Karlsruhe Institute of Technology

Software Design and Quality

Towards Continuous Integration of architectural Performance Models for Lua Applications

Master's Thesis of

Juan Saenz

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr.-Ing. Anne Koziolek
Second examiner:	Prof. Dr. Ralf Reussner
First advisor:	M.Sc. Martin Armbruster
Second advisor:	M.Sc. Manar Mazkatli

19. August 2024 – 19. February 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

As software systems become increasingly complex and large-scale, ensuring that quality requirements, such as performance, are consistently met throughout the software development lifecycle becomes increasingly challenging. Architecture-based Performance Prediction (AbPP) evaluates the performance characteristics of software systems based on architectural Performance Models (aPMs). When kept consistent with the evolving software systems, aPMs enable the assessment of design alternatives without requiring their full implementation. However, maintaining this consistency is a key challenge of AbPP.
Continuous Integration of architectural Performance Models (CIPM) has been introduced to address this challenge by integrating automated aPM updates into Continuous Integration (CI) and Continuous Delivery (CD) pipelines, reducing the effort required to keep models up to date. While CIPM has been successfully applied to Java-based projects, its generalisability to projects of other programming languages requires further investigation.
This thesis addresses two key issues related to evaluating the generalisability of the CIPM: (1) It presents a systematic, semi-automated approach for establishing a suitable data basis to investigate CIPM's generalisability, and (2) it analyses the challenges posed by dynamically typed programming languages for Code Model (CM) generation and serialisation, which is required within the CIPM approach to propagate changes from an incoming commit to the aPM. The latter is explored on the representative example Lua, and includes extension to the existing Lua CM generator and serialiser to address the identified challenges and ensure its applicability to general Lua software projects.
The evaluation of the semi-automated approach results in curated sets of real-world, open-source Lua and Java projects suitable for CIPM research, significantly reducing the manual effort required for their selection. In addition, the application of the extended Lua CM generator and serialiser to a subset of these projects demonstrates its effectiveness, despite the inherent challenges identified for static reference resolution in dynamically typed languages. While the evaluation also highlights certain limitations in the proposed approaches and the broader applicability of CIPM to dynamically typed languages, the results provide promising insights into its potential generalisability. Ultimately, this work contributes to enhancing the usability of the CIPM approach by addressing key obstacles to its broader applicability.

Zusammenfassung

Da Softwaresysteme immer komplexer und umfangreicher werden, wird es immer schwieriger sicherzustellen, dass die Qualitätsanforderungen, wie z. B. die Leistung, während des gesamten Lebenszyklus der Softwareentwicklung durchgängig erfüllt werden. Die Architekturbasierte Leistungsvorhersage (Architecture-based Performance Prediction, AbPP) bewertet die Leistungsmerkmale von Softwaresystemen auf der Grundlage von architektonischen Leistungsmodellen (aPMs). Wenn sie mit den sich entwickelnden Softwaresystemen konsistent gehalten werden, ermöglichen aPMs die Bewertung von Entwurfsalternativen, ohne deren vollständige Implementierung zu erfordern. Diese Konsistenzerhaltung ist eine der zentralen Herausforderungen von AbPP.
Der Ansatz der kontinuierlichen Integration von Architektur-basierten Leistungsmodellen (Continuous Integration of architectural Performance Models, CIPM) wurde eingeführt, um diese Herausforderung zu bewältigen. Das Ziel von CIPM ist es, automatische aPM-Aktualisierungen in Continuous Integration (CI) und Continuous Delivery (CD) Pipelines zu integrieren, und somit den Aufwand für die Konsistenzerhaltung der Modelle zu reduzieren. Während CIPM bereits erfolgreich auf Java-basierte Projekte angewandt wurde, bleibt die Verallgemeinerbarkeit des Ansatzes auf Projekte anderer Programmiersprachen weitgehend unbehandelt.
Die vorliegende Arbeit befasst sich mit zwei Kernfragen im Zusammenhang mit der Verallgemeinerbarkeit von CIPM: (1) Der Erarbeitung eines systematischen, halbautomatischen Ansatzes zur Schaffung einer geeigneten Datenbasis für die Untersuchung der Verallgemeinerbarkeit von CIPM, und (2) der Analyse der Herausforderungen, die dynamisch typisierte Programmiersprachen für die Generierung und Serialisierung von Code-Modellen (CM) im Rahmen des CIPM-Ansatzes darstellen, welche erforderlich sind, um Änderungen von einem eingehenden Commit zum aPM zu übertragen. Letzteres wird anhand des repräsentativen Beispiels Lua untersucht und umfasst die Erweiterung des bestehenden Lua-CM-Generators und -Serialisieres, um die identifizierten Herausforderungen zu bewältigen und seine Anwendbarkeit auf allgemeine Lua-Softwareprojekte sicherzustellen.
Die Evaluation des halbautomatischen Ansatzes ergibt zwei Mengen öffentlich zugänglicher Lua- und Java-Projekte, die sich für die Erforschung des CIPM Ansatzes eignen, unter minimiertem manuellen Aufwand für deren Auswahl. Darüber hinaus zeigt die Anwendung der erweiterten Lua-CM-Generierung und -Serialisierung auf eine Teilmenge dieser Projekte die Effektivität der Erweiterung. Während die Evaluation auch Einschränkungen in den vorgeschlagenen Ansätzen und die breitere Anwendbarkeit von CIPM auf dynamisch typisierte Sprachen aufzeigt, bieten die Ergebnisse vielversprechende Einblicke in seine potenzielle Verallgemeinerbarkeit. Letztendlich trägt diese Arbeit dazu bei, die Anwendbarkeit des CIPM-Ansatzes zu verbessern, indem wichtige Hindernisse für seine breitere Anwendbarkeit behandelt werden.

Contents

List of Figures

List of Tables

1. Introduction

As software systems become more complex and larger in scale, ensuring that quality requirements, such as maintainability, reliability, and performance [38], are consistently fulfilled throughout the software development lifecycle becomes increasingly challenging [1]. A software systems architecture, the allocation and composition of its components [74], plays a pivotal role in achieving these requirements [7, 18]. Consequently, evaluating the impact of different architectural design alternatives on quality requirements is imperative to the fulfilment of these requirements.
Performance is regarded as one of the most significant quality requirements [88, 40, 10], with some taxonomies assigning it to a distinct category [38]. The performance characteristics of architectural alternatives can be assessed by measurement-based evaluations or model-based predictions [75]. However, measurement-based approaches require the implementation of each design alternative, which can lead to a significant effort [90]. *Architecture-based Performance Prediction* (AbPP) offers a more cost-effective alternative by leveraging *architectural Performance Models* (aPMs) to simulate and predict performance [75, 63]. Nevertheless, for AbPP to remain accurate, aPMs must be continuously updated to remain consistent with the evolving codebase to maintain their accuracy [63], which is a key challenge of AbPP [24, 90].
Model-driven software development (MDSD) practices address the challenges associated with maintaining up-to-date models by combining formal models and model transformations to automatically ensure consistency between different models [96]. In the context of today's agile software development, in which software systems are often developed and maintained iteratively using *Continuous Integration* (CI) and *Continuous Delivery* (CD) pipelines, MDSD approaches must be integrated into these pipelines to fully leverage the benefits of AbPP.
To address this challenge, the *Continuous Integration of architectural Performance Models* (CIPM) approach has been proposed [63]. CIPM integrates automated aPM updates into CI/CD pipelines, ensuring models remain consistent with *Development-time* (Dev-time) or *Operation-time* (Ops-time) changes. By maintaining up-to-date models throughout the software lifecycle, CIPM enables accurate performance predictions at every stage while reducing modelling costs.

While CIPM has been applied and evaluated to Java-based projects [4, 63], the extent to which it can be generalised to different programming languages, technologies, and domains remains an open question. Initial research has explored the application of CIPM to Lua-based sensor applications [15, 62], but several limitations remain unaddressed.
Firstly, there is an absence of a systematic approach for establishing a suitable data basis to investigate CIPM's generalisability, such as the generalisability of the required generation and serialisation of *Code Model*s (CMs), which form the basis of CIPM's commit-based

model updates, or of the rules governing consistency preservation between architectural documentation and source code [54].
Secondly, the challenges posed by dynamically typed languages to CIPM's CM generation and serialisation require further evaluation, as the existing approach for Lua has notable limitations, which could impact the accuracy of CIPM's model updates.

The present thesis aims to address these limitations by introducing an approach for selecting evaluation cases suitable for CIPM research and analysing the challenges that dynamically typed programming languages pose for CM generation in the CIPM approach. Furthermore, it extends the existing Lua CM generation to mitigate these challenges. The results indicate that the proposed approach significantly reduces the manual effort required to identify suitable software projects for CIPM research, while the extensions to Lua CM generation effectively address the challenges introduced by Lua's dynamically typed nature.

The remainder of this thesis is structured as follows: Chapter 2 provides the necessary foundational background, Chapter 3 presents the methodological approach, Chapter 4 introduces the proposed solutions, and Chapter 5 evaluates both approaches, followed by a discussion of related work in Chapter 6. Chapter 7 explores future research directions based on the findings of this thesis. Finally, Chapter 8 provides a brief conclusion.

2. Foundations

This section provides the foundational background to the thesis. First, the field of Mining Software Repositories (MSR) is introduced in Section 2.1. Then, models and model-driven software development are described in Section 2.2. Subsequently, the Palladio component model is introduced as an example of an aPM in Section 2.3, and an overview of the Vitruvius approach is provided in Section 2.4. The Continuous Integration of architectural Performance Models (CIPM) approach is presented in Section 2.5. Section 2.6 focuses on the Code Model Generators and Serialisers used within the CIPM framework, a key aspect of this thesis. Finally, Section 2.7 introduces the Lua programming language, which is used as a representative example of a dynamically typed programming language in order to investigate the generalisability of the CIPM approach.

2.1. Mining Software Repositories

Mining Software Repositories (MSR) is a field within software engineering that focuses on analysing data from software repositories to uncover insights about software systems and projects [42, 49, 11, 81, 95, 41, 47]. This section provides a brief introduction of key aspects of MSR approaches relevant for this thesis.

The goals of MSR approaches include understanding software systems and team dynamics, analysing source code changes and bugs, and automating empirical studies [41]. To achieve these goals, MSR approaches employ data mining techniques to extract data from various data sources – such as source code repositories, bug repositories, and archived communications [81] – and analyse this data in relation to specific research goals [49, 47]. The analysed data may include structured data, such as source code from the repositories, and non-structured data, such as natural language texts from documentation artifacts, communications, or wikis [41, 11, 95]. The general process of MSR approaches consists of extracting data from repositories, processing this data, and analysing the resulting information to answer questions about the analysed software projects [47].

A key contribution of MSR is its ability to automate the collection of software projects as a data basis for empirical software engineering studies. This automation facilitates the reproducibility of findings through large-scale analyses, and supports investigations into the generalisability of research results [41].

2.2. Model-Driven Software Development

In a general sense, models are abstract representations of real-world concepts, phenomenons or entities, created to achieve a specific aim. As such, they are used across various disciplines, including science, economics, and software development to understand, analyze, or simulate complex theories, processes, or systems [14].
The three fundamental properties postulated in Stachowiak's General Model Theory [86] give a more precise definition of models:

> "1. **Mapping**: Models are always models of something, i.e. mappings from, representations of natural or artificial originals, that can be models themselves.
>
> 2. **Reduction**: Models in general capture not all attributes of the original represented by them, but rather only those seeming relevant to their model creators and/ or model users.
>
> 3. **Pragmatism**: Models are not uniquely assigned to their originals per se. They fulfill their replacement function a) for particular – cognitive and/ or acting, model using subjects, b) within particular time intervals and c) restricted to particular mental or actual operations." [66].

In software engineering, models have traditionally been used to document structural, behavioural, and deployment characteristics of software systems [87]. Nowadays, two forms of applying models to software development processes are differentiated. When used as part of the process, but not in a central role, the process is called *model-based*. When models are seen as "first-class citizens", i.e. primary artifacts, of the process, the process is called *model-driven* [14, 87]. The main disadvantage of model-based software development is the cost of maintaining consistent models throughout the software development lifecycle, given the dynamic nature of software systems and their evolving requirements [87].
In *model-driven software development* (MDSD), this disadvantage is mitigated through the models' formal nature and their automated implementation. MDSD combines formal models, defined using *Domain-Specific Languages* (DSLs), and *transformations* to automatically keep the consistency between different models, for example, models of different abstraction levels [87, 14]. To this end, MDSD relies on the principle that "Everything is a model": Not only is source code, for instance, seen as a model, but all models in turn are instances of more abstract meta-models [14]. Meta-models define how models of a domain can be created. They provide an abstract syntax, defining the structure of the language, and static semantics, defining rules for its well-formedness. A DSL then contains a meta-model and provides a concrete syntax, enabling the formal description of parts of a domain as instances of meta-models [87]. It should be noted that the term "language" does not refer exclusively to textual DSLs, but also encompasses "graphical languages with corresponding language-specific editors" [96, p.58].
Meta-models themselves are instances of meta-meta models. An example of a self-describing meta-meta model is the Meta Object facility (MOF), used to define meta-models in the Model-Driven Architecture (MDA). The MDA can be seen as a flavour of MDSD and is maintained

by the Object Management Group (OMG), the global standards development consortium responsible for the Unified Modeling Language (UML) [39, 87].

MDSD aims to increase development speed, reusability, and software quality by employing formal models and transformations as described above. *Architecture-Centric* MDSD, a specialized variant of MDSD focused on software architecture, primarily targets these three goals [87]. One method of applying formal architectural models, that significantly contributes to achieving these goals, is Architecture-based Performance Prediction (AbPP) using component-based architectural Performance Models (aPMs). These models encapsulate information relevant to performance analyses of a software system based on its architecture. For instance, resource demands and branch probabilities are modelled by so-called *Performance Model Parameters* (PMPs) [64]. As an example of such an aPM, the Palladio Component Model is presented in the next section.

2.3. The Palladio Component Model

The *Palladio Component Model* (PCM) is part of Palladio, an approach to creating models and performing analysis of component-based software architectures [75]. In this context, a component is defined as ”a contractually specified building block for software, which can be composed, deployed, and adapted without understanding its internals” [75, p. 47]. The PCM is a meta-model providing multiple DSLs to specify architectural software models. It is specifically designed to enable model-based predictions about the performance and reliability of the modelled system [12]. This section presents an overview of the concepts of the PCM relevant to this thesis. Further details can be found in [12, 75].

2.3.1. Synopsis of the Palladio Component Model

The Palladio approach defines four roles with corresponding models in the PCM: Component developers, software architects, system deployers, and domain experts [12]. These users define the PCM’s five submodels, the `Repository`, `System`, `Resource Environment`, `Allocation` and `Usage` Models, employing the corresponding DSLs provided by the PCM [63].
Component developers are responsible for specifying and implementing basic and composite components and depositing them into repositories, forming the `Repository` Model. Repositories are collections of `Interfaces`, `Components`, and `DataTypes` (cf. Fig. 2.1a) [12]. A `BasicComponent` is defined by the interfaces that it requires and provides. The *required* interfaces of a component define the services that it utilises, whereas the *provided* interfaces define the services that it offers. Basic components can be assembled to form composite components, which model more complex entities. The role of interfaces is not inherently providing or requiring; rather, their role is defined by the relationship between the components. The interaction between the required and provided services of a component is further specified via *Resource Demanding Service Effect Specifictions* (RDSEFFs) [12], which are presented in further detail in section 2.3.2.

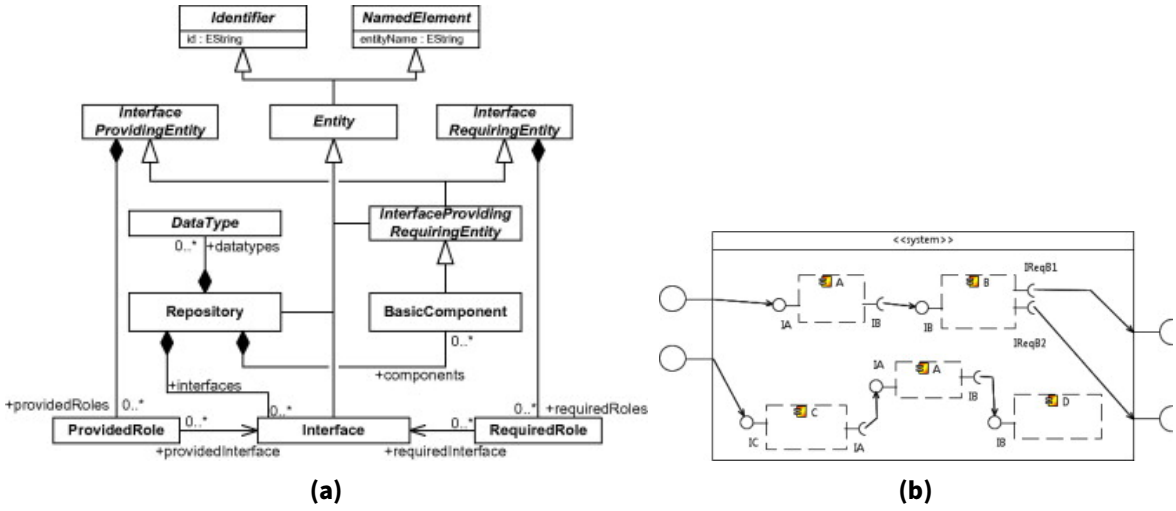


Figure 2.1.: The PCM repository meta-model (a), and an example of a system (b) [12].

Software architects specify `Systems` by embedding components in *assembly contexts* and connecting their required and provided interfaces via *assembly connectors* in the `System` Model. The use of assembly contexts allows for the differentiation of multiple instances of the same component within a system. As an example, consider component `A` in Figure 2.1b, where assembly contexts are denoted by dashed lines [12]. Systems expose the required and provided services of the components at the system's boundary via *delegation connectors*.
It is the responsibility of system deployers to model *resource environments* containing *resource containers* in the `Resource Environment` Model. These resource containers, in turn, comprise *concrete* instances of *abstract* resource types, which are defined in a global *resource repository*. Component developers reference the abstract resource types in the specification of RDSEFFs. System deployers then link the concrete resource instances to the abstract resource types, creating the `Allocation` Model, which allows for the incorporation of the allocation context in analyses conducted using the PCM.
Finally, domain experts utilise their specialized understanding of the software's domain to define `Usage` Models within the CPM. These models encompass usage scenarios that describe the anticipated number of concurrent users, as well as the influence of user behaviour on the control flow of the software and the specified parameters in RDSEFFs [12].

In combination, the PCM models provide a complete representation of all factors that affect component performance, namely the specifications of the components themselves, their composition, the context in which they are deployed, and their usage profiles.

2.3.2. Resource Demanding Service Effect Specifications

As previously stated, RDSEFFs specify the relationship between required and provided services within a given component. More specifically, component developers may provide an RDSEFF to any provided service offered by a component, specifying the control flow of calls to external services and modelling the abstract resource usage of the service in question [51].

Developers cannot know about the eventual deployment environments of the components, their concrete assembly in a system, or the usage profiles of the software. Therefore, they provide abstract resource types, that are later instantiated by system deployers and depend on concrete usage profiles [12].
Since components are meant to be composable, deployable, and adaptable without knowledge of their internal workings [75], only the information relevant to conducting model-based performance analyses is encapsulated in the RDSEFFs [51]. Thus, RDSEFFs contain a sequence of actions providing an *abstract* description of the control flow within a component. While the control flow between `ExternalCallActions` is modelled in RDSEFFs, it is mostly abstracted for internal actions. Only when there is an `InternalCallAction`, a call to a method that is only available within the specific component, the corresponding control flow is modelled. These methods are called `ResourceDemandingInternalBehaviours` and are only modelled if the method is a) called at least twice, and b) contains a call to an external service [54]. Other internal computations are modelled via `InternalActions`, omitting implementation details. Control flow model elements include, among others, `BranchActions`, `ForkActions`, and `LoopActions` [51]. `BranchActions` are used for either/or transitions, while `ForkActions` are used for concurrent executions. Each RDSEFF begins with a `StartAction` and ends with a `StopAction`. Actions other than `ExternalCallActions` are further specified by abstract resource demands. These can be expressed as constants or probabilities and may depend on input parameters [51]. An example consisting of a simplified service and the corresponding RDSEFF is given in Figure 2.2.

```
void execute(int number,
             List array){

  requiredService1();

  // internal computation
  innerMethod();

  if (number>=0)
    for (item in array)
      requiredService2();
  else
    requiredService3();
}
```

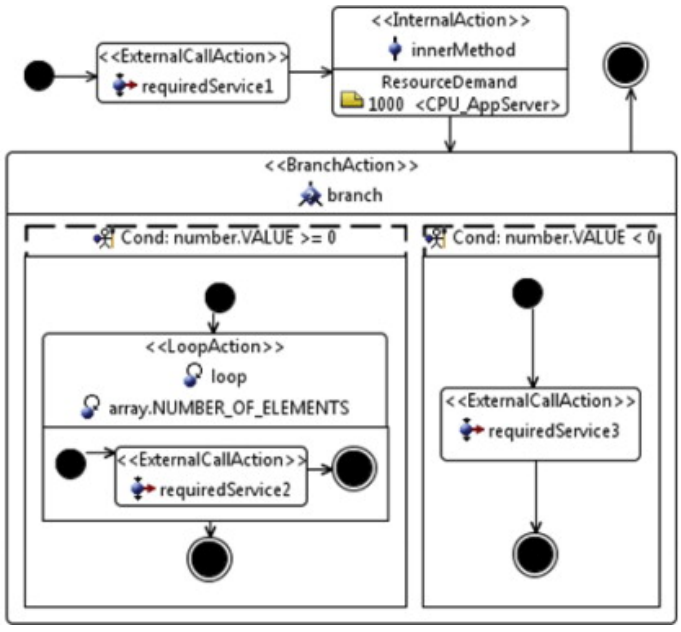


Figure 2.2.: Example source code for the service *execute* and corresponding RDSEFF [51].

2.4. The Vitruvius Approach

VIew-cenTRic engineering Using a VIrtual Underlying Single model (Vitruvius) is an approach for model-driven software development. The main goal of Vitruvius is to reduce the complexity of a complete system model by enabling users to work on models that align with their specific system-related concerns [52]. In the Vitruvius approach, the complete system model is represented by a virtual single underlying meta-model (V-SUMM). The V-SUMM internally consists of multiple meta-models, defining the respective models. An instance of the V-SUMM, the virtual single underlying model (V-SUM), then contains instances of these meta-models. Furthermore, the Vitruvius approach contains *Consistency Preservation Rules* (CPRs) to manage the transformations between the different models [50]. As the name implies, the CPRs ensure the consistency between the Vitruvius models. Upon modification of a model, the corresponding CPRs are triggered, updating the other models. The practical implementation of the Vitruvius approach, hereafter referred to as Vitruvius, also contains a *correspondence model*, which contains information about which elements of the different models are linked via CPRs [50].

