

- [25] Marco Brambilla and Federico M. Facca (2009). 'Building Semantic Web Portals with a Model-Driven Design Approach'. In: IGI Global, pp. 46–106. DOI: 10.4018/978-1-60566-112-4.ch004
- [26] Enrique Chavarriaga and José A. Macías (Dec. 2009). 'A model-driven approach to building modern Semantic Web-Based User Interfaces'. In: *Advances in Engineering Software* 40 (12), pp. 1329–1334. ISSN: 09659978. DOI: 10.1016/j.advengsoft.2009.01.016
- [27] Antonio Vallecillo et al. (Feb. 2007). 'MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods'. In
- [28] Chris Lattner and Vikram S Adve (2004a). 'LLVM: a compilation framework for lifelong program analysis transformation'. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. URL: <https://api.semanticscholar.org/CorpusID:978769>
- [29] Jorge Cardoso (Feb. 2007). 'The Semantic Web Vision: Where Are We?' In: *Intelligent Systems, IEEE* 22, pp. 84–88. DOI: 10.1109/MIS.2007.4338499
- [30] Ian Horrocks (Dec. 2008). 'Ontologies and the semantic web'. In: *Communications of the ACM* 51 (12), pp. 58–67. ISSN: 0001-0782. DOI: 10.1145/1409360.1409377
- [31] Ivan Herman (2013). *Publications of the W3C Semantic Web Activity — w3.org*. [Accessed 17-01-2024]
- [32] John Domingue et al. (2011). *Handbook of Semantic Web Technologies*. Springer
- [33] Abderahman Rejeb et al. (2022). *Charting Past, Present, and Future Research in the Semantic Web and Interoperability*. DOI: 10.3390/fi14060161
- [34] Kasun Anupama Gamage (2017). 'Separation of Concerns for Web Engineering Projects'. In: *Proceedings of the Second Asia-Pacific Conference on Multidisciplinary Research*. Asia-Pacific Conference on Multidisciplinary Research. Colombo, Sri Lanka, pp. 216–221. ISBN: 978-955-4543-42-3
- [35] *Automation of Web services development using model driven techniques* (Mar. 2010). Vol. 3, pp. 190–194. DOI: 10.1109/ICCAE.2010.5452059
- [36] Antonio Cicchetti et al. (Feb. 2011). 'Managing the evolution of data-intensive Web applications by model-driven techniques'. In: *Software and System Modeling - SOSYM 12*, pp. 1–31. DOI: 10.1007/s10270-011-0193-0
- [37] Marco Brambilla and Piero Fraternali (Feb. 2014). 'Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience'. In: *Science of Computer Programming* 89, pp. 71–87. DOI: 10.1016/j.scico.2013.03.010

- [38] Andrea Schauerhuber et al. (Feb. 2006). 'Bridging existing web modeling languages to model-driven engineering: A metamodel for WebML'. in: vol. 155, p. 5. DOI: 10.1145/1149993.1149999
- [39] Alvis Fong et al. (Feb. 2012). 'Generation of Personalized Ontology Based on Consumer Emotion and Behavior Analysis'. In: *Affective Computing, IEEE Transactions on* 3, pp. 1–1. DOI: 10.1109/T-AFFC.2011.21
- [40] K.A.M. Sluijs Van Der (2012). 'Model driven design and data integration in semantic web information systems'. In: DOI: 10.6100/IR732193. URL: <https://doi.org/10.6100/IR732193>
- [41] 'OWL rules: A proposal and prototype implementation' (2005). In: *Journal of Web Semantics* 3 (1). Rules Systems, pp. 23–40. ISSN: 1570-8268. URL: <https://www.sciencedirect.com/science/article/pii/S1570826805000053>
- [42] Yue Pan et al. (2006). 'Model-Driven Ontology Engineering'. In: pp. 57–78. DOI: 10.1007/11890591\_3
- [43] Victoria Torres et al. (2006). 'Building Semantic Web Services Based on a Model Driven Web Engineering Method'. In: pp. 173–182. DOI: 10.1007/11908883\_21
- [44] Samad Paydar and Mohsen Kahani (June 2015). 'A semantic web enabled approach to reuse functional requirements models in web engineering'. In: pp. 241–288. DOI: 10.1007/s10515-014-0144-4
- [45] Santiago Meliá and Jaime Gómez (2006). 'The WebSA Approach: Applying Model Driven Engineering to Web Applications'. In: *J. Web Eng.* 5, pp. 121–149. URL: <https://api.semanticscholar.org/CorpusID:2113605>
- [46] et al. (2007). 'Semantic Web: A state of the art survey'. In: *International Review on Computers and Software*. URL: <https://api.semanticscholar.org/CorpusID:8225359>
- [47] H. Agius (Mar. 2004). 'Review: The Semantic Web'. In: *The Computer Bulletin* 46 (2), pp. 31–31. ISSN: 0010-4531. DOI: 10.1093/combul/46.2.31-a
- [48] F. van Harmelen (Mar. 2004). 'The semantic web: what, why, how, and when'. In: *IEEE Distributed Systems Online* 5 (3), pp. 1–4. ISSN: 1541-4922. DOI: 10.1109/MDSO.2004.1285880
- [49] World Wide Web Consortium (2011). *W3C Validator - Semantic Web Standards*. URL: <https://www.w3.org/2001/sw/wiki/SWValidators>
- [50] D Miles (Feb. 2017). 'ARTICLE: "Research Methods and Strategies Workshop: A Taxonomy of Research Gaps: Identifying and Defining the Seven Research Gaps"'. In: 1, p. 1
- [51] P Gottgtroy et al. (Feb. 2003). 'An ontology engineering approach for knowledge discovery from data in evolving domains'. In