2.5. Continuous Integration of architectural Performance Models

The objective of the Continuous Integration of architectural Performance Models (CIPM) approach is to reduce the overhead associated with the application of AbPP in iterative software development and maintenance. This is achieved by providing a method for the automatic preservation of aPM consistency throughout iterations. The CIPM approach integrates aPM updates into Continuous Integration (CI) and Continuous Delivery (CD) pipelines, resulting in accurate models and PMPs following Development-time (Dev-time) or Operation-time (Ops-time) changes [63]. This enables accurate AbPP at every step of the development process while reducing modelling costs [82].
While the CIPM approach addresses aPMs in general, the approach is currently mainly investigated with the PCM as a concrete aPM. Consequently, this thesis will henceforth address the CIPM approach regarding the PCM in particular, with the understanding that the PCM is a representative example of aPMs in general.

The following sections present a summary of the CIPM approach, tailored to the objectives of this thesis. Section 2.5.1 provides an introductory overview of the CIPM pipeline. In section 2.5.2, the Eclipse Modeling Framework (EMF), a toolset for MDSD employed in the current prototypical CIPM implementation, is introduced. The remaining sections offer details on how CIPM's models are automatically kept consistent with changes in the source code (section 2.5.3), the process of 2.5.4, as well the calibration and validation steps of CIPM (section 2.5.5). Unless otherwise indicated, the information regarding the CIPM approach and its current implementation is drawn from Mazkatli et al. [63], where the CIPM approach is described in greater detail.

2.5.1. Pipeline for Continuous Integration of Performance Models

The CIPM approach introduces the Model-based DevOps pipeline depicted in Figure 2.3, which in this thesis is referred to as the *CIPM pipeline*. The CIPM pipeline comprises a sequence of steps numbered 1-12, split into Dev-time and Ops-time actions. It begins on the Dev-side with the *CI-based update of software models* (1), where changes from an incoming commit are propagated to the Code Model (CM), Instrumentation Model (IM), and the PCM `Repository` and `System` Models situated within the Vitruvius V-SUM (cf. section 2.4). The IM serves as the basis for the *adaptive instrumentation* (2), which generates the instrumented source code utilised during *Performance testing* (3) to acquire the measurements necessary for PMP calibration. These measurements are generated by executing the instrumented code on a test environment. They are subsequently used to perform the *Incremental calibration* (4). The accuracy of the resulting PCM is validated on the same test environment through *Self-validation* (5). If no re-calibration is required, the resulting PCM can be used for *Architecture-based performance prediction* (6), thereby concluding the Dev-time activities of the CIPM pipeline.
The pipeline continues on the Ops side with the *Continuous deployment* (7) of the application, which enables *Monitoring* (8) on data measured using the production environment, and subsequent *Self-Validation* (9). Should the accuracy of the PCM be deemed insufficient, the model is recalibrated during *Ops-time calibration*. The resulting PCM can then be used to perform *Model-based analyses* (11), which inform further *Development planning* (12) leading into the next iteration of CI.

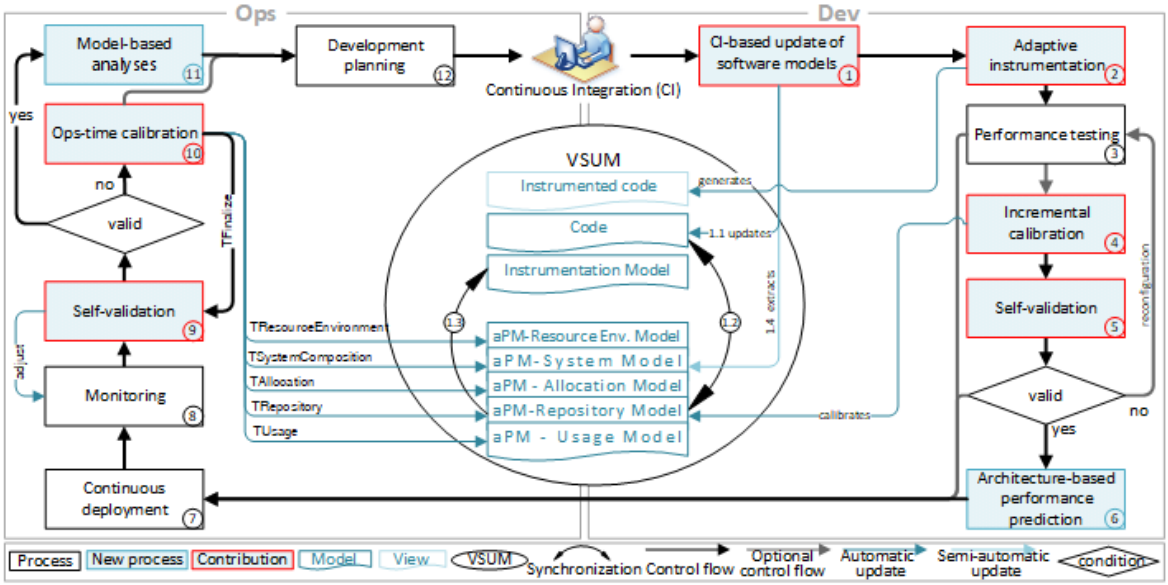


Figure 2.3.: The CIPM pipeline (highlighted contributions do not apply to this thesis) [82].

2.5.2. A Toolset for MDSD: The Eclipse Modeling Framework

As evidenced by the overview given in the previous section, a practical implementation of the CIPM pipeline is confronted with the challenge of numerous complex requirements. These include generating CMs from source code, ensuring consistency between models at various abstraction levels across different DSLs, and monitoring running applications. Fortunately, there are existing tools, frameworks, and approaches that can be leveraged for the existing prototypical implementation of CIPM.
One such framework is the *Eclipse Modeling Framework* (EMF), which plays a significant role in the current prototypical CIPM implementation. The EMF offers numerous tools for MDSD. One of the core elements of the EMF is *Ecore*, a meta-meta model that implements OMG's essential MOF (eMOF), which is a subset of the MOF referenced in section 2.2 [28, 87]. Ecore is utilized by various practical implementations of MDSD approaches. The PCM, for example, is implemented in Ecore [12], and Vitruvius (cf. section 2.4) supports all meta-models based on Ecore. The current CIPM implementation relies on the EMF not only for Ecore models, which are used in the CIPM implementation to enable consistency-preservation using Vitruvius, but also makes use of other EMF tools. For instance, *EMF Compare* [29] is employed during the CI-based update of software models for model comparison, as described in section 2.5.3.

2.5.3. CI-based Inter-Model Consistency

As previously stated, the first step in the CIPM pipeline is the *CI-based update of software models* (cf. Section 2.5.1). This step commences with the generation of a CM in response to an incoming commit. Currently, the prototypical CIPM implementation features a Java and a Lua CM.
Following the generation of the CM, a comparison is made between the resulting CM and the existing CM within the V-SUM using EMF Compare. In order to leverage EMF Compare, a matching strategy must be provided, capable of detecting differences in the corresponding CM versions. For instance, the existing CIPM implementation provides hierarchical matching engines for the Java [4] and Lua [15] CMs, which compare model elements based on their hierarchical location in the CM in addition to their attributes.
This model comparison process yields a sequence of atomic changes, which are then sorted and applied to the V-SUM CM, resulting in the subsequent updates of the `Repository` Model and IM according to the transformations defined by the CPRs.
Given that the CPRs are responsible for model-to-model transformations and component detection in the CM-to-PCM transformation, their implementation depends on the programming language, technology, and domain of the project. For instance, different CPRs are implemented for different CMs (e.g. Java vs. Lua CM), and the CPRs utilised to detect components in microservice-based applications [4] differ from those used to detect components in sensor-based applications [15].
Furthermore, the CPRs are utilised to detect any code statements relating to RDSEFF actions. This mapping is stored in the Vitruvius correspondence model for RDSEFF reconstruction. Changes to the RDSEFF in turn trigger changes to the IM, ensuring that all RDSEFF actions

have corresponding probes in the IM. For a detailed description of the RDSEFF reconstruction process, readers are referred to Burgey's work [15], while the utilisation of the IM in the adaptive instrumentation is covered in section 2.5.4.

In addition to the CM generation and the resulting model transformations described above, the CI-based update of software models updates the PCM `System` Model in response to an incoming commit. This step starts with the creation of a *Service-Call-Graph*, which is generated using static code analysis to identify method calls in the source code and their mapping to the previously constructed `Repository` Model. The Service-Call-Graph is then utilised to resolve the required and provided dependencies, starting from the system boundary, and traversing recursively through all contained assembly contexts. When updating the `System` Model via the Service-Call-Graph, conflicts may arise that must be resolved by a software architect or component developer. For instance, if a required service of one component is provided by multiple components in the `Repository` Model, an architect or developer must decide which component to employ. In the current CIPM implementation, this step only supports Java projects.

2.5.4. Adaptive Instrumentation

Following the CI-based model updates, the IM contains *instrumentation probes* for all RDSEFF actions. More specifically, the IM contains a `ServiceInstrumentationProbe` (SIP) for each RDSEFF, which contains `ActionInstrumentationProbes` (AIPs). The AIPs map to an `InternalAction`, `BranchAction`, or `AbstractLoopAction` within the corresponding RDSEFF [67].
In addition, the Vitruvius correspondence model provides a mapping from probes to RDSEFF actions, and from RDSEFF actions to code statements in the CM. The *adaptive instrumentation* utilises the IM and the Vitruvius correspondence model to create a copy of the CM, which is extended with monitoring statements, referred to as *instrumentation points*, for the SIPs and AIPs present within the IM. This instrumented CM (iCM) can then be serialised into instrumented source code via the CMoGS, which is utilised in both development and production environments to collect the measurement data required for CIPM's calibration steps. The adaptive instrumentation in the current CIPM implementation relies on *Kieker*, a monitoring framework for Java-based applications [92], and thus only supports Java projects.

2.5.5. Calibration and Validation at Dev-Time and Ops-Time

The execution of the instrumented source code generated by the adaptive instrumentation in a development or production environment produces the monitoring records employed for PMP calibration. The resulting records correspond to the instrumentation probes in the IM, i.e. a `service record` is created for every SIP, and an `internal action record`, `branch action record`, or `loop action record` is created for the corresponding AIPs.

The *incremental calibration* at Dev-time first calibrates the PMPs for `ExternalCallActions`, `BranchActions` and `LoopActions`. These PMPs are then used for the subsequent calibration of the `InternalAction` PMPs. The dependencies between input data and PMPs are identified through the use of decision trees for `BranchActions` and regression analysis for the other RDSEFF actions. The `Resource Environment`, `Allocation`, and `Usage` Models of the PCM instance are updated by the same transformation pipeline employed at ops-time, as discussed later in this section. The accuracy of the resulting PCM is then evaluated by the *Self-validation* at Dev-time, where the simulation data from the PCM is compared to the data from the monitoring records. If the accuracy is deemed insufficient, the data from the Self-validation is utilised in the re-calibration of the PMPs to increase the model accuracy.

The *Ops-time calibration* utilises the Ops-time monitoring data as input to a transformation pipeline developed by Monschein [67] to update the PCM models. The transformation pipeline comprises a series of steps that are too intricate to be detailed fully in this work. In summary, it:

1. updates the `Resource Environment` Model,
2. updates the `System` and `Allocation` Models, and
3. calibrates the PMPs in the `Repository` Model and extracts the `Usage` Model.

The resulting calibrated PCM is then validated in a manner analogous to the Dev-time validation. Details regarding the *Ops-time calibration* and the transformation pipeline can be found in [67, 63].

This concludes the overview of the CIPM approach and its pipeline. As demonstrated, the pipeline begins with the generation of a CM, and includes the serialisation of a CM to source code in the adaptive instrumentation. Given that the CM conversion is highly dependent on the programming language, it plays a critical role in determining the generalisability of the CIPM approach. Therefore, the next section focuses on CM conversion in the context of CIPM.

2.6. Code Model Generation and Serialisation

The application of CIPM to a software project relies on automatic generation and serialisation of CMs for the programming languages employed by the given projects. CMs are *generated* from the source code of an incoming commit in the CI-based update of software models, as well as *serialised* into instrumented source code in the adaptive instrumentation.
In previous CIPM literature, the tool employed for CM generation and serialisation was referred to as a *parser and printer* [15, 4], with the *Java Model Parser and Printer* (JaMoPP) being given a language-specific name. The term *Code Model Generator and Serialiser* (CMoGS) is introduced here to describe these tools for general programming languages, and to emphasise that, in the context of CIPM, CM generation encompasses more than merely the conventional parsing of source code into an abstract syntax tree (AST) - it also involves the *reference resolution* between model elements, which will be addressed in more detail in Section 2.6.3.
At present, the prototypical CIPM implementation provides CMs and the respective CMoGS for the Lua and Java programming languages. Armbruster [4] extended JaMoPP [45], which is used for CM conversion of Java CMs, while Burgey [15] extended the Xtext-based grammar from the Melange project [65] in the implementation of the Lua CMoGS. Sections 2.6.1 and 2.6.2 provide more information about the respective CMoGS, followed by a discussion on reference resolution in programming languages and its relevance to the CM conversion process in Section 2.6.3.

2.6.1. The Java Code Model Generator and Serialiser: JaMoPP

The Java Model Parser and Printer (JaMoPP), the CMoGS employed in the current CIPM implementation, was originally developed by Heidenreich et al. [45]. One of their goals was to address the "gap between modelling and programming languages" [43, p. 1] by enabling the usage of EMF modelling tools for Java programs. This version of JaMoPP was designed on the basis of EMFText [44], a tool capable of generating generators and serialisers from textual syntax model representations defined in CS, a specification language designed for EMFText [43]. The original version of JaMoPP supported the Java syntax up to version 6, and was augmented by Armbruster [5] to support up to Java 15. Henceforth, JaMoPP will refer to this augmented version.
In addition to parsing and serialising Java CMs as Ecore models, JaMoPP provides mechanisms for resolving references between model elements [5]. During CM generation, references are initially represented as *proxy objects*, which serve as placeholders. When these proxies are accessed for the first time, JaMoPP resolves them to the actual referenced model elements.

2.6.2. The Lua Code Model Generator and Serialiser: Based on Xtext

The Lua CMoGS utilised in the current CIPM implementation was designed by Burgey [15] using the *Xtext* framework. Xtext supports the development of DSLs using the *Xtext grammar language* [32]. As Xtext is based on ANTLR, a parser generator that uses an $LL(*)$ algorithm [70], grammars defined in Xtext's grammar language cannot include *left-recursive* or *ambiguous* rules [32]. Similar to EMFText, Xtext can generate both generators and serializers based on a syntax definition in the *Xtext grammar*, which defines the meta-model of the language [20]. The meta-models generated in this way are instances of the Ecore meta-meta-model and, as such, can be integrated with other EMF-based tools and frameworks [22, 30]. Like EMFText, the mechanisms for reference resolution provided by Xtext initially represent references as proxy objects during AST parsing, which are resolved when accessed for the first time [30]. The key mechanism for reference resolution in Xtext are *Scope Providers*, which are responsible for determining the set of possible elements that a reference could refer to [31].

Burgey [15] implemented the original Lua CMoGS by extending the Lua Xtext grammar from the Melange project [65], which supports the parsing of Lua 5.1 files to individual models. In addition, they implemented an Xtext Scope Provider for reference resolution. To represent entire projects as a single Lua CM, Burgey [15] introduced specific synthetic rules into the existing Xtext grammar. In this thesis, *synthetic rules* refer to grammar rules that specify meta-model elements that do not directly correspond to the Lua syntax but serve as abstractions. The extended Xtext grammar makes use of synthetic rules to group multiple model elements representing Lua files into a single Lua CM by introducing an `App` model element. Other synthetic rules were added for convenience. For instance, a `BlockWrapper` rule encapsulates information about "all meta-model classes that contain a `Block` of statements" [15, p. 22], and a `Referenceable` rule represents a superclass for all meta-model elements that can be referenced.
In addition, Burgey [15] implemented a reference resolution process that connects related Lua CM elements, such as linking function calls to their corresponding function declarations. Although Burgey's [15] investigations yielded promising results with respect to their Lua CMoGS implementation, they also acknowledge certain limitations. These include limited support of the Lua syntax specification [71], as well as limitations of the reference resolution process, arising from the lack of compile-time type information.

2.6.3. Reference Resolution in Code Model Generators

In the context of programming languages, *reference resolution* (also denoted as "name resolution" [80, 3] or "identifier resolution" [2]) refers to the process of determining which *entity* (e.g. value, function, object) a given *identifier* refers to [2, 68, 80]. For our purposes, an identifier denotes any character string that represents an entity.
Andova et al. [2] provide the following definition of reference resolution (denoted as "identifier resolution" in their work):

> "A declaration establishes a *binding* of an identifier I to some entity X. An *applied occurrence* is an occurrence of I where X is made available through I [...]. Identifier resolution establishes so-called *use-def* relations. Every applied occurrence should correspond to exactly one binding occurrence." [2, pp. 35-36]

An example of a *binding* and an *applied* occurrence is given in Listing 2.1. Note that identifiers that occur multiple times in listings regarding reference resolution are differentiated by their subscripts in this thesis, whenever relevant to the discussion of the listing. For instance, $\texttt{a}_1$ and $\texttt{a}_2$ in Listing 2.1 denote different occurrences of the same identifier `a`.
The declaration in line 2 of that listing denotes a binding occurrence, binding the identifier $\texttt{a}_1$ to the entity of type `A` returned by some function `getA`. Line 3 contains an applied occurrence of the same identifier (denoted as $\texttt{a}_2$), which makes the bound entity `A` available through the identifier $\texttt{a}_2$. Reference resolution implies resolving the reference from $\texttt{a}_2$ over $\texttt{a}_1$ to the entity `A`.

```
1 ...
2 A a1 = getA();
3 a2.setX(10);
4 ...
5
6
7
8
```

Listing 2.1: Java code illustrating binding and applied occurrences.

```
1 class A {
2     private int x = 0;
3     ...
4     public void setX(int x) {
5         this.x = x;
6     }
7     ...
8 }
```

Listing 2.2: Example declaration of class `A` from Listing 2.1.

In high-level programming languages, such as Java and Lua, binding and applied occurrences of identifiers manifest as *declarations* of and *references*, respectively. A declaration binds a variable (an identifier) to an entity, while a reference makes the entity available through the variable. Thus, the process of reference resolution in CMoGS is also referenced to as *modelling reference-declaration relationships* in this thesis.

Reference resolution is closely related to scoping, as a *scope* determines which bindings are active in a specific region of code [80, 2]. In languages with *lexical* scoping, all bindings are determined at compile-time. The *referencing environment*, which represents the set of active bindings at any given point during the execution of a program, can be determined by examining a sequence of scopes defined by the language's *scope rules*. The reference resolution process uses the referencing environment to identify the active binding for a given applied occurrence of an identifier [80].
Another important concept related to reference resolution is typing. *Types*, especially user-defined types (e.g., classes in Java), constitute an integral part of the scoping rules of a language [80, 3]. The scope defined by the type [3] of an entity delimits the referencing environment for any applied occurrence of an identifier relating to a *property* of the given entity's type. Here, a property denotes any entity directly associated to – and made accessible through – another entity, such as methods and fields of Java classes. To illustrate this point,

consider the code snippets presented in Listings 2.1 and 2.2. In Listing 2.1, when resolving the reference to the property `setX` on line 3, the referencing environment will include the scope of class `A`, as specified in Listing 2.2. This enables the reference resolution process to use the scope associated with the type declaration to accurately identify the binding of `setX` in line 4 of Listing 2.2. An identifier such as a_2, which precedes another identifier, delimits the scope of the subsequent identifier through the properties of the given entity. Such preceding identifiers are also referred to as a *scope resolution operator* or *qualifier* [80].

As described previously, the CMoGS employed in the prototypical CIPM implementation execute reference resolution during the generation of the respective CMs. Given that CM generation in CIPM relies only on the information available in an incoming commit, CMoGS rely on *static reference resolution*, i.e. the reference resolution process is constrained to information available at compile time. The CMoGS then process this information to resolve all proxy objects in the ASTs following the initial parsing of the source code to the actual referenced model elements (cf. Sections 2.6.1 and 2.6.2). Consequently, the reference resolution process transforms the ASTs into the graph structure that represents the CM [44]. Once all references have been resolved, the changes in the CM can be propagated to the IM and aPM.

In order to resolve the references between model elements, CMoGS can leverage the referencing environment, which can be determined through scopes and types, as described earlier. However, in many popular programming languages, such as Python, Lua, and JavaScript, type information is only available at run-time. These languages are referred to as *dynamically typed*, in contrast to *statically typed* languages, which provide type information at compile-time [21]. As type information is crucial in determining the referencing environment, CMoGS for dynamically typed languages face the challenge of resolving references without the benefit of this information.

2.7. The Lua Programming Language

Lua is an interpreted, lexically scoped and dynamically typed programming language written in clean C and designed primarily as an extension programming language [71]. It is widely used in areas such as game development, embedded systems and web development [17, 15]. Since its conception in 1993, the language has evolved through various versions, the most recent being 5.0-5.4 [46, 57]. This section provides a brief overview of the Lua language features relevant to the context of this thesis.

As a dynamically typed programming language information about the type of any Lua value – `string`, `boolean`, `number`, `function`, `nil`, `userdata`, `thread`, and `table` – is only available at run-time [71]. Lua provides a single data structure, the `table`, which essentially functions as an associative array that can be indexed by any Lua value except `nil` [71, 85]. As the sole data structure, tables are integral to many Lua programs, which has led to the notion that "In Lua, everything is a table" [23].

All Lua values, including `functions`, are *first-class values* and can be passed as arguments to functions, returned as results, and assigned to variables [71]. Variables are identified by a single name consisting of letters, underscores and digits. Unless explicitly declared as local using the `local` keyword, any variable is considered global.
Lua offers standard libraries implemented in C, which provide functionalities such as string manipulation, mathematical operations, and input/output handling [71].

```
1    f = function () end -- function declaration
2    function func() end -- syntactic sugar for function declaration
3
4    table["a"] -- table index access
5    table.a    -- syntactic sugar for table index access
```

Listing 2.3: Examples of syntactic sugar supported by Lua.

Lua provides syntactic sugar for certain language constructs such as table indexing and function declarations, as shown in Listing 2.3. Note that the respective languages' comment syntax, `"--"` for Lua and `"//"` for Java, is used in this thesis to provide explanations in Listings.

Given the dynamic nature of Lua and its previous use in the CIPM approach [15, 62], it is an interesting candidate for investigating the generalisability of CIPM.

3. Methodological Approach

The methodology of this thesis is structured according to the PRICoBE pattern [84]. This chapter introduces the initial five sections of this pattern: In section 3.1, the **P**roblems addressed by this thesis are introduced, followed by the resulting **R**esearch questions in section 3.2. The general **I**dea of this thesis is presented in 3.3, leading to its **Co**ntributions and resulting **B**enefits described in sections 3.4 and 3.5, respectively.
The last section of the PRICoBE pattern, the **E**valuation process, is described in section 5.2 of chapter 5.

3.1. Problems

As stated in section 2.5, the CIPM approach was implemented for Java applications in prior research [4], where it was evaluated on microservice-based applications. An initial investigation into the generalisability of the CIPM approach to other programming languages was conducted by Burgey [15].
From this initial work, following observations can be made regarding the investigation of the generalisability of the CIPM approach to other programming languages: The code model generation, adaptive instrumentation and `System` model extraction are directly dependent on the programming language, necessitating language-specific implementation and evaluation. While the CPRs are only implicitly dependent on the programming language through the language-specific code model, they are explicitly dependent on the technologies used in a project and its domain. Therefore, they represent an additional aspect of the practical application of CIPM that requires adjustment when evaluating the CIPM approaches' generalisability. With regard to the Dev-time and Ops-time calibration and validation steps of the CIPM pipeline, further investigation is required to determine whether any adaptation to new technologies and programming languages is necessary.
In summary, the first two steps of the CIPM pipeline, the *CI-based update of software models* and the *adaptive instrumentation* (cf. Section 2.5.1), are known to require language-specific implementation and evaluation, while the remaining steps do not directly depend on the programming language in use and may or may not be applicable without adaptation.
In their work, Burgey [15] focused on the adaptation and evaluation of the first step, including an evaluation of the generation and update of the Lua CM, the PCM, and the IM in response to an incoming commit on sensor-based applications. Although this work yielded promising results regarding the generalisability of CIPM, it also revealed challenges in the application of the *CI-based update of software models* to dynamically typed programming languages, especially in the CM generation, which will be discussed in more detail shortly.

In addition, further investigation is required into the remaining steps of the CIPM pipeline to evaluate the CIPM approach's generalisability to other programming languages. This forms the central problem statement of this thesis:

> **Central Problem Statement.** The CIPM approach in its current state only supports Java-based applications for the complete CIPM pipeline. The generalisability of the complete CIPM pipeline to other programming languages requires further evaluation.

This statement is explored in more depth in the remainder of this section, with a view to the challenges identified in previous research, resulting in the more specific problems that this thesis aims to address.

Burgey's [15] investigations into the generalisability of the CIPM approach to the Lua programming language focused on SICK AppSpace [94] applications. Unfortunately, these case studies are not openly accessible (with the exception of some samples [93]), which complicates further investigation into the generalisability of the CIPM approach. This is because, without the consent of the proprietors of these case studies, researchers must select other suitable case studies, which decreases the comparability of the results. This issue is further compounded by the fact that although the requirements for the specific case studies evaluated in previous CIPM research have sometimes been discussed in the respective publications (e.g. [4, 15]), statements as to *general* requirements of software projects suitable for the evaluation of CIPM are not readily available. Consequently, the process of assembling comprehensive sets of openly accessible projects that serve as a data basis for investigations into the CIPM approach is rendered more arduous. However, establishing such a data basis is essential for various research objectives related to CIPM, such as investigations into it's generalisability. For instance, it enables investigations into the generalisability of CPRs based on the analysis of different software projects [97] and the applicability of the CIPM approach to projects of new programming languages[15, 62]. These observations are summarised by this thesis' first problem statement:

P1 The absence of a systematic approach for the selection of openly accessible software projects for CIPM evaluation hinders the comparability of CIPM research.

- **P1.1** There is no comprehensive set of requirements for assessing a software project's suitability for investigating the generalisability of the CIPM approach.
- **P1.2** The selection of software project for research investigating the generalisability of the CIPM approach is inefficient due to the lack of automation.
- **P1.3** There is no structured overview of openly accessible projects that are suitable for investigations into the generalisability CIPM approach.

As mentioned previously, the investigations conducted by Brugey [15] revealed challenges in the application of the CI-based update of software models inherent to dynamically typed programming languages. Burgey's analysis highlighted the absence of types as a factor hindering the implementation of type scoping in the reference resolution of the CM generator, and they proposed an extension to the `Repository Model` by a Lua *any type* as a

solution to this issue. In addition, the author emphasises the complexity of implementing "correct resolution of variables throughout the code model" [15, p. 30] as a hindrance to their CM generator supporting the complete Lua syntax.
During the preliminary analysis of suitable case studies for this thesis, it became evident that these limitations of the original Lua CM generator proposed by Burgey render it impractical for general application. Although successful in their evaluations of SICK AppSpace apps, the limited support for the Lua syntax impedes the parsing of projects that make use of the unsupported syntactical constructs. Additionally, reference resolution, one of the main characteristics of the Java CM generator [45, 43, 5], is reportedly not correctly implemented by the original Lua CM generator [15]. These limitations suggest challenges in the design of CM generators for dynamically typed programming languages that require further investigation, leading to this thesis' second problem statement:

P2 The generalisability of the CIPM approach's CM generation to dynamically typed programming languages requires further evaluation.

- **P2.1** There is a lack of in-depth analysis of the challenges in the design of CM generators for dynamically typed programming languages.
- **P2.2** The original Lua CM generator does not support the complete Lua language.
- **P2.3** The original Lua CM generator does not adequately support reference resolution.
- **P2.4** The original Lua CM generator is not evaluated on openly accessible software projects.

The problem statements **P1** and **P2**, while concentrating on specific elements of the central problem statement, are fundamental to its examination, as they concern the foundations of the application of the CIPM approach to dynamically typed programming languages.

3.2. Research Questions

Based on the problems described above, this thesis focuses on the following research questions.

R1 How can openly accessible software projects for CIPM evaluation be selected systematically?

- **R1.1** What are key requirements that determine a software project's suitability for investigating the generalisability of the CIPM approach?
- **R1.2** How can the effort of selecting suitable software projects for investigating the generalisability of the CIPM approach be reduced?
- **R1.3** How can an overview of openly accessible projects suitable for investigations into the generalisability of the CIPM approach be established?

R2 Can the CIPM approach be applied to dynamically typed programming languages?

R2.1 What are the requirements for a CM generator applicable in CIPM?

R2.2 What are the challenges in satisfying these requirements for dynamically typed programming languages such as Lua?

R2.3 How can the missing Lua language constructs be supported by the Lua CM generator?

R2.4 How can the reference resolution be supported by the Lua CM generator?

3.3. Idea

The objective of this thesis is to address the aforementioned research questions. To this end, this thesis presents an approach for the semi-automated collection of software projects as a data basis for investigating the generalisability of the CIPM (**R1**), followed by the an analysis of the challenges of designing CM generators for dynamically typed programming languages in the context of CIPM and an approach to extending the existing Lua CM generation to address these challenges (**R2**).

In relation to **R1**, the requirements for software projects suitable for investigating the generalisability of the CIPM approach are analysed and discussed to serve in the design of a semi-automated approach for CIPM evaluation case selection. This approach is inspired by the general process of MSR approaches (cf. Section 2.1) in that data is extracted from source code repositories, processed and analysed to automate evaluation case selection. It should be noted, that this thesis focuses on the generation of collections of software projects from the perspective of selecting evaluation cases for investigations into the CIPM approach, given their value for the subsequent investigations into **R2**. However, such an approach can also be valuable to generate collections of software projects for other goals, such as analysis of commonalities of various software projects for the design of generalised CPRs, as performed for instance by Weng [97].

In relation to **R2**, the thesis derives requirements for CM generators applicable to the CIPM approach, drawing from the functionalities of the CM generator used in the prototypical implementation of the CIPM approach for Java (JaMoPP). The potential challenges in meeting these requirements when designing CM generators for dynamically typed programming languages, such as Lua, are explored, and the Lua CM generator is extended to address these challenges, representing a key contribution of this thesis. Specifically, the underlying Xtext grammar is expanded to fully capture Lua's syntax, and a more comprehensive reference resolution mechanism is introduced to improve the accuracy and completeness of the generated code models.

The evaluation in this thesis focuses on the application of the extended Lua CM generator to a selection of openly accessible Lua projects, derived from the evaluation cases identified in response to **R1**. This allows for an initial evaluation of the generalisability of the extended CM generator, and by extension, the CIPM approach, to dynamically typed languages such as Lua.

3.4. Contributions

This thesis makes the following contributions based on the research questions and the idea described in the previous sections.

C1 Systematic approach to semi-automated selection of evaluation cases for research regarding the generalisability of the CIPM approach.

- **C1.1** Collection of requirements to guide the selection of evaluation cases for CIPM research.
- **C1.2** Semi-automated process for the generation of a set of evaluation cases based on these requirements.

C2 Evaluation of the applicability of CIPM to dynamically typed programming languages.

- **C2.1** Collection of requirements for CIPM CM generators.
- **C2.2** Extension of the Lua CM generator to support the complete language and more accurately resolve references.
- **C2.3** Evaluation of the extended Lua CM generator.

3.5. Benefits

The contributions described above result in the following benefits.

B1 Increased efficiency in the selection of evaluation cases for CIPM research.

B2 Generalisation of the applicability of the CM generator to Lua-based projects by supporting the complete Lua language including reference resolution.

4. Approach

This chapter presents the two key approaches explored in this thesis. Section 4.1 introduces the approach to Semi-Automated CIPM Evaluation Case Selection. Section 4.2 analyses the challenges of dynamically typed programming languages for CIPM Code Model Generators and Serialisers (CMoGS) and outlines the proposed extensions to the original Lua CMoGS to address these challenges.

4.1. An Approach to Semi-Automated CIPM Evaluation Case Selection

The *approach to Semi-Automated CIPM Evaluation Case Selection* proposed in this thesis is designed as a streamlined, configurable, and lightweight approach to semi-automated selection of suitable evaluation cases for CIPM research. It's core principles are a set of *Evaluation Case Requirements for CIPM* and a *process for Semi-Automated CIPM Evaluation Case Selection*, both of which are presented in the following sections. Section 4.1.1 discusses the differences between the conventional notion of software requirements and evaluation case requirements, followed by the definition of a fundamental set of *CIPM Evaluation Case Requirements* in Section 4.1.2. Section 4.1.3 provides an overview of automated methods for the collection of *requirement assessment data* (RADs). A set of RADs corresponding to the set of evaluation case requirements is provided in Section 4.1.4. The *process for Semi-Automated CIPM Evaluation Case Selection*, which makes use of the automated methods and RADs, is defined in Section 4.1.5, followed by a concise description of its prototypical implementation in Section 4.1.6. Finally, the generalisability of the approach to Semi-Automated CIPM Evaluation Case Selection is explored in Section 4.1.7.

4.1.1. A Note on the Term "Requirements"

Before examining requirements for CIPM evaluation cases, it is important to highlight the subtle differences between the conventional notion of *software requirements* in software engineering, and the term *evaluation case requirements* introduced here. In contrast to the conventional understanding of software requirements, which can be exemplified by Glinz's [37] definition as

> "1. A need perceived by a stakeholder.

2. A capability or property that a system shall have.
3. A documented representation of a need, capability or property."[37, p. 17],

evaluation case requirements refer to the suitability of software projects as evaluation cases, as opposed to the attributes of software systems. An *evaluation case* is the object of study in a case study [77], and in the context of this thesis refers to a software project on which a research goal is evaluated. Evaluation case requirements are not intended to specify the capabilities and properties of the software itself; rather, they identify the characteristics that make a software project suitable for use as an evaluation case in specific research contexts.

To improve readability, this thesis uses the term "requirement" in both senses: to refer to the characteristics of suitable evaluation cases in the remainder of this section (Section 4.1), as well as in the conventional sense when addressing the specification of requirements for CMoGS in Section 4.2. In instances where the distinction between software and evaluation case requirements is not immediately apparent from the context, explicit differentiation will be made.

4.1.2. A Set of Evaluation Case Requirements for CIPM

In previous CIPM research, the requirements that guided the selection of the corresponding evaluation cases were often formulated according to the specific research questions ([15, 4]). The objective of this section is to compile a set of fundamental CIPM evaluation case requirements for the approach to Semi-Automated CIPM Evaluation Case Selection, by generalising and extending these existing requirements. The generalisation process entails the detachment of existing requirements from the specific research questions, thereby adapting them to the broader context of CIPM research. In addition, the extension process will introduce fundamental requirements for CIPM case studies that were not explicitly addressed in prior research.
Note that the compilation of a comprehensive, categorised, and detailed set of these requirements falls outside the scope of this thesis. Instead, the objective is to establish a preliminary set of high-level requirements to streamline and guide the selection of suitable evaluation cases, laying the groundwork for future expansion.

The specification of the requirements in the remainder of this section is structured as follows: Each identified requirement is briefly introduced in a highlighted section in form of a gray box. Requirements are given a number (REQ$\mathbf{i}_{EC}$), followed by a concise description. For requirements directly generalised from previous works, this description is followed by a reference to the corresponding work(s). Additionally, the categorisation into *essential* (**E**) and *research-goal dependent* (**D**) is indicated. *Essential* requirements are general to any CIPM related research, whereas the degree as to which *research-goal dependent* requirements need to be fulfilled varies depending on the specific research goals. Each highlighted section is followed by a brief discussion of the respective requirement and its categorisation.

REQ1$_{EC}$ Validated architectural documentation is available for the project. [4] **E**

This requirement addresses the need for validated architectural documentation in any form. Such documentation may include, but is not limited to, pre-existing textual or graphical representations of the system's architecture provided by the project maintainers, or an aPM manually specified by the researchers and validated by the project's architect. The presence of validated architectural documentation is essential for any research using the CIPM approach, as it provides the necessary reference for evaluating the aPM generated through the CIPM process. Armbruster [4] presents a related, albeit more specific, version of this requirement, which emphasises the existence of a manually specified PCM instance for a specific commit. However, their version does not explicitly require the PCM instance to be validated.

REQ2$_{EC}$ The project has a Git commit history consisting of several commits. [15, 4] **E**

The CIPM approach is centred around the integration of automatic aPM updates into CI/CD pipelines based on propagating changes from incoming commits to the aPM. As such, it is essential for the project to have a Git repository with a sufficient number of commits. The precise number of commits required is dependent on other factors, such as the project scale as described in **REQ9**$_{EC}$.

REQ3$_{EC}$ The Git commit history of the project contains a combination of architecturally relevant and architecturally irrelevant commits. [4] **E**

In order to evaluate the accuracy of the updates to the aPM by the CIPM approach, it is essential to consider a combination of architecturally relevant and architecturally irrelevant commits. This combination is necessary to ensure that updates only occur when a commit actually contains architecturally relevant changes.

REQ4$_{EC}$ The project is an open-source project with an openly accessible Git repository. [4] **E**

This requirement emphasises that for any CIPM-related research to be reproducible, it must be evaluated using at least one open-source project with a publicly accessible Git repository. Armbruster [4] proposed a similar requirement, stating that "publicly accessible repository allows the independent repetition of the evaluation." [4, p. 49]. In addition to the reproducibility aspect, Armbruster [4] highlights the ease-of-use resulting from conventional open source licences. In light of the necessity for comparability and verifiability in research, it is imperative that any CIPM-based evaluation ensures that at least one evaluated project is open-source and its repository is publicly accessible.

REQ5$_{EC}$ Automated performance tests are available for the project. **D**

In addition to the updating of the aPM's structural representation of the project architecture, the CIPM approach updates the model's PMPs, enabling performance predictions with the resulting aPM. To perform the calibration and validation of these PMPs, and to evaluate the accuracy of the resulting performance predictions, automated performance tests for the

project are required. However, not all CIPM-related research addresses the complete CIPM pipeline (e.g., [67, 54, 4, 15]). Indeed, even the present thesis is focused on a portion of the pipeline. For CIPM research that does not involve the direct evaluation of the accuracy of the performance prediction using the resulting aPM, the existence of automated performance test is less relevant. Thus, this requirement is categorised as research-goal dependent.

REQ6$_{EC}$ The project is actively developed or maintained. **D**

This requirement ensures that the CIPM approach is evaluated on projects which are currently in development or being maintained. This offers benefits for various CIPM-related research goals, for instance when investigating the actual practical integration of the CIPM pipeline into a running CI/CD pipeline. It also increases the chances of the availability of project actors (e.g. architects, developers, deployers) for inclusion into the research project. Furthermore, fulfilling this requirement can support the fulfilment of other requirements. For instance, an aPM manually specified by the researchers could be validated in cooperation with a project architect (**REQ1**$_{EC}$), or architecturally relevant commits identified with the help of the project maintainers (**REQ3**$_{EC}$). However, given that these benefits impact specific research goals to a greater extent than others, this requirement is categorised as research-goal dependent.

REQ7$_{EC}$ The technologies used in the project and its domain are aligned with the problems the given research intends to address. [4, 15] **D**

As discussed in Section 2.5.3 and Section 3.1, the CPRs used by the CIPM approach depend on the specific technology and domain of the project CIPM is applied to. Research relating to the generalisability of the CIPM approach, for instance, could focus on evaluating multiple projects with diverse technologies and of various domains. Examples of specific instances of this requirement can be found in previous work; Armbruster [4], for example, specified this requirement as "The case study is a Java- and Microservice-based application." [4, p. 49], while Burgey's [15] evaluations were performed on sensor-based applications.

REQ8$_{EC}$ The programming languages used in the project are aligned with the problems the given research intends to address. [15, 4] **D**

As demonstrated by Burgey's [15] research and further explored in the present thesis, the CI-based update of software models and adaptive instrumentation of the CIPM approach are dependent on the programming language used in the project. Consequently, it is crucial to select projects that use programming languages that are well-suited to the specific goals of the CIPM-related research being conducted. If the goal is to evaluate the generalisability of the CIPM approach, projects with programming languages that have not yet been evaluated – or multiple projects with differing programming languages – could be selected as evaluation cases. In contrast, if the research focuses on goals independent of programming language, the selection should prioritise projects that use programming languages already supported by the prototypical implementation of CIPM. This helps avoid additional implementation overhead related to supporting new languages.

REQ9$_{EC}$ The project's scale, in terms of lines of code and number of commits, is aligned with the problems the given research intends to address. **D**

In order to investigate the scalability of the CIPM approach, the lines of code (LOC) of a project, as well as the number of commits, can be considered. Since the CMoGS currently employed by the CIPM approach work on the entire code-base of the project, a correlation between LOC and the execution time of the generation and serialisation performed by a CMoGS can be expected. Furthermore, to evaluate the efficiency of the CIPM approach on multiple commits, the number of commits is a relevant metric.

4.1.3. Automated Collection of Requirement Assessment Data

Following the establishment of a set of general evaluation case requirements, the next step is to perform an assessment of the satisfaction of these requirements in relation to a collection of evaluation case candidates.
The most straightforward but also the most effort-intensive method for this step, especially when considering an extensive collection of potential evaluation cases, is **Manual Inspection** ($\mathbf{A}_{\text{Manual}}$). The approach to Semi-Automated CIPM Evaluation Case Selection is aimed at reducing this effort, by leveraging *automated methods* to collect RADs in an Automated Filtering Process. This section provides an overview of various automated methods to support the assessment of requirement satisfaction, focusing on practical solutions without delving too deeply into the intricacies of each approach. Applying any of these methods to an evaluation case candidate results in RADs, which are the subject of the following section. The *Automated Filtering Process* is described in Section 4.1.5.

The automated methods are presented in order from least to greatest expected effort to implement and/or execute. Each requirement from the previous section will be assigned to the method with the least expected effort required to provide sensible support in the assessment of its satisfaction. Table 4.1 provides an overview of the requirements, their short description for later reference, their categorisation into essential or research-goal dependent, and the corresponding automated methods that require the least expected effort for a meaningful assessment of their satisfaction.

Project Statistics and Metadata ($\mathbf{A}_{\text{Stat}}$). This relatively low-effort method is applicable to the assessment of requirements that can be evaluated using easily accessible statistics or metadata from Git repositories, as well as information available through common repository management tools such as *GitHub* [33] and *GitLab* [91]. The requirements REQ2$_{EC}$ (Several Commits), REQ4$_{EC}$ (Open-Source), REQ6$_{EC}$ (Activity), REQ8$_{EC}$ (Pr. Language) and REQ9$_{EC}$ (Scale) can be assessed with this approach, as RADs providing information regarding these requirements, such as the number of commits, commit frequency over time, licensing information, and main programming language, are attainable through this method. A key benefit of this method, in comparison to the two methods that will be introduced subsequently, is that it does not always require additional validation through $\mathbf{A}_{\text{Manual}}$, thus making it the method with the least expected effort.

Requirement	Short description	Category	Method with lEE
$REQ1_{EC}$	Arch. Doc.	**E**	$\mathbf{A}_{\text{Keyword}}$
$REQ2_{EC}$	Several Commits	**E**	$\mathbf{A}_{\text{Stat}}$
$REQ3_{EC}$	Arch. Changes	**E**	$\mathbf{A}_{\text{ReMl}}$
$REQ4_{EC}$	Open-Source	**E**	$\mathbf{A}_{\text{Stat}}$
$REQ5_{EC}$	Perf. Tests	**D**	$\mathbf{A}_{\text{Keyword}}$
$REQ6_{EC}$	Activity	**D**	$\mathbf{A}_{\text{Stat}}$
$REQ7_{EC}$	Tech. & Domain	**D**	$\mathbf{A}_{\text{ReMl}}$
$REQ8_{EC}$	Pr. Language	**D**	$\mathbf{A}_{\text{Stat}}$
$REQ9_{EC}$	Scale	**D**	$\mathbf{A}_{\text{Stat}}$

Table 4.1.: Overview of the evaluation case requirements for CIPM, their categorisation (**E**ssential or research-goal **D**ependent), and the methods for the assessment of their satisfaction with the **l**east **E**xpected **E**ffort (lEE).

Simple Keyword Search ($\mathbf{A}_{\text{Keyword}}$). Another relatively low-effort method applicable to the assessment of requirement satisfaction is the analysis of keyword and keyphrase occurrences in file names, directory names, and file contents within a project's repository. It is expected to be effective in the assessment of $REQ1_{EC}$ (Arch. Doc.) and $REQ5_{EC}$ (Perf. Tests). General keywords such as `"architectural"` and `"architecture"` ($REQ1_{EC}$) or `"tests"` ($REQ5_{EC}$) can be detected, as well as more specific key phrases like `"architectural documentation"` or keywords and -phrases related to specific technologies, such as test frameworks like *JUnit* [48]. The number of occurrences of a set of keywords and -phrases for a given requirement then form a RAD. However, the results of this method are not definitive, as it does not consider the context in which the keywords appear. Therefore, the findings from this method should be validated through $\mathbf{A}_{\text{Manual}}$.