- [52] Tom Preston-Werner (June 2013). *Semantic versioning 2.0.0*. URL: <https://semver.org/>
- [53] Pradeepa Somasundaram (Jan. 2024). *Top 20 software documentation tools of 2024*. URL: <https://document360.com/blog/software-documentation-tools/>
- [54] Egon Willighagen (Feb. 2014). *Accessing biological data in R with semantic web technologies*. DOI: 10.7287/peerj.preprints.185v2
- [55] Stefan Biffl and Marta Sabou (Feb. 2016). *Semantic Web Technologies for Intelligent Engineering Applications*. ISBN: 978-3-319-41488-1. DOI: 10.1007/978-3-319-41490-4
- [56] Deniztan Ulutaş Karakol et al. (May 2016). ‘Semantic definition and matching for implementing national spatial data infrastructures’. In: *Journal of Spatial Science* 61, pp. 1–19. DOI: 10.1080/14498596.2016.1142397
- [57] Reinaldo França et al. (Feb. 2021). ‘An Overview and Technological Background of Semantic Technologies’. In: pp. 1–21. ISBN: 9781799866992. DOI: 10.4018/978-1-7998-6697-8.ch001
- [58] Alexander Maedche and Steffen Staab (Feb. 2002). ‘Applying Semantic Web Technologies for Tourism Information Systems’. In: DOI: 10.1007/978-3-7091-6132-6\_32
- [59] Gustavo Rossi, Oscar Pastor et al. (Feb. 2008). *Web Engineering: Modelling and Implementing Web Applications*. ISBN: 978-1-84628-922-4. DOI: 10.1007/978-1-84628-923-1
- [60] Nathalie Moreno et al. (Feb. 2008). ‘An Overview Of Model-Driven Web Engineering and the Mda’. In: pp. 353–382. ISBN: 978-1-84628-922-4. DOI: 10.1007/978-1-84628-923-1\_12
- [61] Kevin Vlaanderen et al. (Feb. 2008). ‘Model-Driven Web Engineering in the CMS Domain: A Preliminary Research Applying SME’. in: vol. 19, pp. 226–237. ISBN: 978-3-642-00669-2. DOI: 10.1007/978-3-642-00670-8\_17
- [62] Ali Fatolahi et al. (2011). *Model-Driven Web Development for Multiple Platforms*
- [63] Pedro Valderas and Vicente Pelechano (Feb. 2011). ‘A Survey of Requirements Specification in Model-Driven Development of Web Applications’. In: *TWEB* 5, p. 10. DOI: 10.1145/1961659.1961664
- [64] José Rivero et al. (Feb. 2011). ‘Improving Agility in Model-Driven Web Engineering.’ In: vol. 734, pp. 163–170
- [65] J A Garcia-Garcia et al. (Feb. 2013). ‘NDT-Driver: A Java Tool to Support QVT Transformations for NDT’. in: pp. 89–101. ISBN: 978-1-4614-4950-8. DOI: 10.1007/978-1-4614-4951-5\_8