Advanced Data Mining, Reverse Engineering & Machine Learning ($\mathbf{A}_{\text{ReMl}}$). For the purposes of this thesis, $\mathbf{A}_{\text{ReMl}}$ encompasses all advanced data mining, reverse engineering, and machine learning methods that go beyond $\mathbf{A}_{\text{Stat}}$ and $\mathbf{A}_{\text{Keyword}}$. These methods are also applicable to the assessment of requirement satisfaction, making them relevant to the discussion of the proposed approach to Semi-Automated CIPM Evaluation Case Selection. They are expected to aid in the assessment of the requirements not covered by the other methods, $REQ3_{EC}$ (Arch. Changes) and $REQ7_{EC}$ (Tech. & Domain). However, due to the significant expected effort of these methods, they are excluded from the approach to maintain its lightweight nature. Consequently, they are not discussed in further detail.

4.1.4. A Set of Requirement Assessment Data

As mentioned in the previous section, the automated methods are used in the approach to Semi-Automated CIPM Evaluation Case Selection to collect *Requirement Assessment Data* (RAD). A RAD_{i,R_jP_k} provides information relevant to the assessment of requirement R_j for project P_k. RADs are used to filter the evaluation case candidates, as well as support

$\mathbf{A}_{\text{Manual}}$ in the selection of the final evaluation cases through the corresponding information provided by a RAD.
This section illustrates the notion of RADs by providing an initial selection of RADs and their corresponding filters for the set of Evaluation Case Requirements for CIPM, building upon the examples provided in the previous section for each automated collection method. It should be noted that this selection excludes RADs for REQ3_{EC} (Arch Changes.) and REQ7_{EC} (Tech. & Domain), as these are not automatically collected by the presented approach. An overview of the RADs and corresponding filters specified in this section for each requirement is given in Table 4.2. The extensibility of the proposed selection is discussed at the end of this section.

Requirement	Method with lEE	Example RAD	Example Filter
REQ1_{EC} (Arch. Doc.)	$\mathbf{A}_{\text{Keyword}}$	$OCC(KW_A)$	$OCC(KW_A) \geq 1$
REQ2_{EC} (Several Commits)	$\mathbf{A}_{\text{Stat}}$	$\lvert C \rvert$	$\lvert C \rvert \geq 100$
REQ3_{EC} (Arch. Changes)	$\mathbf{A}_{\text{ReMl}}$	-	-
REQ4_{EC} (Open-Source)	$\mathbf{A}_{\text{Stat}}$	License ls	$ls \in LS$
REQ5_{EC} (Perf. Tests)	$\mathbf{A}_{\text{Keyword}}$	$OCC(KW_T)$	$OCC(KW_T) \geq 1$
REQ6_{EC} (Activity)	$\mathbf{A}_{\text{Stat}}$	AVG_{C/W_n}	$AVG_{C/W_n} \geq 1$
REQ7_{EC} (Tech. & Domain)	$\mathbf{A}_{\text{ReMl}}$	-	-
REQ8_{EC} (Pr. Language)	$\mathbf{A}_{\text{Stat}}$	MPL	MPL = "Lua"
REQ9_{EC} (Scale)	$\mathbf{A}_{\text{Stat}}$	$LOC(\texttt{.lua})$	$LOC(\texttt{.lua}) \geq 1000$

Table 4.2.: Overview of the evaluation case requirements for CIPM, the methods for the assessment of their satisfaction with the **l**east **E**xpected **E**ffort., an example RAD, and an example for a filter for the RAD.

REQ1_{EC} (Arch. Doc). This requirement can be assessed by the number of occurrences of a set of keywords and keyphrases relating to the requirement. It is denoted here as

$$OCC(KW),$$

where KW represents a set of suitable keywords for the detection of requirement-related keywords. For instance, the keywords "architecture" and "architectural" can be used as a set for the detection of architecture documentation. This RAD can be configured further by specifying the locations in which these keywords and keyphrases are searched: in file names, directory names, file contents, or any combination. In the remainder of this thesis, this specification is denoted through sets of tuples such as

$$KW_A = \{(\text{"architecture"}, dfc), (\text{"architectural"}, c)\},$$

with the first entry specifying the keyword and the second entry specifying the searched locations. The locations are denoted by f (file names), d (directory names), c (file contents), or a combination (e.g., dfc) for each keyword or keyphrase.
A filter for $OCC(KW)$ is defined by a threshold for the minimum number of occurrences.

Given that a single document containing architectural documentation suffices to satisfy this requirement, the recommended filter is defined as

$$OCC(KW) \geq 1.$$

As this RAD is generated by $\mathbf{A}_{\text{Keyword}}$, the detected occurrences need to be validated by $\mathbf{A}_{\text{Manual}}$ after the Automated Filtering Process to ensure the satisfaction of REQ1_{EC} (Arch. Doc.).

REQ2_{EC} (Git Repo.). This requirement can be assessed by the total number of commits for the given repository. This RAD is denoted as $|C|$, where C refers to the set of all commits. Again, this RAD can be filtered by configuring a threshold. The precise configuration of the threshold is research-goal dependent.

REQ4_{EC} (Open-Source). The RAD for this requirement is specified by the licence ls of the given project. This RAD can be collected by $\mathbf{A}_{\text{Stat}}$, for example via the Github REST API [34]. A set of open-source licenses, designated as LS, can be used to define the filter: The requirement is satisfied iff ls is contained in LS. This filter is denoted as $ls \in LS$.

REQ5_{EC} (Perf. Tests). As with REQ1_{EC} (Arch. Doc.), the RAD for this requirement is specified as $OCC(KW_T)$ and a threshold filter. Here, KW_T refers to a set of keywords and keyphrases suitable for performance test detection.

REQ6_{EC} (Activity). The assessment of project activity is achieved through the utilisation of the average number of commits over the preceding n weeks as RAD, denoted as AVG_{C/W_n}. As with the other RADs expressing quantities, a threshold is employed as a filter for this RAD.

REQ8_{EC} (Pr. Language). The RAD proposed for this requirement is the main programming language MPL detected in the project. It can be detected, for example, by comparing the LOC of all files with programming language-specific file extensions, or by leveraging metadata provided by the API of a repository management tool like Github. The filter for this RAD compares the detected main programming language to the desired programming language, for instance MPL = "Lua".

REQ9_{EC} (Scale). Two RADs are specified for this requirement, the number of commits $|C|$ as described for REQ2_{EC} (Several Commits), and $LOC(ext)$, where ext denotes the file extension of the files considered in LOC computation. The corresponding filters are given by thresholds for $|C|$ and $LOC(ext)$. Their exact specification depends on the requirements regarding the scale of evaluation cases for the given research goals.

Extensibility of RADs. The RADs discussed in this section represent an initial set designed to address the corresponding requirements introduced in Section 4.1.2. These RADs can be extended in the future by introducing additional RADs to accommodate further assessment needs.
For instance, an additional RAD could be introduced for requirements assessed through $\mathbf{A}_{\text{Keyword}}$ by assigning weights to keywords and keyphrases. This would allow more specific keywords or keyphrases to have a greater influence on the Automated Filtering Process, adjusting the precision of the assessment.

Similarly, if collecting a license for $REQ4_{EC}$ (Open-Source) through $\mathbf{A}_{\text{Stat}}$ is not a possibility, for example if no repository management tool providing a corresponding API is accessible, an alternative RAD could be generated through $\mathbf{A}_{\text{Keyword}}$. This would involve specifying a set of keywords and keyphrases KW_{ls} specific to detecting license information in a project, to generate a corresponding RAD $OCC(KW_{ls})$.
Another example extends the RADs for $REQ6_{EC}$ (Activity). In this case, the activity of a project could be further assessed by a RAD containing information about the ratio of open to closed issues on Github, or the number of merged pull requests within a given timeframe.
In addition, RADs relating to the requirements $REQ3_{EC}$ (Arch Changes.) and $REQ7_{EC}$ (Tech. & Domain) were excluded in the presented selection, based on the assumption that suitable RADs cannot be generated for these requirements through $\mathbf{A}_{\text{Stat}}$ or $\mathbf{A}_{\text{Keyword}}$. For instance, $\mathbf{A}_{\text{Keyword}}$ could be used to detect keywords and keyphrases related to architectural changes in commit messages to assess $REQ3_{EC}$, but it seems improbable that commit messages would contain accurate information about architectural changes in the respective commits.
However, by compromising on the lightweight nature of the approach presented in this thesis, it may be possible to generate RADs for these requirements using methods from $\mathbf{A}_{\text{ReMl}}$. As such, the approach could be extended to include automated filtering for these requirements by incorporating RADs collected through $\mathbf{A}_{\text{ReMl}}$.

4.1.5. A Process for Semi-Automated CIPM Evaluation Case Selection

The *Process for Semi-Automated CIPM Evaluation Case Selection* proposed in this section leverages $\mathbf{A}_{\text{Stat}}$ and $\mathbf{A}_{\text{Keyword}}$ within an *Automated Filtering Process* to minimise the effort of manual inspection ($\mathbf{A}_{\text{Manual}}$). As discussed in the previous sections, this process excludes $\mathbf{A}_{\text{ReMl}}$ to maintain a lightweight nature. Figure 4.1 illustrates the process.

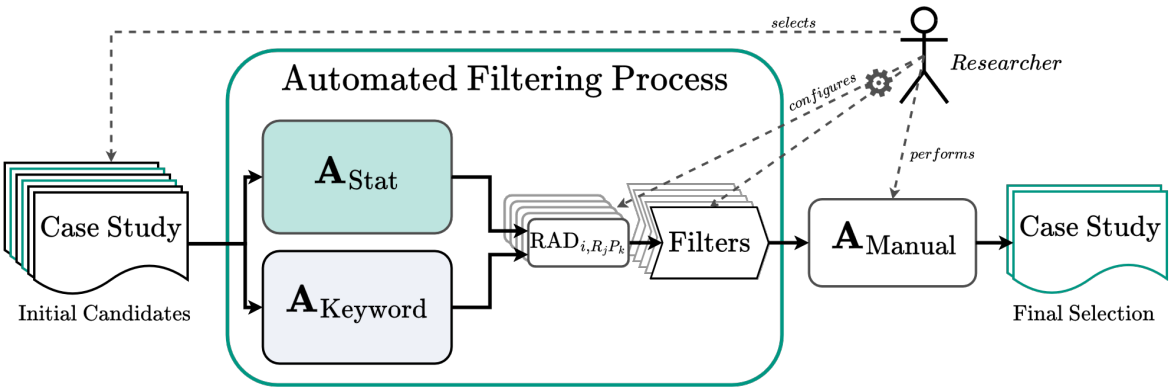


Figure 4.1.: The process of Semi-Automated CIPM Evaluation Case Selection.

The process begins with a set of *initial candidates* selected by the researcher, which are processed by applying $\mathbf{A}_{\text{Stat}}$ and $\mathbf{A}_{\text{Keyword}}$ to generate a set of RADs for each candidate. The resulting set of RADs for each project is then filtered by applying the corresponding filter for each RAD, yielding a set of *filtered RADs*. As outlined in the previous section, the specific RADs and filters can be configured by the researchers to align with the specific goals of the

evaluation, allowing for flexibility and adjustments. The filtered set of RADs supports the subsequent manual inspection step ($\mathbf{A}_{\text{Manual}}$) in the final evaluation case selection.
$\mathbf{A}_{\text{Manual}}$ benefits not only from the reduced number of candidates resulting from the Automated Filtering Process, but also from the information provided by the RADs. For instance, in circumstances where evaluation cases of varying scale are required for the evaluation of a particular research goal, the RADs for REQ9_{EC} (Scale) can guide the selection of evaluation cases that meet this requirement.

Note that, by excluding $\mathbf{A}_{\text{ReMl}}$ from the Automated Filtering Process, the manual inspection does not benefit from this process regarding requirements REQ3_{EC} (Arch. Changes) and REQ7_{EC} (Tech & Domain). Thus, only a subset of the requirements REQ1_{EC}-REQ9_{EC}, given by

$$REQ_{AFP} = \{\text{REQ1}_{EC}, \text{REQ2}_{EC}, \text{REQ4}_{EC}, \text{REQ5}_{EC}, \text{REQ6}_{EC}, \text{REQ8}_{EC}, \text{REQ9}_{EC}\}$$

is assessed by this process. Nevertheless, the filtering process and the preprocessed information from the RADs are expected to significantly reduce the effort required for evaluation case selection.

The combination of the set of *Evaluation Case Requirements for CIPM* specified in Section 4.1.2 and the *Process for Semi-Automated CIPM Evaluation Case Selection* presented in this section form the *Approach to Semi-Automated CIPM Evaluation Case Selection* proposed in this thesis. This approach streamlines the selection of evaluation cases by providing a set of requirements and a semi-automated assessment process. It can be applied to various CIPM-related research goals, through the configuration of the collected RADs and applied filters in the *Automated Filtering Process*. In addition, the approach is lightweight, focusing on fundamental requirements and using low expected effort methods to generate RADs. However, it is also extensible, allowing for adaptation if specific research goals necessitate it.

4.1.6. Implementation of the Automated Filtering Process

This thesis implements a prototypical Automated Filtering Process as outlined in the previous section. This implementation processes a set of initial evaluation case candidates in the form of Github project names (i.e. `OWNER/REPO`), generating a set of RADs for each project. First, the repository for each candidate is cloned, followed by a keyword- and LOC analysis. The Github REST API [34] is leveraged for all RADs generated through $\mathbf{A}_{\text{Stat}}$ except $LOC(ext)$. For this RAD, the lines for all files with the corresponding extension ext are computed and aggregated. The implementation employs concurrency to process candidates, and allows the configuration of two timeouts, t_{cloning} and $t_{\text{candidate}}$, to avoid excessive execution times. t_{cloning} limits the duration of the repository cloning process, while the duration of the complete data collection process, encompassing both the cloning process and the keyword- and LOC analysis, is limited by $t_{\text{candidate}}$. It is therefore recommended that these timeouts are selected such that $t_{\text{candidate}} \geq t_{\text{cloning}}$ holds. Given that not all projects can be processed by this implementation – for instance in the case that the Github API returns an invalid response, or that one of the timeouts is exceeded – the implementation returns sets of RADs

for a number of *processed candidates*, which is a subset of the initial candidates.
The resulting sets of RADs per project are then filtered using configurable filters, producing an output set of RADs for all projects that successfully pass the filtering process. The set of RADs and filters used in the implementation align with those specified in Section 4.1.4.

4.1.7. Generalisability of the Approach to Semi-Automated CIPM Evaluation Case Selection

The approach to Semi-Automated CIPM Evaluation Case Selection proposed in this thesis, although developed within the context of CIPM-related research, is generalisable to other research fields. The requirement set, automated methods applied to RAD collection, the RADs themselves, and the corresponding filters can be extended and adapted to suit other research contexts that rely on software projects as evaluation cases. This section explores the generalisability of the proposed set of Evaluation Case Requirements for CIPM to other research areas.

REQ1$_{EC}$ (Arch. Doc) can be generalised to any research field investigating software architecture, where the availability of architectural documentation is essential for the evaluation of research goals, such as research fields situated in the general area of Architecture-Centric MDSD (cf. 2.2).
REQ2$_{EC}$ (Several Commits) can be generalised to research on the evolution of the code-bases of software projects, as the presence of multiple commits in the project's history is crucial in assessing changes over time.
REQ3$_{EC}$ (Arch. Changes) applies wherever architectural changes throughout a project's lifecycle are studied. An example of this is research investigating architectural degradation over a software project's lifetime. In this sense, this requirement can be regarded as a more restrictive version of REQ2$_{EC}$ (Several Commits).
REQ4$_{EC}$ (Open-Source) can be generalised to any research that follows open science policies [73], or that aims to ensure the comparability and verifiability of results (e.g., FAIR principles [98]). In such cases, software projects used as evaluation cases must satisfy this requirement to enable open access to the project's code and ensure its reusability, thereby facilitating the replication and comparability of findings.
REQ5$_{EC}$ (Perf. Tests) is generalisable to research areas focused on software performance, such as software performance engineering or quality assurance research. In these fields, the availability of automated performance tests can support the evaluation of the evaluated approaches by providing performance measurements.
REQ6$_{EC}$ (Activity), similar to REQ2$_{EC}$ (Several Commits), can be generalised to research on the evolution of software projects. Furthermore, the satisfaction of this requirement indirectly ensures the active involvement of project stakeholders, which increases the likelihood of having project actors available for research that involves their participation.
REQ7$_{EC}$ (Tech. & Domain) and **REQ8**$_{EC}$ (Pr. Language) can be generalised to research focusing on specific technologies, domains or programming languages. For instance, as demonstrated by the research topic of this thesis, research exploring the generalisability of existing approaches to new technologies, domains or programming languages requires

evaluation cases with corresponding chracteristics.
Finally, **REQ9**$_{EC}$ (Scale) can be generalised to any research where the scale of evaluation cases influences their suitability for evaluating the respective research goals. For example, studies focused on approaches enabling the scalability of software systems may require projects of varying size to assess how well an investigated approach performs as the considered systems grow in scale. Similarly, research on software maintenance and refactoring can benefit from evaluating projects of different scales to understand how codebase size impacts the effectiveness of maintenance strategies.

As discussed in this section, the approach to Semi-Automated CIPM Evaluation Case Selection can be generalised to a wide range of research fields. Each requirement in the set of Evaluation Case Requirements for CIPM may apply to various research areas, making the approach suitable for diverse research contexts that utilise software projects as evaluation cases. Moreover, additional requirements, along with corresponding RADs and filters, can be specified in the future to further enhance the approach's generalisability. Overall, these points indicate the applicability of the approach's core principles to the selection of evaluation cases across multiple research domains.

4.2. An Approach to Code Model Generation and Serialisation for Dynamically Typed Programming Languages

This section outlines the approach to designing *Code Model Generators and Serialisers* (CMoGS) for dynamically typed programming languages. Section 4.2.1 defines the requirements for CMoGS employed in the CIPM approach, drawing upon prior CIPM research. Section 4.2.2 examines the challenges associated with satisfying one key requirement – reference resolution – when designing CMoGS for dynamically typed programming languages. Section 4.2.3 then discusses recovery mechanisms as a means to address these challenges. Next, Section 4.2.4 assesses the limitations of the original Lua CMoGS in relation to the defined requirements. Sections 4.2.5 and 4.2.6 present the extended Lua CMoGS proposed in this thesis, which addresses these limitations, thus enhancing its usability for the CIPM approach. Section 4.2.7 provides a brief overview of the implementation. Finally, Section 4.2.8 describes the integration of the extended Lua CMoGS into CIPM's CI-based update of software models.

4.2.1. Requirements for Code Model Generators and Serialisers

As outlined in Section 2.6, the current prototypical implementation of the CIPM approach includes CMoGS for the Java and Lua programming languages. The Lua CMoGS is a more recent addition and has not yet been extensively researched. Furthermore, certain limitations have been identified [15], as briefly summarised in Section 3.1 and discussed in greater depth later in this chapter (cf. Section 4.2.4). In contrast, the Java CMoGS, JaMoPP, has been widely used in research related to CIPM [54, 4, 67, 63], rendering it more suitable for the analysis of requirements for CMoGS in the context of CIPM. Based on the functionalities provided by JaMoPP [45, 43], the following requirements are derived for CMoGS:

REQ1$_{CM}$ The CMoGS defines an *Ecore meta-model* for the complete syntax of the programming language.

For CIPM to be applicable to any project of the given programming language, the complete syntax of the language must be represented by the meta-model. In addition, in order to guarantee compatibility between the CMoGS and the current CIPM implementation, it is necessary for the meta-model to be defined as an Ecore model. This aspect of **REQ1**$_{CM}$ provides the added advantage of enabling the use of the resulting CM in combination with other EMF-based MDSD tools.

REQ2$_{CM}$ The CMoGS provides a *generator*, capable of transforming source code into instances of the meta-model, and a *serialiser*, capable of transforming instances of the meta-model into source code.

The first step of the CIPM pipeline, the CI-based update of software models, requires the generation of CMs from source code (cf. Section 2.5.3), while the second step of the pipeline, the adaptive instrumentation, requires the serialisation of source code from the CMs in

order to generate the instrumented code (cf. Section 2.5.4), resulting in $REQ2_{CM}$. The CM generation and serialisation must be valid and accurate. In the case of generation, validity refers to the conformity of the CM to the meta-model, whereas accuracy refers to the one-to-one representation between the original code and the CM [6]. In the case of serialisation validity refers to the generation of valid Lua code by the CMoGS, and accuracy refers to the one-to-one representation between CM elements and corresponding Lua language constructs.

REQ3$_{CM}$ The CMoGS *resolves all references* between model elements *as closely as possible* to the programming language's semantics, ensuring that reference-declaration relationships are accurately modelled.

Reference resolution is a crucial component of CMoGS, given that the generation of RDSEFFs and, consequently, the accuracy of the performance predictions with the resulting aPM are contingent upon it. The classification of control flow elements within RDSEFFs (see Section 2.3.2), for instance, is directly influenced by reference resolution: a function call can only be classified as either `ExternalCallAction` or `InternalCallAction`, if the referenced function can be resolved. In the CIPM pipeline, reference resolution is performed statically in response to an incoming commit. Consequently, a CMoGS is only capable of processing static information. As will be discussed in the following section, this results in challenges for CMoGS in the context of dynamically typed programming languages, where static reference resolution cannot resolve all references in the general case.

4.2.2. Reference Resolution for Dynamically Typed Programming Languages

As outlined in Section 2.6.3, reference resolution benefits significantly from the availability of type information. CMoGS for statically typed programming languages can effectively leverage these advantages. However, since dynamically typed programming languages do not provide type information at compile-time, the static reference resolution mechanisms in CMoGS for these languages cannot rely on this information. The challenges arising from the absence of types in static reference resolution are explored further in this section, using the Lua programming language as an example.

The complexity of static reference resolution in the absence of types can be illustrated as follows. In statically typed programming languages, the return types of functions and types of entities are known at compile-time. The reference resolution process can use this type information to resolve accesses to properties of entities, such as the fields of a Java Object, by performing a lookup on the associated types. In contrast, in dynamically typed programming languages, this type information is not available statically. Consequently, the reference resolution process must compute all properties of an entity whenever one of the entity's properties is accessed. This involves evaluating return values from function calls as well as evaluating expressions that specify entity property access, such as indexes in Lua's tables.

Consider Listings 4.1 and 4.2, which contain the Java code already presented in Listing 2.1 from Section 2.6.3 alongside the equivalent Lua code.

```
1 ...
2 A a1 = getA();
3 a2.setX(10);
4 ...
```

Listing 4.1: Java code illustrating binding and applied occurrences from Section 2.6.3.

```
1 ...
2 local a1 = getA()
3 a2:setX(10)
4 ...
```

Listing 4.2: Lua code corresponding to the Java code in Listing 4.1.

In the Lua case, the return value of the function `getA()` is not statically known. In contrast to the Java case, where the properties of the entity referenced by a_1 can be inferred from the type `A` returned by `getA()`, the properties in the Lua case depend on run-time information, as they cannot be fully determined without knowing the value returned by `getA`. To compute these properties, the reference resolution process would need to evaluate the return value of `getA` in order to resolve the final reference to `setX(10)`.
A similar problem is exemplified in Listing 4.3. As Lua allows access to tables using any value except `nil`, arbitrarily complex expressions can be used to access their fields.

```
1 x = t["x"]   -- table access using string key
2 x = t.x      -- table access using syntactic sugar for string key
3 x = t[indexExp()] -- table access using arbitrarily complex expression
```

Listing 4.3: Examples of table accesses in Lua.

According to Rice's Theorem [76], all "non-trivial semantic properties of programs [...] are undecidable" [21, p.25]. In the context of static reference resolution, this implies that determining the return value of function calls or expressions statically is undecidable in the general case. Consider the example in Listing 4.4, which is based on a similar example showing the undecidability of static analysis resulting from conditionals given in [53].

```
1 getA = function()
2     if math.random() > 0.5 then -- math.random() returns float in [0,1)
3         return {x = 0} -- new table containing field 'x'
4     else
5         return {y = 0} -- new table containing field 'y'
6     end
7 end
```

Listing 4.4: Example of a Lua function requiring run-time information for return value resolution.

Without knowledge about the value returned by the `random` function, static analysis cannot determine which table is returned by `getA`. As a consequence, static reference resolution in dynamically typed programming languages can only be performed approximately. In

other words, not all references can be automatically resolved by this process in the general case.

As discussed at the end of the previous section, reference resolution is an important requirement for CMoGS in the CIPM approach, due to the dependence of the RDSEFF generation on the use-def relations between function calls and the corresponding declarations. This is due to the fact that knowledge of the location of a function call relative to the declaration determines if the called function is located inside or outside of the calling component, i.e. if the call is internal or external. This information can then be used to determine the presence, or absence, of the function within the RDSEFF of a given component's service.
However, as demonstrated in this section, in the case of dynamic programming languages, not all references can be resolved in the general case. The following section introduces recovery mechanisms as a means to deal with unresolvable references.

4.2.3. Recovery Mechanisms for Unresolvable References

The implementation of a reference resolution mechanism for CMoGS represents a challenging endeavour, even in the context of statically typed programming languages. The primary challenge encountered in this case, is the resolution of references to dependencies outside of the code base of the parsed project, i.e. external dependencies [6]. Previous CIPM research introduced the notion of *recovery mechanisms* [5, 6] to address these challenges. The purpose of recovery mechanisms is to ensure the validity and accuracy of the CM generation, as required by $REQ2_{CM}$ [6]. To this end, recovery mechanisms introduce *synthetic elements*[1] into the CM, whenever encountering an unresolvable reference. An *unresolvable reference* is defined as a reference for which the resolution process cannot identify any suitable non-synthetic declarations. In contrast, an *unresolved* reference is either unresolvable, or has not yet been processed by the reference resolution mechanism. References that are resolved to a synthetic element through a recovery mechanism are reffered to as *synthetic references*. Three recovery mechanisms were introduced in previous work to address unresolvable references [5, 6].

Trivial Recovery. The trivial recovery mechanism uses a single root model element as a container for all synthetic model elements. When a reference cannot be resolved, a synthetic model element with a name corresponding to the referenced element is created, and stored in this container, without considering the context of the reference. This synthetic element is then designated as the referenced element. The initial proposal of this mechanism [6] focuses on its applications for JaMoPP, specifically to the resolution of external dependencies, including the extraction of types from various import declarations. For instance, in the case of an unresolvable method call of type `void` in a Java class, a synthetic method is created and placed inside the container class designated as `SyntheticClass` [6].

[1] In contrast to synthetic rules, which refer to Xtext grammar rules specifying synthetic meta-model elements (cf. Section 2.6.2), synthetic elements in the context of recovery mechanisms refer to CM elements (i.e., elements in instances of the meta-model) without a corresponding syntactical construct in the original code.

Simple & Advanced Recovery. The simple and advanced recovery mechanisms extend trivial recovery by considering the context of references in synthetic element creation [5]. The main idea is to generate synthetic entities, such as classes, and enrich them with information about their properties, such as methods and fields. When a reference to an entity or one of its properties cannot be resolved, a synthetic model element is created for that entity. This synthetic element is assigned a valid, arbitrary name, and its accessed properties are recorded. While the simple recovery mechanism focuses on immediate context information to enhance efficiency, the goal of the advanced recovery mechanism is to reconstruct the original code as accurately as possible, thereby enabling the serialisation of the complete CM, including synthetic elements. However, in situations where contextual information is limited, the mechanism may only provide a best guess for the reconstruction.

As evidenced by the description of these recovery mechanisms, they were conceptualised in the context of statically typed programming languages. Nevertheless, the core principles can be effectively applied to CM generation in dynamically typed languages as well. For example, in the case of unresolvable references to entities or properties, the trivial recovery mechanism could be used to resolve these references to synthetic elements. This would enhance the validity and accuracy of the resulting CM.
Given that static reference resolution in dynamically typed programming languages is inherently approximate, recovery mechanisms offer a practical solution to address this limitation: while they do not provide complete accuracy, they ensure that the reference resolution process can proceed and produce a valid model. The original Lua CMoGS, for instance, uses trivial recovery for unresolvable function calls [15].

4.2.4. The original Lua Code Model Gernerator and Serialiser

This section explores the extent to which the original Lua CMoGS fulfils the requirements REQ1_{CM}-REQ3_{CM}. As described in Section 2.6.2, this CMoGS is based on Xtext, leveraging the automated generation of generators and serialisers from the specification of an Xtext grammar. As such, the satisfaction of the requirements REQ1_{CM} and REQ2_{CM} are closely related, since both the Ecore meta-model as well as the generator and serialiser for the CMoGS are automatically generated by Xtext based on the specification of an Xtext grammar for Lua.

REQ1_{CM} & REQ2_{CM}. Burgey [15] acknowledge that the original CMoGS does not support the complete Lua language, describing this limitation as "Incomplete Support for Lua Syntax Sugar" [15, p. 30], emphasising the complexity and time-consuming nature of implementing a grammar that captures the full syntax of Lua. They illustrate this limitation on the example of syntactic sugar for function calls (cf. Listing 2.3 in Section 2.7), which is not supported by the original CMoGS. A closer analysis revealed further limitations of the original CMoGS with respect to REQ1_{CM} and REQ2_{CM}, which are discussed in the following, on the examples of table accesses and import statements.
Table Accesses. As noted in Section 2.7, tables, as Lua's sole structured data type, constitute an integral part of any Lua program that requires the use of data structures. Furthermore, given that all values in Lua are first-class, tables and table accesses can appear in various

contexts, such as on the left-hand-side (lhs) or right-hand-side (rhs) of an assignment, as arguments to function calls, or return values in function declarations. The original Lua CMoGS is limited to the syntactic sugar syntax Lua provides for table accesses on the lhs, such as `t.x = 1`.
Import statements. The Lua syntax permits the definition of import statements through the use of a call to the function `require`, which receives a string specifying the uri of the import. This parameter can be supplied either in round brackets, or directly as a string, omitting the brackets. Furthermore, it is possible to access table fields of tables returned by imports by assigning them to a variable. Listing 4.5 provides examples for the different import statements.

```
1 -- passing import uri in brackets
2 require("path.to")
3 -- passing import uri without brackets
4 require"path.to"
5 -- field access on an import
6 local external_service = require("path.to").service
```

Listing 4.5: Examples of different import statements in Lua.

The original Lua CMoGS supports the first variant (line 2), but does not support the other two (lines 4 and 6).

REQ3$_{CM}$. Regarding the reference resolution mechanism of the original Lua CMoGS, Burgey [15] notes limitations related to the dynamically typed nature of Lua, stating that it "did not permit us to easily add type scoping to the reference resolution." [15, p. 30]. For instance, the reference resolution uses the notion of a `QualifiedName` as an identifier in the context of *Table Accesses Reference Resolution*, i.e. the resolution of accesses to table fields. This means, that nested table accesses are considered as a single identifier. Consequently, references such as `b.x` in line 4 of Listing 4.6 cannot be resolved, since the referencing environment is unsuccessfully searched for the identifier "`b.x`".

```
1 a = {}    -- declaration of a as a table
2 a.x_1 = 1  -- declaration of x as a field of a
3 b = a     -- declaration of b, b points to a
4 x_2 = b.x_3  -- access of b.x_3 (referencing 1 through x_1)
```

Listing 4.6: Example for limitation arising from using qualified names for resolving table accesses.

These limitations of the original Lua CMoGS underscore the need for an extension to ensure the satisfaction of REQ1$_{CM}$-REQ3$_{CM}$.

4.2.5. Extension of the Lua Code Model Generator and Serialiser

To address the limitations of the original Lua CMoGS outlined in the previous section, this thesis proposes an extension designed to satisfy requirements $REQ1_{CM}$-$REQ3_{CM}$. This section provides insights into the extension in relation to these requirements.

4.2.5.1. Extension towards $\mathbf{REQ1}_{CM}$ & $\mathbf{REQ2}_{CM}$

The first two requirements for CMoGS, while challenging in implementation, as noted by Burgey [15], are conceptually straightforward when utilising frameworks such as Xtext. By leveraging the mechanisms provided by Xtext – that is the generation of an Ecore meta-model as well as a generator and serialiser from the specification of an Xtext grammar – the primary task becomes defining an Xtext grammar for Lua in such a way that the resulting meta-model, generator, and serialiser conform to the requirements $REQ1_{CM}$ and $REQ2_{CM}$. The extension corresponding to these requirements involves the adaptation of the existing Xtext grammar specification to more accurately represent the Lua reference specification [71]. To address the issues regarding table access resolution caused by the use of qualified names as identifiers, the extension restricts identifiers in the CM to those that conform to the Lua reference specification:

> "Names (also called identifiers) in Lua can be any string of letters, digits, and underscores, not beginning with a digit." [71]

The extension includes support for the Lua language up to version 5.2. Figure 4.2 contains an excerpt of the extended meta-model.

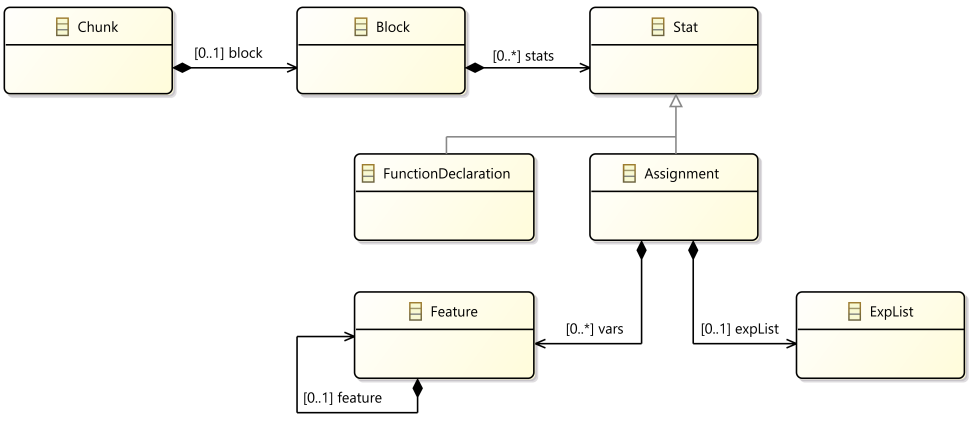


Figure 4.2.: Excerpt of the extended Lua Ecore meta-model.

In the Lua programming language, a `Chunk` is defined as a "unit of compilation" [71], and contains a `Block`. `Blocks`, in turn, consist of a number of `Statements`, represented by the `Block` and `Stat` elements in the meta-model, respectively. The reference grammar defines various types of `Statements`, which are all represented by meta-model elements. Two of these are depicted in the excerpt as examples: the `FunctionDeclaration` and `Assignment`

elements. Given that in Lua, each `Assignment` allows multiple assignments, i.e. may define multiple reference-declaration relationships, the `Assignment` meta-model element contains a list of `Features` as well as an expression list (`ExpList`). Features represent a significant addition of the extended Lua CMoGS, and will be addressed in greater detail later in this section.

4.2.5.2. Extension towards **REQ3**$_{CM}$.

As described in Section 2.6.2, the original Lua CMoGS leverages the reference resolution mechanisms provided by Xtext, which initially represent references as proxy objects during AST parsing. These proxy objects are resolved to the actual referenced elements through Scope Providers, which are queried for suitable candidates from the referencing environment of the referencing element.
The *referencing element* is the element containing the reference. This reference is either a proxy object, or a pointer to the referenced model element if the reference was already resolved. Any referencing element contains an identifier, which is used to detect suitable candidates to reference, also denoted as *referenceables*.
Algorithm 1 presents pseudocode outlining the fundamental reference resolution process in the original CMoGS, illustrating the traversal of Lua `Blocks` and the corresponding scopes.

Algorithm 1 Lua Scope Traversal

Input: A referencing element, i.e. the CM element containing the reference (ELE)
Output: List of candidates from the given ELE's referencing environment (best match first).

```
1: BLOCK ← getContainingBlock(ELE)
2: while BLOCK ≠ NULL do
3:     CAND ← getBlockScope(ELE, BLOCK)
4:     if CAND ≠ ∅ then
5:         return CAND
6:     end if
7:     BLOCK ← getContainingBlock(BLOCK)
8: end while
9: return getGlobalScope(ELE)          ▷ Returns candidates from import statements
```

It involves the traversal of all Lua `Blocks` that are ancestors of the referencing element, starting from its parent `Block`. For each `Block`, `getBlockScope` (line 3) detects all suitable candidates based on a comparison of the corresponding identifiers. If any candidates are found, the results are returned; if not, the search continues in the `Block`'s parent `Block`. In the event that no `Blocks` in a given Lua `Chunk` contain any candidates, the `Chunks` specified by any import statements are considered. The referencing element is then linked to the first candidate in the returned List.

The present thesis proposes and implements a series of extensions to the original reference resolution mechanism, which are discussed in greater detail in the remainder of this section and can be summarised as follows:

1. **Enhancement of resolution correctness**.
2. **Extension of resolvable references**.
   a) Add Feature Path Resolution to address the issues relating to *Table Access Resolution* (cf. Section 4.2.4).

      Feature Path Resolution refers to the resolution of Features and Feature Paths, which are introduced in more details shortly, and encompasses b) and c).

   b) Add support for Function Call Resolution.

      Function Call Resolution refers to the resolution of accesses to fields of tables returned by functions, and entails an evaluation of the function's return value.

   c) Add support for Dynamic Field Access Resolution.

      Dynamic Field Access Resolution refers to the resolution of dynamic field accesses, and entails an evaluation of index expressions (e.g. `table[i]`).

   d) Add support for reconstruction of Lua standard libraries through *implicit libraries*.
3. **Adaptation of trivial recovery** to the extended Lua meta-model and reference resolution mechanism (covered in Section 4.2.6).

Enhancement of resolution correctness. The fundamental functionality of Algorithm 1 is largely adopted by the extension. The adjustments made here focus on corrections relating to the visibility and shadowing of binding occurrences. For example, the original CMoGS assigns the first binding occurrence in a given `Block` to any applied occurrence in the same `Block`, whereas the extension assigns the last binding occurrence preceding the applied occurrence. This ensures that in the event of subsequent binding occurrences of the same identifier the correct referenceable is assigned to the referencing element [71], as illustrated by the example in Listing 4.7.

```
1 ...
2 f1 = internal_service    -- binding assigned by original CMoGS
3 f2 = external_service    -- binding assigned by extended CMoGS
4 f3()                     -- reference to f2
5 ...
```

Listing 4.7: Example of a binding occurrence (f_1) being shadowed by subsequent binding (f_2).

Extension of resolvable references. The main contribution of the extension proposed here is the integration of support for table access resolution, function call resolution and dynamic field resolution. To facilitate this support, the extended meta-model incorporates the `Feature` element, which serves to represent the `var`, `prefixexp`, and `functioncall` rules

from the reference grammar [71] depicted in Listing 4.8. The terminal `Name` in this listing refers to Lua identifiers as specified previously (see Section 4.2.5.1).

```
1 var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
2 prefixexp ::= var | functioncall | '(' exp ')'
3 functioncall ::= prefixexp args | prefixexp ':' Name args
```

Listing 4.8: Excerpt from the Lua reference grammar [71] given in eBNF, containing the rules represented by the Feature meta-model element.

These grammar rules define how accesses to table fields can be chained in Lua. Chained accesses are represented in the extended reference resolution mechanism by introducing the notion of Feature Paths.
A *Feature Path* consists of a sequence of Features, and can be used to encapsulate sequences of accesses on properties of entities, which in the case of the Lua programming languages are always tables. An example of a Lua Feature Path is given in Listing 4.9.

```
root_table  .returns_table  ()  ["field_access"]
   root          access      call      access
```

Listing 4.9: Example of a Feature Path in the Lua Programming Language, annotated with Feature types.

The first Feature in a Feature Path is referred to as the *root Feature*, and the last as the *leaf Feature*. The extended reference resolution mechanism distinguishes between different Feature types:

1. The aforementioned *root Feature* references either a table, or a function declaration that returns a table, represented by an identifier.
2. An *access Feature* denotes access to an entity through the properties of a parent entity. In Lua, it is represented either by an expression enclosed in square brackets (`["field_access"]` in the example), or the syntactic sugar provided for such an access (`.returns_table` in the example).
3. A *function call Feature* denotes a call to the function specified by the previous Feature, and is represented by a parameter list in round brackets, a table constructor, or a string in Lua [71].

In the context of Lua reference resolution, the leaf Feature of a Feature Path can denote either a binding or an applied occurrence. When the Feature Path appears on the lhs of a Lua `Assignment`, it denotes a binding occurrence (declaration) of the leaf Feature.
Given that in Lua, all values are considered as first-class values, possible locations for applied occurrences (references) of leaf features include function arguments, return values, or the rhs of declarations. In addition, all non-leaf Features of a Feature Path denote references. A Feature Path consisting of a single Feature can be considered as a simple identifier.

Feature Paths are utilised by the extended reference resolution mechanism to implement Feature Path Resolution as illustrated by Algorithm 2. Note that this algorithm is embedded into the fundamental `Block` resolution process as part of `getBlockScope`, as discussed for Algorithm 1.

Algorithm 2 Feature Path Resolution

Input: A referencing Feature (ELE), Block provided by Algorithm 1 (BLOCK)
Output: Set of candidates from the ELE's referencing environment

1: $FP \leftarrow buildFeaturePath(ELE)$ ▷ Feature Path from root to ELE
2: $FPC \leftarrow buildFeaturePathCandidates(BLOCK, FP)$
3: **for** $FEATURE \in FP$ **do**
4: $\quad FPC \leftarrow filter(FPC, FEATURE)$
5: $\quad FPC \leftarrow extend(FPC, FEATURE)$
6: **end for**
7: $FPC \leftarrow getCompletelyMatched(FPC)$
8: **return** FPC

Algorithm 2 begins with the construction of a Feature Path `FP` (line 1) from the provided referencing Feature, which forms the leaf of the constructed `FP`. Then, an initial set of Feature Path Candidates `FPC` is identified by `buildFeaturePathCandidates` (line 2). A Feature Path Candidate consists of a Feature Path and a pointer, pointing at the next Feature to consider in the filtering process (line 5).
The initial candidates consist of all Feature Paths on the lhs of declarations in the scope delimited by the given `Block`. The pointers of the initial candidates are set to their root Features. `FPC` is then extended and filtered by considering each Feature contained in `FP` (lines 3-5).
The filtering process (line 4) involves a comparison of the currently considered Feature `FEATURE` of `FP` with the corresponding Feature from each candidate, specified by the candidate's pointer. This comparison is performed for every Feature in `FP`, starting from the root Feature, and traversing `FP` towards the leaf Feature `ELE`. The basis of this comparison are comparable identifiers generated for each Feature. In the case of the root Feature, this identifier corresponds to the Lua identifier, whereas for access Features, the Feature identifier is evaluated through Dynamic Field Access Resolution. This results in string representations for all identifiers, which are compared for equality to identify matches.
In the event of a match between identifiers, the candidate's pointer is moved to the next Feature, otherwise the candidate is removed from `FPC`. Whenever a pointer of a candidate would surpass the leaf Feature after a match, the candidate is marked as *completely matched*. Completely matched candidates are discarded by any subsequent filtering step, as they represent a Feature Path that contains less Features than `FP`.
After the filtering, the extension process (line 5) extends the current candidate set `FPC` with new candidates, based on existing completely matched candidates, and the currently considered `FEATURE`.
For completely matched candidates, the extension process considers the value assigned to the leaf, that is, the value on the rhs of the declaration containing the candidate. This value is evaluated, with additional candidates being constructed, for instance, if the value itself is

a referencing element. This entails Feature Path Resolution on the corresponding value.
Further additional candidates are constructed if the currently considered FEATURE is a non-leaf function call Feature, by evaluating the table returned from the corresponding function declaration through Function Call Resolution. This process is comparable to applying Algorithm 2 to the returned Feature Path's leaf, except that the returned candidates are not filtered by completely matched candidates (line 7). Instead, the pointers of these candidates are set to the Feature subsequent to the returned leaf Feature.
Once the filtering and extension processes have been applied to FPC for each Feature in FP, the remaining non-completely matched candidates in FPC are discarded (line 7), and the resulting set is returned. This set contains all completely matched candidates such that the first candidate in the set represents the best match for the referencing Feature (ELE).
The following example illustrates the processes involved in Feature Path Resolution.

```
1 getA1 = function()   -- declaration of getA
2     t1 = {}
3     t2.x1 = 10
4     return t3
5 end
6
7 a1 = getA2() -- declaration referencing the table returned by getA
8 b1 = a2       -- declaration referencing the table returned by getA
9 x2 = b2.x3    -- access to x3, referencing x1
```

Listing 4.10: Example of Lua Feature Path Resolution to resolve x_3.