- [66] Thisaranie Kaluarachchi and Manjusri Wickramasinghe (2023). 'A systematic literature review on automatic website generation'. In: *Journal of Computer Languages* 75, p. 101202. ISSN: 2590-1184. DOI: <https://doi.org/10.1016/j.cola.2023.101202>. URL: <https://www.sciencedirect.com/science/article/pii/S2590118423000126>
- [67] Thisaranie Kaluarachchi and Manjusri Wickramasinghe (2024). 'WebDraw: A machine learning-driven tool for automatic website prototyping'. In: *Science of Computer Programming* 233, p. 103056. ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scico.2023.103056>. URL: <https://www.sciencedirect.com/science/article/pii/S0167642323001387>
- [68] Gustavo Rossi, Matias Urbieto et al. (Dec. 2016). '25 Years of Model-Driven Web Engineering: What we achieved, What is missing'. In: *CLEI Electronic Journal* 19.3. Special issue devoted to CIBSE 2016. DOI: 10.19153/cleiej.19.3.1. URL: <https://doi.org/10.19153/cleiej.19.3.1>
- [69] Veera Harish Muthazhagu and Surendiran B (2024). 'Exploring the Role of AI in Web Design and Development: A Voyage through Automated Code Generation'. In: *2024 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE)*, pp. 1–8. DOI: 10.1109/IITCEE59897.2024.10467409
- [70] Chris Lattner and Vikram S. Adve (2004b). 'LLVM: a compilation framework for lifelong program analysis & transformation'. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. URL: <https://api.semanticscholar.org/CorpusID:978769>
- [71] LLVM Discussion Forums (Aug. 2018). *Transpiler Question*. Accessed: 2024-08-10. URL: <https://discourse.llvm.org/t/transpiler-question/49589/4>
- [72] LLVM Documentation (Aug. 2024). *Writing an LLVM Backend*. <https://llvm.org/docs/WritingAnLLVMBackend.html>. Writing an LLVM Backend - LLVM 20.0.0git documentation. LLVM Project
- [73] Vaivaswatha Nagaraj et al. (2020). 'Compiling a Higher-Order Smart Contract Language to LLVM'. in: *CoRR* abs/2008.05555. arXiv: 2008.05555. URL: <https://arxiv.org/abs/2008.05555>
- [74] Dimitri Racordon (2021). 'From ASTs to Machine Code with LLVM'. in: *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*. URL: <https://api.semanticscholar.org/CorpusID:237246533>
- [75] Alon Zakai (2011). 'Emscripten: an LLVM-to-JavaScript compiler'. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, pp. 301–312. ISBN: 9781450309424. DOI: 10.1145/2048147.2048224. URL: <https://doi.org/10.1145/2048147.2048224>

- [76] Baer Darius (2010). 'Expectations for a Fourth Generation Language'. In: *Proceedings of the Conference on Fourth Generation Languages (4GL)*. URL: <https://api.semanticscholar.org/CorpusID:8164685>
- [77] Brijender Kahanwal (Oct. 2013). 'Abstraction Level Taxonomy of Programming Language Frameworks'. In: *International Journal of Programming Languages and Applications (IJPLA)* 3.4, pp. 1–12. DOI: 10.5121/ijpla.2013.3401
- [78] Union of RAD (2024). *Documentation*. <https://li3.me/docs/api/lithium/1.0.x/lithium/template/helper/Html::link/>. lithium::link() – Framework API v1.0.x – Documentation – li3 PHP-Framework. Union of RAD
- [79] Angular Team (2024). *Angular Documentation*. Available at: <https://angular.io/docs>
- [80] Express Team (2024). *Express Documentation*. Available at: <https://expressjs.com/en/starter/installing.html>
- [81] Mongoose Team (2024). *Mongoose Documentation*. Available at: <https://mongoosejs.com/docs/>
- [82] Auth0 Team (2024). *Auth0 Documentation*. Available at: <https://auth0.com/docs>
- [83] Redux Team (2024). *Redux Documentation*. Available at: <https://redux.js.org/introduction/getting-started>
- [84] Douglas Schmidt (Jan. 2006). 'Model-driven engineering'. In: *IEEE Computer, Computer Society* 39, pp. 41–47
- [85] Burak Yetiştiren et al. (2023). 'Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt'. In: *arXiv preprint arXiv:2304.10778*
- [86] Anh Nguyen Duc et al. (Oct. 2023). *Generative Artificial Intelligence for Software Engineering -A Research Agenda*. DOI: 10.2139/ssrn.4622517
- [87] Mario Bochicchio and Nicola Fiore (2005). *MODE: A Tool for Conceptual Modeling of Web Applications*. Via per Arnesano – 73100 – Lecce, Italy: Department of Innovation Engineering, University of Lecce
- [88] Ilma Ainur Rohma and Ade Azurat (June 2024). 'Code Generator Development to Transform IFML (Interaction Flow Modelling Language) into a React-based User Interface'. In: *Jurnal Ilmu Komputer dan Informasi* 17, pp. 109–120. DOI: 10.21609/jiki.v17i2.1178

.