Consider the Lua code in Listing 4.10, which combines components from previous examples provided in this thesis. Assume that the Feature Path Resolution process is applied to the resolution of x_3 (line 8), and that no elements outside of the example are part of its referencing environment. The Feature Path FP constructed from this Feature is denoted as $[b_2, x_3]$. The initial set of Feature Path Candidates is given by $FPC = \{[get\_A'_1]_C, [a'_1]_C, [b'_1]_C\}$. A fully matched candidate is denoted by "*" and the pointer position is denoted by "'".
The first iteration considers $FEATURE = b_2$. The candidates are filtered based on b_2, resulting in the filtered candidate set $\{[b'_1]^*_C\}$, containing a single, completely matched candidate.
The extension process then extends FPC to $\{[b'_1]^*_C, [t'_1]^*_C, [t_2, x'_1]\}$, where the first two elements are completely matched and the pointer of the third points at x_1. This process involves the resolution of a_2 to a_1, as well as the evaluation of the table resulting from the call to $getA_1$.
The next iteration considers $FEATURE = x_3$, resulting in the candidate set $FPC = \{[t_2, x'_1]^*_C\}$. The subsequent extension does not provide any new candidates, and $FPC = \{[t_2, x'_1]^*_C\}$ is returned as the result. This leads to the applied occurrence x_3 being linked to the binding occurrence x_1 by the overarching reference resolution mechanism, concluding the example for Feature Path Resolution.

In addition to Feature Path Resolution, the extension of resolvable references encompasses the support for reconstruction of Lua's standard libraries. The extended Lua CMoGS

automatically incorporates files containing information about Lua's standard libraries as *implicit libraries* into the CM, thereby enabling the resolution of references to library functions. However, this mechanism is confronted with challenges arising from the Lua standard library functions being written in C, which are discussed in greater detail in Section 4.2.7.

It should be noted that the all references resolved by the reference resolution mechanism, as well as the container utilised for trivial recovery, are disregarded by the Serialiser provided by the extended CMoGS. This ensures that serialised models conform to the Lua language specification.

4.2.6. Adaptation of Trivial Recovery, Reference Types & Causes for Synthetic References

The changes introduced to the Lua CM by the extended CMoGS necessitate adaptations to the recovery mechanism employed by the original. This section provides details on these adaptations as performed in this thesis.

The extended trivial recovery mechanism resolves unresolvable Feature Paths to synthetic Feature elements. To this end, a synthetic `Chunk` is created as a container for all synthetic elements. This `Chunk` consists of a single `Block`. For any unresolvable Feature Path, a synthetic Assignment containing a synthetic Feature on its lhs is inserted into this `Block`. This synthetic Feature is assigned an identifier corresponding to the generated identifier for the unresolvable Feature (i.e., the leaf Feature of the unresolvable Feature Path). The leaf Feature of the unresolvable Feature Path can then be linked to the corresponding synthetic Feature, resulting in a synthetic reference.

In order to assess the extent to which references are resolved non-synthetically or synthetically, this section introduces a categorisation of *reference types*. As described in section 4.2.3, non-synthetic references are those referencing the actual referenced element in the CM, i.e. the CM element corresponding to the referenced entity in the source code, whereas synthetic references are those resolved through the recovery mechanism. In addition, synthetic references are further categorised according to the *cause* for their synthetic resolution.

The references resolved by the extended Lua CMoGS can be categorised into types based on what they reference (e.g., a Lua `function` or `table`). As discussed previously, the resolution of function calls, i.e. references to Lua `functions`, is particularly important for the CIPM approach. The following reference types are distinguished in the extended Lua CMoGS:

1. The **Root type (R)** refers to the resolution of root Features that are not followed by a function call Feature, e.g., `root_table` in Listing 4.9.
2. The **Table Access type (TA)** refers to the resolution of access Features that are not followed by a function call Feature, e.g., `["field_access"]` in Listing 4.9.
3. The **Function Call type (FC)** refers to to the resolution of any Feature followed by a function call Feature, e.g. `returns_table()` in Listing 4.9.

The main focus of the Lua CMoGS reference resolution mechanism is to resolve as many Function Call (FC) references as possible without relying on trivial recovery. This is crucial because FC references determine which RDSEFFs need to be reconstructed.
Given the concept of Feature Paths introduced earlier, Root (R) and Table Access (TA) references also play a significant role in achieving non-synthetic FC resolution. If an R or TA reference within a Feature Path is resolved synthetically (i.e., through the recovery mechanism), all subsequent references in that path will also be resolved synthetically, as the corresponding access cannot be correctly resolved.
From the perspective of Feature Path Resolution (Algorithm 2), if no Feature Path Candidates can be found for an earlier Feature, then no candidates can be found for any subsequent Features in the same Feature Path. For example, in 4.9, if the R reference to `root_table` in `root_table.returns_table()` is resolved to a synthetic element, then the FC reference to `returns_table()` must also be resolved synthetically.

Different *causes* can lead to the resolution of a reference through the recovery mechanism. Identifying the underlying cause of a synthetic reference is valuable, as it highlights the strengths and limitations of the Lua CMoGS. Table 4.3 summarises the key causes relevant to the reference resolution process in the extended Lua CMoGS.

Cause	Short Description
Function Resolution (C_{FR})	Function Call Resolution failed.
Dynamic Field Access Resolution (C_{DA})	Dynamic Field access Resolution failed.
External Library Resolution (C_{Ext})	Resolution of external library failed.
Implicit Import Resolution (C_{SL})	Resolution of standard library member failed.
Argument Access Resolution (C_{Arg})	Resolution of argument access failed.
Unidentified (C_U)	Subsumes all unidentified causes.

Table 4.3.: Short descriptions of causes for synthetic reference resolution (resolution through trivial recovery) in the extended Lua CMoGS.

- **Function Resolution** (C_{FR}): Occurs when a Feature corresponding to an access on a function call return value cannot be resolved (cf. Function Call Resolution in previous Section).
- **Dynamic Access Resolution** (C_{DA}): Occurs when a table access index expression cannot be resolved, affecting subsequent Features (cf. Dynamic Field Access Resolution in previous Section).
- **External Library Resolution** (C_{Ext}): Indicates that a reference is unresolvable due to an access to an external library, typically caused by missing source code required for CM generation.
- **Implicit Import Resolution** (C_{SL}): Refers to synthetic references resulting from unsuccessful resolutions to the Lua standard library.

- **Argument Access** (C_{Arg}): Arises when an access to a field of an argument cannot be resolved, for example the access `arg.field` within the body of a function that takes `arg` as a parameter.
- **Unidentified** ($_U$): Finally, *Unidentified* (C_U) subsumes all causes that are not detectable through cause detection, i.e. unidentifiable causes. For instance, if a declaration contains a dynamic access that cannot be resolved, as in

  ```
  table[<some expression that resolves to "member">] = <some entity>,
  ```

  a subsequent applied occurrence of `"member"` in `table.member` will be resolved synthetically, but the cause cannot be determined with certainty given that the expression could not be resolved.

These causes can be identified by processing each referencing element in the generated Lua CM that references a synthetic element. In CMs generated through the extended Lua CMoGS, these are always Feature elements. Algorithm 3 outlines the cause detection for such a Feature (*SF*). The first step is to determine the first synthetically referencing Feature (*FSF*) in *SF*'s Feature Path (line 2), given that the first synthetically resolved Feature in any Feature Path results in the synthetic resolution of all subsequent Features in that path. Then, *FSF*'s reference chain is traversed by $inferCause(FSF)$ (line 3).
The cause detection assumes that root Features are resolvable unless they reference binding occurrences provided in external libraries, i.e. the inferred cause is C_{Ext} whenever *FSF* is a root Feature. If *FSF* is not a root Feature, $inferCause(FSF)$ infers the cause by analysing the Feature preceding *FSF*. The preceding Feature is not resolved synthetically, as ensured in line 2. Thus, the cause detection can leverage information about the referenced element of the preceding Feature to infer the cause for *FSF*, and, by extension, *SF*. For instance, if the referenced element is contained in the Lua standard library, the inferred cause is C_{SL}, if it is an argument, the inferred cause is C_{Arg}, and if it is a function declaration, the inferred cause is C_{FR}.

Algorithm 3 Cause Detection

Input: Feature in the CM that references a synthetic element (SF)
Output: Cause for the synthetic reference
1: $FP \leftarrow buildFeaturePath(SF)$
2: $FSF \leftarrow getFirstSyntheticInFeaturePath(FP)$
3: $Cause \leftarrow inferCause(FSF)$
4: **return** *Cause*

Limitations of the extended Lua CMoGS. As discussed in the previous section, the extension to the Lua CMoGS attempts to avoid synthetic reference resolution caused by C_{FR}, C_{DA}, and C_{SL}, through the introduction of Feature Path Resolution and the reconstruction of Lua's standard libraries. However, synthetic references caused by references to external libraries (C_{Ext}) and argument accesses (C_{Arg}) are not considered in the extension. Additional limitations relating to the prototypical implementation of the extended Lua CMoGS are discussed in the next section.

4.2.7. Implementation Details: Limitations & Boundaries

This section offers a synopsis of notable details concerning the prototypical implementation of the extended CMoGS performed as part of this thesis. The focus here is on the limitations regarding implicit imports and the boundaries of the evaluation of expressions for Table Access Resolution. Details regarding the integration of the extended Lua CMoGS into the broader prototypical implementation of the CIPM approach are provided in the subsequent section.

The prototypical implementation of the extended Lua CMoGS provides support for implicit imports as discussed previously. However, their reconstruction depends on the availability of Lua code specifying their implementation. Given that Lua's standard libraries are implemented in C, implementation details are not available as Lua code. Consequently, the prototypical implementation relies on generating implicit imports from Lua files extracted from the sumneko Lua language server [58], which primarily contain information relevant to providing developers with assistance within Integrated Development Environments (IDEs). As this information is predominantly provided through Lua comments, which are ignored by the CMoGS generator implementation, and annotations (e.g., for types and function parameters), this information is of limited use for the reference resolution mechanism. As a result, the reference resolution frequently relies on trivial recovery for references to functionalities provided by the Lua standard library (cf. C_{SL} in previous Section). In addition, implicit imports are not serialised by the provided implementation, given that they are implicitly available in any Lua program through the Lua standard library, and their limited representation in the CM results in limited usefulness for CIPM's adaptive instrumentation.

Moreover, the prototypical implementation provides mechanisms for the evaluation of expressions in the context of Table Access Resolution, as discussed in the previous section. As outlined in Section 4.2.2, the static evaluation of such expressions is inherently constrained. Consequently, any implementation of Table Access Resolution must adhere to certain boundaries regarding expression evaluation. In the case of the prototypical implementation of the extended CMoGS, these boundaries are defined as follows:

- Evaluation of string and number literals, e.g. `table["access"]` and `table[0]`.
- Evaluation of assignments to string and number literals, e.g. `table[access]` for `access = "access"`.

The evaluation of index expressions excludes all other expressions. This includes function return values, binary expressions (e.g. additions), and grouped expressions (i.e. combinations of expressions in round brackets).

4.2.8. Integration into the CI-based Update of Software Models

As described in Section 2.5.3 CIPM's CI-based update of software models involves generating a CM in response to an incoming commit and detecting changes between the newly generated CM and the existing CM in the Vitruvius V-SUM. The detected changes are applied to the V-SUM CM, triggering updates to the PCM `Repository` Model and CIPM's IM based on the specified CPRs.

Since the extended CMoGS includes extensions to the Lua meta-model, integrating it into CIPM requires technical adaptations to the matching strategy originally used for Lua CM change detection and the exiting CPRs [15]. However, CPRs are not only language-dependent but also specific to the technology and domain of the project to which CIPM is applied. As the original Lua CPRs were designed for specific proprietary evaluation cases, they cannot be directly reused, necessitating their in-depth extension for new applications.

To explore the usability of the extended Lua CMoGS within CIPM's CI-based update of software models while maintaining a manageable scope, this thesis focuses on adapting the matching strategy, but only includes a limited adaptation of the CPRs:

- Technical adaptations to the existing prototypical implementation to enable the execution of the CI-based update of software models using the extended Lua CMoGS.
- The introduction of a new component detection strategy, the `FileDetectionStrategy`, for component detection. This detection strategy allows to specify a mapping from project files to their corresponding components, which is applied to generate components when updating the PCM based on the Lua CM.

These extensions facilitate updates to the `Repository` Model of the PCM with regard to the generation of Components, as well as the corresponding `OperationInterfaces`. The latter are the PCM meta-model representation of the interfaces contained in the `Repository` Model. In the extension, `OperationInterfaces` are identified by any function call in one component referencing a function declaration in a different component.
While these extensions do not include the reconstruction of RDSEFFs, a potential approach for a more complete generalisation of the CPRs building upon the `FileDetectionStrategy` is outlined as future work (Chapter 7).

5. Evaluation

This chapter presents the evaluation of the approaches introduced in this thesis. Section 5.1 defines the metrics used in the evaluation, followed by a description of the GQM-plan designed for this purpose in Section 5.2. Sections 5.3 and 5.4 detail the conducted experiments and discuss the corresponding results. Finally, Section 5.5 provides a summary of the evaluation.

5.1. Metrics

This section introduces metrics used in the evaluation performed in this thesis.

5.1.1. Generation-Serialisation Similarity

The *Generation-Serialisation Similarity* metric measures the correctness of the generation and serialisation mechanisms provided by a CMoGS. This metric was referred to as "Textual Differences" by Burgey [15, p. 40], and compares the *original* code – used as input for the CMoGS – with the *serialised* code resulting from generating a CM from the original code and then serialising that CM.
Since constructs such as whitespace and comments can be ignored by the CMoGS without affecting the satisfaction of the requirements REQ1$_{CM}$-REQ3$_{CM}$, these elements may introduce expected discrepancies between the original and serialised code. To account for this, both the original and serialised code undergo a filtering process, that removes all whitespace and comments. After filtering, the original and serialised code are compared character by character. This combined filtering and comparison process is denoted by $\sim$.
Thus, the Generation-Serialisation Similarity metric is defined as

$$Sim(O, CMoGS) = O \sim CM^{-1}(CM(O)) \quad \in \{0, 1\},$$

where:

- O refers to the original code,
- $CMoGS$ represents the CMoGS under consideration, and
- $CM^{-1}(CM(O))$ denotes the serialisation (CM^{-1}) of the CM generated from the original code ($CM(O)$).

A similarity score of $Sim(O, CMoGS) = 0$ signifies issues with the generation and serialisation mechanism provided by the CMoGS, meaning the original and serialised codes are *non-similar*. Conversely, $Sim(O, CMoGS) = 1$ indicates the expected functionality, with the original and serialised codes being *similar*.
However, this metric only verifies the validity and accuracy of the generation and serialisation mechanisms provided by the CMoGS as required by $REQ2_{CM}$; it does not guarantee a meaningful underlying language meta-model. For instance, an Xtext grammar consisting of a single rule that permits any sequence of characters would always produce similar original and serialised code. Therefore, any application of this metric must be complemented by an inspection of the corresponding language meta-model to ensure its meaningfulness.

5.1.2. The Jaccard Index

The *Jaccard Index* is a measure of similarity between two sets. It is defined as the ratio of the size of the intersection of the sets to the size of their union, and therefore also known as Intersection over Union (IoU) (e.g. [13]). For two sets A and B, the Jaccard Index $JI(A, B)$ is given by

$$JI(A, B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

[13]. The value of the Jaccard coefficient ranges from 0 to 1, where 0 indicates no overlap between the sets, and 1 indicates identical sets.
In the context of comparing two CIPM models M_1 and M_2, the Jaccard Index has been used to evaluate the similarity by comparing their respective sets of elements [67, 4, 15]. By defining a matching function that determines if two elements $m_1 \in M_1$ and $m_2 \in M_2$ match, the number of matching elements can be divided by the combined number of matching and non-matching elements to compare the two sets.

5.2. GQM Plan

The **G**oal-**Q**uestion-**M**etric-Plan (GQM-Plan) [83] is a top-down approach to define evaluation procedures. It begins by establishing goals for the evaluation, followed by formulating questions related to each goal to determine its achievement. Finally, measurable metrics are defined to provide quantifiable answers to these questions. This section presents the GQM-Plan for this thesis, building on the methodological approach presented in chapter 3.

G1 *The approach to Semi-Automated CIPM Evaluation Case Selection reduces the effort of manual inspection ($\boldsymbol{A}_{Manual}$) and supports the selection of suitable CIPM evaluation cases.*

G1.1 *The Automated Filtering Process effectively reduces the number of evaluation case candidates with a viable set remaining for $\boldsymbol{A}_{Manual}$.*

This goal focuses on evaluating the Automated Filtering Process as part of the approach to Semi-Automated CIPM Evaluation Case Selection. Since the overarching aim of this approach is to identify suitable evaluation cases for assessing the CIPM approach, the Automated Filtering Process must yield a non-empty set of filtered candidates. In other words, evaluation cases should still be identifiable within the initial set of software projects after the Automated Filtering Process based on REQ_{AFP} and a suitable configuration of RADs and filters. At the same time, the process should effectively reduce the size of this set to minimize the manual effort required by $\mathbf{A}_{\text{Manual}}$.

Q1.1.1 To what extent are software projects suitable for use as CIPM evaluation cases based on the Automated Filtering Process and the set of requirements REQ_{AFP}?

M1.1.1a Satisfaction rate per requirement assessed by the Automated Filtering Process on a set of initial candidates.

The *satisfaction rate* for a requirement is defined as the ratio of candidates from a given set of candidates that satisfy that requirement, relative to the total number of processed candidates. The satisfaction rate is expressed as a percentage.

M1.1.1b Ratio of essential candidates identified by the Automated Filtering Process relative to the total number of processed candidates.

Essential candidates are those that satisfy all *essential* requirements assessed by the Automated Filtering Process, that is REQ1_{EC}, REQ2_{EC}, and REQ4_{EC}. Given that filtered candidates need to satisfy all requirements, filtered candidates are always represent a subset of the essential candidates, i.e.

$$\forall \text{ initial candidate sets} : \{\text{filtered candidates}\} \subseteq \{\text{essential candidates}\}$$

Q1.1.2 To what extent does the Automated Filtering Process reduce the number of project candidates requiring manual inspection ($\mathbf{A}_{\text{Manual}}$)?

M1.1.2 Ratio of the number of filtered candidates returned by the *Automated Filtering Process* for a set of initial candidates, relative to the total number of processed candidates. The ratio is expressed as a percentage.

Q1.1.3 How does the composition of the initial candidate set influence the execution time of the Automated Filtering Process?

M1.1.3 Execution time of the Automated Filtering Process for different sets of initial candidates.

G1.2 *Evaluation cases suitable for CIPM evaluation can be selected through the Approach to Semi-Automated CIPM Evaluation Case Selection.*

Q1.2.1 Can evaluation cases for CIPM evaluation be selected using the semi-automated approach?

M1.2.1a Satisfaction (yes/no): Existence of Lua candidates remaining for the evaluation of **G2** after manual inspection ($\mathbf{A}_{\text{Manual}}$) of selected projects resulting from applying the Automated Filtering Process to a set of Lua projects as initial candidates.

M1.2.1b Satisfaction (yes/no): Existence of Lua candidates remaining for the evaluation of **G3** after manual inspection ($\mathbf{A}_{\text{Manual}}$) of selected projects resulting from applying the Automated Filtering Process to a set of Lua projects as initial candidates.

Q1.2.2 Are evaluation cases selected using the Approach to Semi-Automated CIPM Evaluation Case Selection actually suitable for the evaluation of CIPM?

In contrast to the other questions regarding **G1**, this question pertains to the *actual suitability* of the resulting evaluation cases *to the evaluation of CIPM*.

M1.2.2a Satisfaction (yes/no) of **G2** through application of selected Lua projects resulting from *M1.2.1a* to the evaluation this goal.

M1.2.2b Satisfaction (yes/no) of **G3** through application of selected Lua projects resulting from *M1.2.1b* to the evaluation this goal.

G2 *The extended Lua CMoGS fulfils the requirements for CIPM CMoGS ($REQ1_{CM}$-$REQ3_{CM}$).*

G2.1 *The Lua CMoGS defines an Ecore meta-model for the complete Lua programming language ($REQ1_{CM}$).*

Q2.1 Is the complete Lua programming language represented in the meta-model of the Lua CMoGS?

M2.1 Satisfaction (yes/no) of **G2.2**.

The satisfaction of **G2.2** ensures that all Lua language constructs are parsed and serialized by the CMoGS, implying that the meta-model defined by the CMoGS represents the complete Lua language.

G2.2 *The Lua CMoGS provides a generator and serialiser, capable of transformations between Lua CM and Lua code ($REQ2_{CM}$).*

Q2.2.1 Can the Lua CMoGS correctly parse and serialise all Lua language constructs?

M2.2.1a Ratio of passed benchmark tests evaluating the correctness of generation and serialisation. The correctness is evaluated using the Generation-Serialisation Similarity *Sim*. The ratio is expressed as a percentage.

M2.2.1b Generation-Serialisation Similarity of collections of Lua files L, containing all Lua language constructs, and the serialised code. The similarity is measured using $Sim(L, CMoGS)$, as illustrated in Figure 5.1.

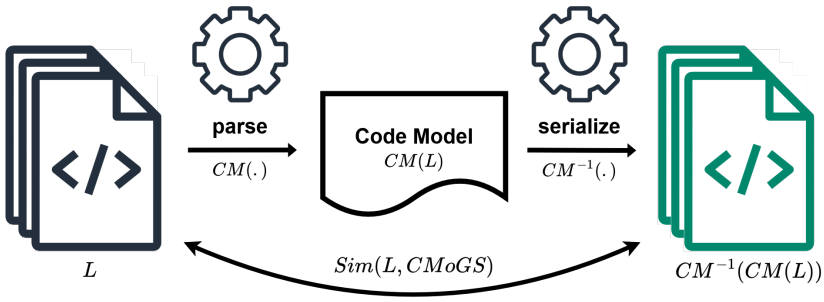


Figure 5.1.: Illustration of the evaluation procedure for M2.2.1.

Q2.2.2 Can the Lua CM parser correctly parse multiple commits of real-world applications?

M2.2.2 Generation-Serialisation Similarity of Lua source code S_i after commit c_i from a real-world evaluation case for a series of commits C_S. The similarity is measured using $Sim(S_i, CMoGS)$.

G2.3 *The Lua CMoGS resolves all references between the model elements ($REQ3_{CM}$).*

Q2.3.1 Are the references resolved without a recovery mechanism resolved correctly?

M2.3.1 Ratio of passed benchmark tests evaluating the correctness of reference resolution in selected cases. The ratio is expressed as a percentage.

The correctness is evaluated by comparing the referenced elements in the CM to the expected referenced elements according to the Lua language semantics.

Q2.3.2 Can recovery mechanisms be applied to all references that cannot be resolved to CM elements of their corresponding source code elements?

M2.3.2 Satisfaction (yes/no) of all unresolvable references being resolved through trivial recovery by the extended Lua CMoGS for real-world evaluation cases.

Q2.3.3 To what extend are references in the CM resolved to their corresponding source code elements?

The metrics for this question are based on the types of references, the causes for synthetic reference resolution, and the corresponding limitations of the extended Lua CMoGS as outlined in Section 4.2.6. Based on these limitations as well as the limitations of the prototypical implementation discussed in Section 4.2.7, synthetic references are divided into two categories, *critical* and *non-critical* synthetic references.

Critical synthetic references are references resolved through trivial recovery with causes the extended Lua CMoGS is expected to address, that is synthetic references caused by Function Call Resolution (C_{FR}) or Dynamic Field access Resolution (C_{DA}). In addition, synthetic references with unidentified causes are regarded as critical, given that no further information about their cause is made available through the Cause Detection Algorithm.

Non-Critical synthetic references subsume references resolved through trivial recovery with any other causes. Given the limitations of the extended CMoGS discussed in the aforementioned sections, the causes C_{Ext}, C_{SL}, and C_{Arg} always result in synthetic references. Therefore, non-critical synthetic references can be regarded as references that are expected to be always resolved synthetically by the extended Lua CMoGS.

M2.3.2a Distribution of total resolved references, critical synthetic references, and non-critical synthetic references for a given evaluation case. When multiple commits are considered for an evaluation case, the values are aggregated across all commits.

M2.3.2b Distribution of resolved references, critical synthetic references, and non-critical synthetic references for each reference type for a given evaluation case. When multiple commits are considered for an evaluation case, the values are aggregated across all commits.

M2.3.2c Distribution of causes for each reference type of a given evaluation case. When multiple commits are considered for an evaluation case, the values are aggregated across all commits.

G3 *The applicability of CIPM's CI-based update of software models is evaluated on real-world, open source Lua projects.*

This goal was previously addressed in [15] for sensor-based Lua applications. However, it requires re-evaluation using the extended Lua CMoGS and the CPRs specific to the case study used in this thesis. To maximize comparability, the evaluation procedure outlined in [15, pp. 40-43] is adopted for this goal.

G3.1 *Changes resulting from a commit are accurately propagated to the V-SUM CM.*

This goal ensures that the change-driven update of the V-SUM CM functions as intended, resulting in a V-SUM CM that represents the repository state of the case study after the specified commit.

Q3.1 Are the changes resulting from a commit accurately propagated to the V-SUM CM?

M3.1 Difference between CM_i, the V-SUM CM after commit c_i, and the result of $CM(S_i)$. The expected results are $JI(CM_i, CM(S_i)) = 1$.

G3.2 *Architectural changes to the CMs are accurately propagated to the PCM and the IM.*

Q3.2.1 Are the components defined by the component detection strategy accurately propagated to the PCM?

M3.2.1 Satisfaction (yes/no) of all components defined by the component detection strategy being represented in the PCM `Repository` model created by the CIPM approach for each commit c_i of a series $\{c_0, ..., c_n\}$.

Given that the component detection strategy defined in this thesis maps files to components (cf. Section 4.2.8), the resulting components in the PCM `Repository` model can be directly compared to the components defined by the component detection strategy. This comparison is performed manually.

Q3.2.2 Are the architectural changes to the CMs accurately propagated to the PCM?

M3.2.2 Difference between PCM `Repository` model created by the CIPM approach after a series of commits $PCM_R(\{c_0, ..., c_n\})$ and the PCM `Repository` model created directly from the last commit $PCM_R(c_n)$. The expected results are $JI(PCM_R(\{c_0, ..., c_n\}), PCM_R(c_n)) = 1$.

5.3. G1: The Approach to Semi-Automated CIPM Evaluation Case Selection

This section presents the experiment conducted to evaluate the achievement of **G1**, the results and their analysis, and a discussion of their implications.

5.3.1. Experiment $\mathbf{E_{EC}}$

The goal of Experiment $\mathbf{E}_{\text{EC}}$ is to evaluate the approach to Semi-Automated CIPM Evaluation Case Selection (**G1**) proposed in this thesis. In particular, this experiment aims to answer **Q1.1.1**, **Q1.1.2**, **Q1.1.3**, and **Q1.2.1** by performing the Process for Semi-Automated CIPM Evaluation Case Selection using the set of Evaluation Case Requirements for CIPM specified in the approach.
To this end, the prototypical implementation of the Automated Filtering Approach is applied to three sets of initial Lua and Java project candidates, IC_{Lua}, IC_{Java}, and IC_{Union}. These sets consist of Lua and Java projects, as specified in Table 5.1. It is anticipated that this experiment will result in a number of case studies that can be utilised in subsequent experiments, thereby providing answers to **Q1.2.2**.

Initial Candidate Set	Specification	Size
IC_{Lua}	*Github-Ranking Top 100 Lua* projects [36].	100
IC_{Java}	*Github-Ranking Top 100 Java* projects [35]	100
IC_{Union}	$IC_{\text{Lua}} \cup IC_{\text{Java}}$	200

Table 5.1.: Sets of initial project candidates for Experiment $\mathbf{E}_{\text{EC}}$.

The selection of Lua and Java projects from their respective Top 100 GitHub rankings ensures that all initial candidates represent popular projects that provide repositories and are accessible via the GitHub API. The set IC_{Lua} represents a reasonable selection of initial candidates concerning goals **G2** and **G3** of this thesis. IC_{Java} is included to examine the generalisability of the Approach to Semi-Automated CIPM Evaluation Case Selection to other programming languages. Since prior research on the CIPM approach has focused on Java projects, using Java for IC_{Java} provides continuity with these studies. Finally, IC_{Union} is used to evaluate the application of the Automated Filtering Approach to an initial candidate set that includes multiple programming languages.

In addition to the selection of initial candidates, the approach to Semi-Automated CIPM Evaluation Case Selection requires the configuration of RADs and filters in accordance with the specific research goals. The RADs and filters used in this experiment correspond to those specified in Section 4.1.4, with the configurations being based on the examples

provided therein. The sets of keywords for the RADs using $OCC(KW)$ are therefore defined as

$$KW_A = \{(\text{"architecture"}, df), (\text{"architectural"}, df)\}$$

for REQ1_{EC} (Arch. Doc.), and

$$KW_T = \{(\text{"test"}, df), (\text{"t"}, d), (\text{".t"}, f))\}$$

for REQ5_{EC} (Perf. Tests). A set of open-source licenses LS_G provided by Github [55] is used for REQ4_{EC} (Open-source), ensuring the comparability to the license ls of a candidate retrieved through the Github API. Lastly, the prototypical implementation requires the specification of the timeouts t_{cloning} and $t_{\text{candidate}}$ for concurrent operations, which are set to

$$\begin{aligned} t_{\text{cloning}} &= 30\text{min.}, \\ t_{\text{candidate}} &= 60\text{min.} \end{aligned}$$

for this experiment.
The two applied configurations CFG_{Lua} and CFG_{Java} are specified in Table 5.2. The only difference between these are programming language-specific configurations of the file extension employed in *LOC* computation, and the main programming language used as filter for *MPL*.

Requirement	RADs		Fiilters	
	CFG_{Lua}	CFG_{Java}	CFG_{Lua}	CFG_{Java}
REQ1_{EC} (Arch. Doc.)	$OCC(KW_A)$	$OCC(KW_A)$	$OCC(KW_A) \geq 1$	$OCC(KW_A) \geq 1$
REQ2_{EC} (Git Repo.)	$\|C\|$	$\|C\|$	$\|C\| \geq 100$	$\|C\| \geq 100$
REQ4_{EC} (Open-Source)	License ls	License ls	$ls \in LS_G$	$ls \in LS_G$
REQ5_{EC} (Perf. Tests)	$OCC(KW_T)$	$OCC(KW_T)$	$OCC(KW_T) \geq 1$	$OCC(KW_T) \geq 1$
REQ6_{EC} (Activity)	AVG_{C/W_8}	AVG_{C/W_8}	$AVG_{C/W_8} \geq 1$	$AVG_{C/W_8} \geq 1$
REQ8_{EC} (Pr. Language)	MPL	MPL	MPL = "Lua"	MPL = "Java"
REQ9_{EC} (Scale)	LOC(`.lua`)	LOC(`.java`)	LOC(`.lua`) ≥ 1000	LOC(`.java`) ≥ 1000

Table 5.2.: Configuration of RADs and filters for the configurations CFG_{Lua} and CFG_{Java} used in experiment $\mathbf{E}_{\text{EC}}$.

Using the sets of initial candidates and configurations, three executions of the Automated Filtering Process are conducted. The first run AFP_{Lua} is executed with CFG_{Lua} on IC_{Lua}, while the second run AFP_{Java} is executed with CFG_{Java} on IC_{Java}. This enables an assessment of the Automated Filtering Process on divers sets of initial candidates and programming languages specific configurations. A third run, AFP_{Union}, is executed with configuration CFG_{Lua} on IC_{Union} to evaluate the behaviour in the event of initial candidates from multiple programming languages.

Additionally, the execution time for each run is measured, and the average LOC over all processed candidates for each run is calculated based on the corresponding RADs, to evaluate the influence of the composition of the initial candidate sets on the execution time of the Automated Filtering Process. All runs are executed on a Windows 10 operating system (version 10.0.19045, Build 19045), with an Intel i5-3570 CPU and 8 GB of RAM.

Run	Initial Candidate Set	Configuration
AFP_{Lua}	IC_{Lua}	CFG_{Lua}
AFP_{Java}	IC_{Java}	CFG_{Java}
AFP_{Union}	IC_{Union}	CFG_{Lua}

Table 5.3.: Automated Filtering Process run configurations for $\mathbf{E}_{\text{EC}}$.

Following the Automated Filtering Process, a manual inspection of a selection of the filtered candidates resulting from AFP_{Lua} is performed to select the final evaluation cases used for the subsequent evaluations of **G2** and **G3**, completing the Process for Semi-Automated CIPM Evaluation Case Selection.

5.3.2. $\mathbf{E_{EC}}$: Results

This section presents the results of experiment $\mathbf{E}_{\text{EC}}$ on the basis of the measurements taken for metrics M1.1.1a-M1.2.1b.

Results for AFP_{Lua}. Figure 5.2 depicts the satisfaction rate for each requirement as assessed by the Automated Filtering Process in AFP_{Lua} (M1.1.1a). Of the initial 100 candidates contained within IC_{Lua}, 86 were processed by the Automated Filtering Process, while 14 candidates were excluded due to errors (cf. Section 4.1.6). The nature of these errors included invalid responses from the Github API, the absence of license information through the Github API, and invalid repositories resulting from file name incompatibilities with the Windows operating system, on which the experiment was executed. REQ1_{EC} (Arch. Doc.) demonstrated the lowest satisfaction rate, with six projects (6.9%) satisfying this requirement. This was followed by REQ6_{EC} (Activity), which was satisfied by 39 projects (44.84%). The remaining requirements were satisfied by over 80% of the initial candidates. Given that set of candidates from IC_{Lua} only contains Lua projects, all candidates satisfy REQ8_{EC} (Pr. Language).

The processed candidates for IC_{Lua} include and identical set of essential and filtered candidates, resulting in 4.65% for both M1.1.1b and M1.1.2, which measure the ratio from essential and filtered candidates relative to the total number of processed candidates, respectively. These results are depicted in Table 5.7 alongside the corresponding results for IC_{Java}.
The four filtered candidates returned by AFP_{Lua} and the corresponding RADs are shown in table 5.4. Note that in the following, all tables depicting candidates and the corresponding RADs are sorted by REQ1_{EC} (Arch. Doc.). Additionally, the RADs for REQ4_{EC} (Open-Source)

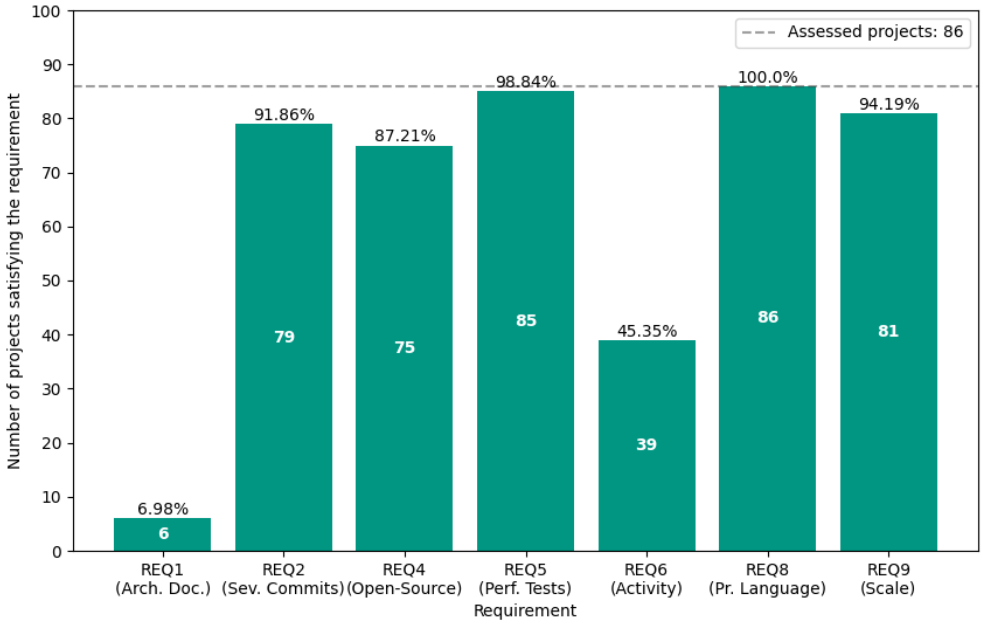


Figure 5.2.: Satisfaction rate for each requirement after the Automated Filtering Process in relation to the total number of initial candidates for AFP_{Lua} (M1.1.1a).

and REQ8_{EC} (Pr. Languages) are excluded from tables showing only filtered candidates to improve conciseness and clarity.
To further illustrate the filtering through RADs, Table 5.5 contains all candidates that satisfied the this requirement. From these six candidates, two were excluded from the filtered candidates. The candidate "leandromoreira/cdn-up-and-running" fails to satisfy REQ2_{EC} (Several Commits) with $72 < 100$ commits, REQ6_{EC} (Activity) with $0 < 1AVG_{C/W_8}$, and $141 < 1000$ LOC. Similarly, "junyanz/CycleGAN" fails to satisfy REQ2_{EC} (Several Commits) and REQ4_{EC} (Open-Source), with 99 commits and "other" returned as the license from the Github API, which is not contained in LS_G.

Project Name	REQ1_{EC} (Arch. Doc.)	REQ2_{EC} (Several Commits)	REQ5_{EC} (Perf. Tests)	REQ6_{EC} (Activity)	REQ9_{EC} (Scale)
ntop/ntopng	5	23966	739	29.375	269583
apache/apisix	4	4366	643	4.25	59892
xmake-io/xmake	1	15385	764	41.875	186557
Saghen/blink.cmp	1	880	14	45.75	10209

Table 5.4.: Filterd candidates and corresponding RADs yielded by the Automated Filtering Process in AFP_{Lua}. RADs for REQ4_{EC} (Open-Source) and REQ8_{EC} (Pr. Language) omitted for clarity.

Project Name	$REQ1_{EC}$ (Arch. Doc.)	$REQ2_{EC}$ (Several Commits)	$REQ4_{EC}$ (Open-Source)	$REQ5_{EC}$ (Perf. Tests)	$REQ6_{EC}$ (Activity)	$REQ8_{EC}$ (Pr. Language)	$REQ9_{EC}$ (Scale)
ntop/ntopng	5	23966	gpl-3.0	739	29.375	Lua	269583
apache/apisix	4	4366	apache-2.0	643	4.25	Lua	59892
leandromoreira/cdn-up-and-running	3	72	bsd-3-clause	5	0.0	Lua	141
junyanz/CycleGAN	1	99	other	10	0.0	Lua	3594
xmake-io/xmake	1	15385	apache-2.0	764	41.875	Lua	186557
Saghen/blink.cmp	1	880	mit	14	45.75	Lua	10209

Table 5.5.: All candidates satisfying $REQ1_{EC}$ (Arch. Doc.) after the Automated Filtering Process in AFP_{Lua}, and the corresponding RADs.

Project Name	$REQ1_{EC}$ (Arch. Doc.)	$REQ2_{EC}$ (Several Commits)	$REQ4_{EC}$ (Open-Source)	$REQ5_{EC}$ (Perf. Tests)	$REQ6_{EC}$ (Activity)	$REQ8_{EC}$ (Pr. Language)	$REQ9_{EC}$ (Scale)
doocs/advanced-java	10	762	cc-by-sa-4.0	5	0	Java	0
crossoverJie/JCSprout	2	713	mit	59	0	Java	7058
google/ExoPlayer	2	18875	apache-2.0	985	0	Java	479128
alibaba/arthas	1	2044	apache-2.0	278	0.125	Java	78192
doocs/source-code-hunter	1	621	cc-by-sa-4.0	40	0.375	Java	0

Table 5.6.: Essential candidates from IC_{Java} that are not included in the set of filtered candidates for AFP_{Lua} and the corresponding RADs.

Results for AFP_{Java}. The requirement satisfaction rate for AFP_{Java} (M1.1.1a) is depicted in Figure 5.3. Here, the processed candidates constitute 82 of the initial 100 candidates. As with AFP_{Lua}, REQ1_{EC} (Arch. Doc) demonstrates the lowest satisfaction rate with 25.71%, followed by REQ6_{EC} (Activity) with 59.76%. The remaining requirements have satisfaction rates in excess of 90%, with REQ2_{EC} (Several Commits), REQ5_{EC} (Perf. Tests) and REQ8_{EC} (Pr. Language) being satisfied by all processed projects.
The ratio of essential candidates to processed candidates is 23.17% (M1.1.1b), while the ratio of filtered candidates to processed candidates amounts to 17.07% (M1.1.2), as shown in Table 5.7. The filtered candidates from and the corresponding RADs are depicted in Table 5.8. The essential candidates that are not included in the set of filtered candidates are shown in Table 5.6. These candidates all fail to satisfy REQ6_{EC} (Activity). For two of these candidates, "doocs/advanced-java" and "doocs/source-code-hunter", the Automated Filtering Process did not detect any Java code at all, with the RAD for REQ9_{EC} (Scale) indicating $LOC(\texttt{.java}) = 0$.

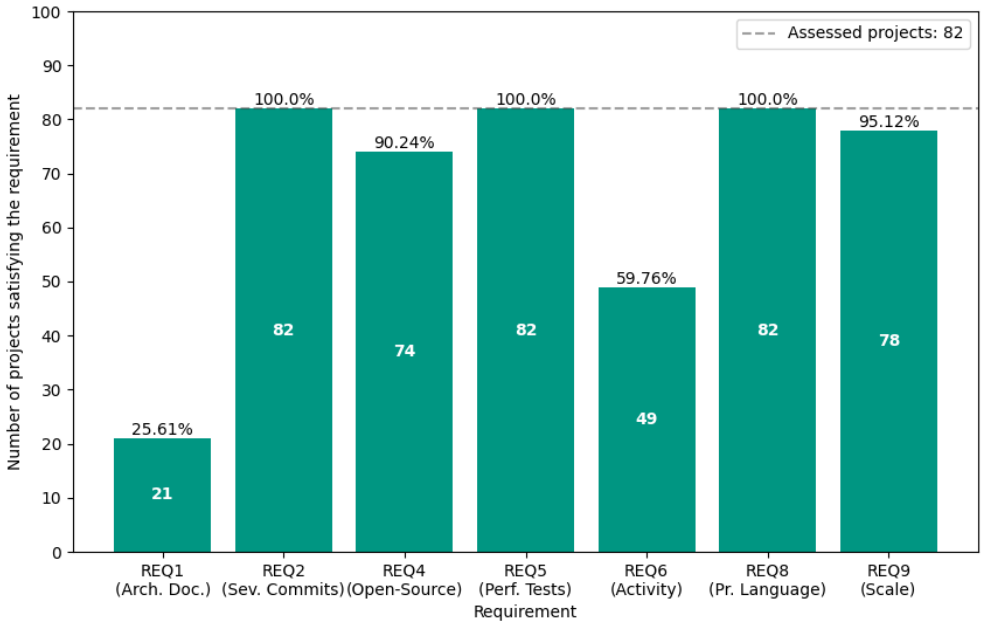


Figure 5.3.: Satisfaction rate for each requirement after the Automated Filtering Process in relation to the total number of initial candidates for AFP_{Java} (M1.1.1a).

Results for AFP_{Union}. The final run was primarily conducted to verify the correct behaviour of the Automated Filtering Process when applied to initial candidates from multiple programming languages. The requirement satisfaction rate for this run is shown in Figure 5.4, with 163 processed candidates out of the 200 initial candidates. Once again, REQ1_{EC} (Arch. Doc.) has the lowest satisfaction rate. However, REQ8_{EC} (Pr. Language) at 52.76% (Pr. Language) and REQ9_{EC} (Scale) at 50.31% exhibit satisfaction rates comparable to REQ6_{EC} (Activity) at 51.53%. The resulting filtered candidates, listed in Table 5.9, are identical to those of AFP_{Lua}, with minor differences in the RAD assessed for REQ6_{EC} (Activity).

Run	Initial Candidates	Processed Candidates	Essential Candidates	Ratio Essential	Filtered Candidates	Ratio Filtered
AFP_{Lua}	100	86	4	4.65%	4	4.65%
AFP_{Java}	100	82	19	23.17%	14	17.07%

Table 5.7.: Ratios of essential candidates to processed candidates (Ratio Essential) for IC_{Lua} and IC_{Lua} (M1.1.1b), and ratios of filtered candidates to processesd candidates (Ratio Filtered) resulting from AFP_{Lua} and AFP_{Java} (M1.1.2).

Project Name	REQ1_{EC} (Arch. Doc.)	REQ2_{EC} (Several Commits)	REQ5_{EC} (Perf. Tests)	REQ6_{EC} (Activity)	REQ9_{EC} (Scale)
apache/flink	66	36476	9523	36.375	2424952
apache/shardingsphere	33	45008	5788	65.625	556737
NationalSecurityAgency/ghidra	12	13829	3624	52.0	2936290
apache/kafka	9	14924	2741	69.75	1207506
apache/rocketmq	8	8737	845	4.875	341040
spring-projects/spring-boot	6	52643	5812	86.75	781210
eugenp/tutorials	6	18722	18343	37.25	970446
apolloconfig/apollo	5	2925	391	1.5	70319
spring-projects/spring-framework	4	32080	4852	54.25	1416413
jenkinsci/jenkins	2	35956	1417	36.75	317124
conductor-oss/conductor	2	3816	831	7.375	157816
SeleniumHQ/selenium	2	32561	1250	31.125	173714
apache/skywalking	1	8201	953	8.0	191872
keycloak/keycloak	1	27313	3537	49.625	901476

Table 5.8.: Filterd candidates and corresponding RADs yielded by the Automated Filtering Process in AFP_{Java}. RADs for REQ4_{EC} (Open-Source) and REQ8_{EC} (Pr. Language) omitted for clarity.

Table 5.10 provides an overview of the execution time of the different runs (M1.1.3), alongside the average LOC per candidate calculated using $LOC(\texttt{.ext})$ as determined by the Automated Filtering Process. The process required 618s to process IC_{Lua}, with 68691 average LOC; 3631s for IC_{Java}, with 333222 average LOC; and 3710s for IC_{Union} with 24049 average LOC.

Manual inspection. Experiment E_{EC} includes a manual inspection ($\mathbf{A}_{\text{Manual}}$) of the filtered candidates from AFP_{Lua} to select a final set of evaluation cases for goals **G2** and **G3**. Typically, $\mathbf{A}_{\text{Manual}}$ involves assessing evaluation cases based on requirements REQ3_{EC} (Arch.

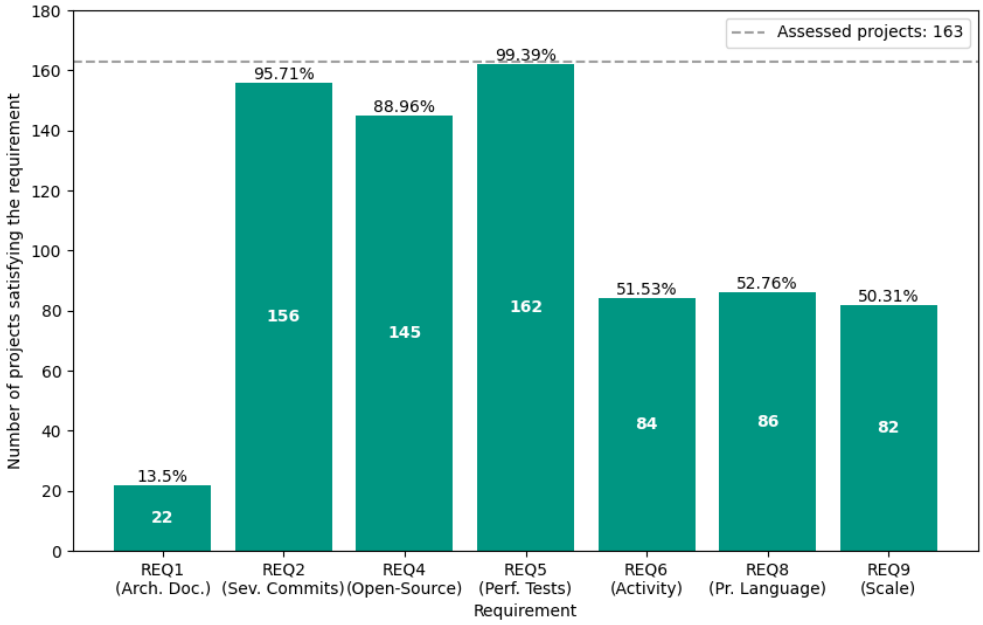


Figure 5.4.: Ratio of candidates satisfying each requirement after the Automated Filtering Process in relation to the total number of initial candidates for AFP_{Union}.

Project Name	REQ1_{EC} (Arch. Doc.)	REQ2_{EC} (Several Commits)	REQ5_{EC} (Perf. Tests)	REQ6_{EC} (Activity)	REQ9_{EC} (Scale)
ntop/ntopng	5	23966	739	28.25	269583
apache/apisix	4	4366	643	4.0	59892
xmake-io/xmake	1	15385	764	43.75	186557
Saghen/blink.cmp	1	880	14	46.875	10209

Table 5.9.: Filterd candidates and corresponding RADs yielded by the Automated Filtering Process in AFP_{Union}. RADs for REQ4_{EC} (Open-Source) and REQ8_{EC} (Pr. Language) omitted for clarity.

Changes) and REQ7_{EC} (Tech. & Domain) for the evaluation of CIPM-related research goals. However, **G2** forms an exception. Although **G2** is indirectly related to CIPM research – as it pertains to the satisfaction of REQ1_{CM}-REQ3_{CM} for CMoGS, which were defined based on the requirements of the CIPM approach – its focus is on the general satisfaction of these requirements without direct connection to the CIPM approach. Therefore, all filtered candidates resulting from AFP_{Lua} are selected for the evaluation of G2, without further assessment based on REQ3_{EC} and REQ7_{EC}.
In contrast, **G3** is directly related to CIPM, as it involves the integration of the Lua CMoGS into the CIPM pipeline. The following outlines the assessment through $\mathbf{A}_{\text{Manual}}$ for the selection of final candidates for this goal, as performed in the experiment.
Given that the Automated Filtering Processed assesses REQ5_{EC} (Perf. Tests) and REQ1_{EC}

Run	Initial Candidates	Processed Candidates	Filtered Candidates	Average LOC ($REQ9_{EC}$)	Execution Time (s)
AFP_{Lua}	IC_{Lua}	86	4	68691	618
AFP_{Java}	IC_{Java}	82	14	333222	3631
AFP_{Union}	IC_{Union}	163	4	24049	3710

Table 5.10.: Summarised information about the different runs of the Automated Filtering Process performed in $\mathbf{E}_{\text{EC}}$, with average LOC over all initial candidates and execution times.

(Arch. Doc.) through $\mathbf{A}_{\text{Keyword}}$, without considering the context in which the keywords and -phrases appear, the results require validation by $\mathbf{A}_{\text{Manual}}$ (cf. Section 4.1.5). Because the evaluation of **G3** does not include an evaluation of the calibration of PMPs and the accuracy of performance prediction through the aPM, $REQ5_{EC}$ is not assessed further by $\mathbf{A}_{\text{Manual}}$ in this experiment.
With respect to $REQ1_{EC}$ (Arch. Doc.), the keyword match detected for candidate "Saghen/blink.cmp" points to an `architecture.md` file describing the pipeline architecture of the project [78]. Similarly, the candidates "apache/apisix" and "ntop/ntopng" provide architectural documentations [26, 69]. Unfortunately, the match for the candidate "xmake-io/xmake" was detected in an `architectures.lua` file, which does not appear to be related to architectural documentation of the corresponding candidate [99], excluding it from further consideration.
$REQ3_{EC}$ (Arch. Changes) is assumed to be satisfied by all filtered candidates, given the high commit count and LOC, which suggest the existence of architecturally relevant commits, as well as a reasonably high probability of containing architecturally irrelevant commits.
Finally, because this thesis investigates the generalisability of the CIPM approach and proposes an approach to generalised CPRs, $REQ7_{EC}$ (Tech. & Domain) is considered of lesser importance for the evaluations of goal **G3**.
Thus, the final selection of suitable evaluation cases, resulting from the application of the Approach to Semi-Automated CIPM Evaluation Case Selection in this experiment, consists of the remaining three software projects: "ntop/ntopng", "apache/apisix", and "Saghen/blink.cmp".

5.3.3. E_{EC}: Discussion

This section discussed the results from experiment E_{EC} based on questions Q1.1.1-Q1.2.1, and their implications for the satisfaction of **G1**.

Satisfaction rate per requirement. The results show that most evaluation case requirements for CIPM ($REQ2_{EC}$, $REQ3_{EC}$, $REQ4_{EC}$, $REQ5_{EC}$, $REQ8_{EC}$, $REQ9_{EC}$) are satisfied by a large number of popular projects in Lua and Java. This might suggest that the RAD and filter configurations used in the experiment were too inclusive. However, for most of these requirements (except $REQ5_{EC}$), this is not the case.
For instance, the configuration of $REQ2_{EC}$ (Several Commits) ensures a sufficient number of commits, which in turn increases the likelihood of satisfying $REQ3_{EC}$ (Arch. Changes), as examined briefly through the manual inspection ($\mathbf{A}_{\text{Manual}}$) conducted in the experiment. Setting this filter to a relatively low threshold also allows the resulting RAD to serve as an additional indicator for $REQ9_{EC}$ (Scale), in cases where research goals require the consideration of the number of commits as part of a candidate's scale. Similarly, selecting a suitably low threshold for $LOC(\texttt{.ext})$ ($REQ9_{EC}$) helps in identifying projects of varying scale via $\mathbf{A}_{\text{Manual}}$.
The high satisfaction rates for $REQ4_{EC}$ (Open-Source) and $REQ8_{EC}$ (Pr. Language) can be attributed to the selection of initial candidates. Since all projects from the Top 100 GitHub repositories are at least openly accessible and correspond to the respective programming language, these requirements had an inherently high satisfaction rate.
In contrast to these requirements, the satisfaction rate of $REQ5_{EC}$ (Perf. Tests) suggest that the keywords and -phrases used in KW_T may have been too inclusive. While this has little impact on the present thesis – given that $REQ5_{EC}$ is of lesser importance for the goals considered here – future fork should refine KW_T (e.g., using more precise keywords and -phrase like "performance test" or "load test") and analyse the resulting matches to improve the assessment of this requirement.
$REQ7_{EC}$ (Tech. & Domain) was not assessed by either the Automated Filtering Process or $\mathbf{A}_{\text{Manual}}$ in this thesis, as the aspired generalisation of the CIPM approach is designed for general applicability across different technologies and domains. However, since this requirement may be relevant for other research objectives, its satisfaction rate and potential methods for automated assessment should be explored in future work.
The remaining requirements, $REQ1_{EC}$ (Arch. Doc.) and $REQ6_{EC}$ (Activity), showed the lowest satisfaction rates, and are discussed in more detail in the following.

Existence of essential & filtered candidates. The results indicate that both sets of initial candidates, IC_{Lua} and IC_{Java}, contain projects suitable for use as CIPM evaluation cases based on the Automated Filtering Process and the requirement set REQ_{AFP}. However, only 4.65% of the Lua candidates and 23.17% of the Java candidates satisfy the essential requirements $REQ1_{EC}$ (Arch. Doc.), $REQ2_{EC}$ (Several Commits), and $REQ4_{EC}$ (Open-Source). Among these, $REQ1_{EC}$ (Arch. Doc.) has the lowest satisfaction rate across both sets, suggesting that a significant proportion of software projects lack architectural documentation.
As discussed in Section 4.1.2, $REQ6_{EC}$ (Activity) can support the fulfilment of $REQ1_{EC}$ by increasing the likelihood of availability of project actors who may contribute to validated

architectural documentation. However, REQ6$_{EC}$ also has a relatively low satisfaction rate compared to the other requirements. Despite this, with 45.35% of Lua candidates and 59.76% of Java candidates satisfying REQ6$_{EC}$, a considerable number of candidates still satisfy this requirement.
For IC_{Lua}, the filtered candidates returned by the Automated Filtering Process are identical to the essential candidates. For IC_{Java}, however, five out of 19 essential candidates are excluded from the filtered candidates. Notably, the Automated Filtering Process detected 0 LOC for two of these candidates, highlighting that GitHub's Top 100 ranking does not necessarily guarantee the presence of code in the corresponding programming language. This underscores the importance of assessing REQ2$_{EC}$ (Several Commits) and REQ9$_{EC}$ (Scale).
Overall, these findings demonstrate that the Automated Filtering Approach significantly reduces the number of initial candidates, thereby minimising the manual effort required for selecting evaluation cases in CIPM-related research.

AFP$_{\text{Union}}$ & Execution times. The application of the Automated Filtering Process to the IC_{Union} produced the expected behaviour, yielding filtered candidates identical to those of AFP_{Lua}. The minor discrepancies observed in the RAD for REQ6$_{EC}$ (Activity) can be attributed to the sequential execution of the respective runs, with AFP_{Union} being executed several hours after AFP_{Lua}, resulting in different commits being included in the corresponding timeframes.
The satisfaction rates of REQ8$_{EC}$ (Pr. Language) and REQ9$_{EC}$ (Scale) confirm the correct filtering of projects based on MLP = "Lua" and LOC(`.lua`), in accordance with CFG_{Lua}.
However, the implementation of the Automated Filtering Process does not process the same number of candidates in AFP_{Lua} and AFP_{Java}. While a total of 168 candidates were processed across AFP_{Lua} and AFP$_{\text{Java}}$, only 163 candidates were processed in AFP_{Union}. This discrepancy suggests technical issues in the implementation, possibly due to concurrent processing causing excessive requests to the GitHub API within a short time frame. To address this, the implementation should be improved by incorporating retry mechanisms for failed requests and delays to prevent multiple concurrent API requests.

Limitations. In addition to the aforementioned limitations regarding KW_T and the discrepancies in processed candidates between runs, further limitations of the approach to Semi-Automated CIPM Evaluation Case Selection were identified through experiment E$_{EC}$.
One key limitation is the lack of support for projects using multiple programming languages. As demonstrated by AFP_{Union}, the current approach only allows configurations specific to a single programming language, discarding all projects with a different MPL. However, selecting evaluation cases that involve multiple languages is valuable for researching the generalisability of the CIPM approach, for instance for research investigating automatic updated to aPMs for projects using multiple programming languages.
Additionally, the approach does not currently account for programming language versions. Since the CMoGS applied in the CIPM approach may be restricted to certain language versions – such as the extended Lua CMoGS, which supports only up to Lua 5.2 – evaluation cases be filtered not just by programming language but also by version to ensure compatibility with the corresponding CMoGS.

Implications for G1. The results demonstrate that the Automated Filtering Process satisfies subgoal **G1.1**. It significantly reduces the number of candidates requiring manual assessment via $\mathbf{A}_{\text{Manual}}$, with IC_{Lua} being reduced by 95.35% and IC_{Java} by 82.93%. Additionally, the generated RADs support the selection process through $\mathbf{A}_{\text{Manual}}$, for instance by providing insights into a project's commit history through the RAD $|C|$ for REQ2_{EC} (Several Commits), which helps to infer the likelihood of REQ3_{EC} (Arch. Changes) being satisfied.
Furthermore, the results show that by applying $\mathbf{A}_{\text{Manual}}$ to the filtered candidates as the final step in the Process for Semi-Automated CIPM Evaluation Case Selection, viable evaluation cases for assessing G2 and G3 can be selected. This provides initial evidence toward satisfying **G1.2**.

In summary, while the ultimate suitability of the selected evaluation cases can only be fully assessed after evaluating **G2** and **G3**, experiment E_{EC} already demonstrates promising results for **G1**.

5.3.4. Threats to Validity

Several threats to validity must be considered for the performed evaluation of the Approach to Semi-Automated CIPM Evaluation Case Selection.

One significant threat to internal validity is the selection of the initial candidate sets. The evaluation relies on repositories sourced from GitHub, specifically those from the the Top 100 rankings for the Lua and Java programming languages. This essentially results in a pre-filtering step, which may bias the selection process, as it excludes projects that may be relevant but do not appear in these rankings.
Another internal validity threat arises from the configuration of the filtering process. For instance, the configurations used in this evaluation consider specific sets of keywords for certain requirements, such as REQ1_{EC} (Arch. Doc.). However, the selected keyword sets may not fully or accurately capture the intended requirements, leading to false negatives and resulting in the exclusion of suitable projects.
This issue with keyword-based filtering exemplifies a broader threat to validity: the evaluation does not include an explicit assessment of the precision of the selected RADs. Precision, in this context, refers to whether a RAD truly captures the information it is intended to capture – for instance, if the occurrences of certain keywords regarding architectural documentation actually identify the presence of architectural documentation. While the RADs and filters were chosen based on reasonable assumptions to ensure they reflect the required information, their precision was not systematically evaluated.

A key threat to external validity is the reliance on GitHub as the sole source for evaluation case selection. Restricting the selection to GitHub repositories introduces a platform-specific bias, and as a result, the findings based on this dataset may not generalise to software projects hosted elsewhere.
Another external validity threat relates to the specific scope of the Approach to Semi-Automated CIPM Evaluation Case Selection. The approach focuses on generating a data basis for investigations into the generalisability of the CIPM approach from the perspective

of identifying sets of evaluation cases. However, it does not account for other research objectives, such as the extraction of commonalities between software projects for evaluating the generalisability of the CPRs employed in CIPM. Consequently, the findings may not extend to other research efforts requiring sets of software projects as a data basis for investigations into the generalisability of the CIPM approach.

5.4. G2 & G3: Extended Lua Code Model Generator and Serialiser

This section presents the evaluation of the extended Lua CMoGS. To this end, the experiments conducted to evaluate the satisfaction of the requirements REQ1$_{CM}$-REQ3$_{CM}$ are described, followed by the corresponding results and their discussion.

5.4.1. Experiment E$_{CM,T}$

The first experiment to evaluate the satisfaction of the requirements REQ1$_{CM}$-REQ3$_{CM}$, denoted as E$_{CM,T}$, consists of executing a series of benchmark tests from two test suites, T_{GS} and T_{RR}.

Test Suite T$_{GS}$. T_{GS} evaluates the generalisation and serialisation mechanisms of the extended Lua CMoGS, denoted as $CMoGS_E$ in the following, by applying them to selected cases of Lua input code and computing the Generation-Serialisation Similarity *Sim*. T_{GS} consists of 39 test cases, selected based on the limitations of the original Lua CMoGS (cf. Section 4.2.4), denoted as $CMoGS_O$, as well as expected functionality extracted from the Lua 5.2 reference [71]. To illustrate the process performed by the test cases in T_{GS}, consider Listing 5.1, which contains the Lua import statements already discussed in Section 4.2.4.

```
1 require("path.to")
2 require"path.to"
3 local external_service = require("path.to").service
```

Listing 5.1: Examples of different import statements in Lua used as input for test cases in T_{GS}.

Let these statements be denoted as s_i for the statement in line i of this listing. For each statement s_i, a test case exists in T_{GS} applying the generation and serialisation mechanisms of the extended Lua CMoGS to the given statement to compute $Sim(s_i, CMoGS_E)$. The test fails iff $Sim \neq 1$.
In addition their application to $CMoGS_E$, the benchmark tests are also applied to $CMoGS_O$, to illustrate the differences in their behaviour. This comparison is not represented by a dedicated metric in the GQM-Plan as it is performed solely for illustrative purposes.

Test Suite T$_{RR}$. The test cases contained in T_{RR} test the Generation-Serialisation Similarity analogous to the test cases in T_{GS}. However, they also test for correct reference resolution, by ensuring that no references were resolved through trivial recovery. In addition, test cases in T_{RR} assert that the resolved references in the generated CM correspond to the references in the Lua code used as input for the CM generation. To this end, the expected referenced elements for referencing elements in the CM are defined and verified in the test case. The Lua sumneko language server [58] is used as a gold standard for the definition of the expected referenced elements. The verification is performed automatically for 8 test cases, and manually (by inspecting the resulting CM) for the remaining cases.

T_{RR} consists of 44 test cases selected based on the challenges for static reference resolution in dynamically typed programming languages discussed in Section 4.2.2 and the implementation boundaries defined in Section 4.2.7. As such, T_{RR} focuses on test cases evaluating the functionality of Feature resolution of $CMoGS_E$, as presented in Section 4.2.5.2. Additionally, it contains tests evaluating the functionality of the resolution of other Lua language features, such as the functionality of the `goto` keyword introduced in Lua 5.2 [71], which are not resolved using Feature resolution.

5.4.2. $E_{CM,T}$: Results

The results for metrics M2.2.1a and M2.3.1, obtained from experiment $E_{CM,T}$, are presented in Table 5.11. All test cases in T_{GS} passed successfully, while one test case failed in T_{RR}, resulting in a 97.74% pass rate for T_{RR}.

Test Suite	Total Number of Tests	Number Passed	Ratio Passed
T_{GS}	39	39	100%
T_{RR}	44	43	97.74%

Table 5.11.: Results for metrics M2.2.1a and M2.3.1 meaured by conducting $E_{CM,T}$.

As mentioned in the description of $E_{CM,T}$, the tests from T_{GS} were additionally executed on $CMoGS_O$. An excerpt of the results is presented in Table 5.12 to further illustrate the testing process and the limitations of the original CMoGS. In total, 33 of the 39 test cases failed for $CMoGS$, resulting in a pass rate of 15.38%.

5.4.3. $E_{CM,T}$: Discussion

Experiment $E_{CM,T}$ yielded promising results regarding the satisfaction of $REQ1_{CM}$-$REQ3_{CM}$.

All tests from T_{GS} were successfully passed by $CMoGS_E$, indicating that the extension developed in this thesis significantly improves the usability of the Lua CMoGS, especially when compared to the performance of $CMoGS_O$.

However, the tests from T_{CM} reveal a limitation of the implemented extension, as one test case from this suite failed. The failed test case involves resolving index expressions from string literals containing dots, e.g., `table["acc.ess"]`. The implementation fails to handle such cases correctly due to technical limitations, instead resorting to trivial recovery. This could significantly impact the accuracy of the Lua CMoGS reference resolution mechanism in software projects that heavily use dot notation for table accesses. Nevertheless, the successful execution of all other tests from T_{CM} suggests that the reference resolution process functions correctly in most cases.

That said, as famously stated by Dijkstra:

Category	ID	Input	Result for CMoGS$_O$
Import Statement	t1$_{GS}$	`require("path.to")`	x
Import Statement	t2$_{GS}$	`require"path.to"`	
Import Statement	t3$_{GS}$	`f = require("path.to").func`	
Table Access lhs	t4$_{GS}$	`t.x = 1;`	x
Table Access lhs	t5$_{GS}$	`t["x"] = 1;`	
Table Access lhs	t6$_{GS}$	`t[0] = 1;`	
Table Access lhs	t7$_{GS}$	`t[func()] = 1;`	
Table Access rhs	t8$_{GS}$	`x = t.x;`	
Table Access rhs	t9$_{GS}$	`x = t["x"];`	
Table Access rhs	t10$_{GS}$	`x = t[0];`	
Table Access rhs	t11$_{GS}$	`x = t[func()];`	
Return Value Access	t12$_{GS}$	`func().x;`	
Return Value Access	t13$_{GS}$	`func()["x"];`	
Return Value Access	t14$_{GS}$	`func()[0];`	
Return Value Access	t15$_{GS}$	`func()();`	

Table 5.12.: Excerpt of the results of the benchmark tests from T_{GS} for REQ2$_{CM}$ for $CMoGS_o$. An 'x' signifies that the corresponding benchmark test was passed.

"Testing shows the presence, not the absence of bugs." [16, p. 16]

Thus, the results from E$_{CM,T}$ provide only an indication of the satisfaction of REQ1$_{CM}$-REQ3$_{CM}$ and, consequently, a partial answer to questions Q2.1-Q2.3. The experiment discussed in the remainder of this section aim to further reinforce this indication.

5.4.4. Experiment E$_{CM,EC}$

Experiment E$_{CM,EC}$ is the second experiment designed evaluate the satisfaction of the requirements REQ1$_{CM}$-REQ3$_{CM}$ by $CMoGS_E$. This experiment applies the CMoGS to a collection of evaluation cases within CIPM's CI-based update of software models.

The selection of Lua evaluation cases aims to cover all Lua 5.2 language constructs. To achieve this, the Lua 5.2 test suite [56] is used as one evaluation case. While this test suite does not represent a real-world software project, it can be reasonably assumed to encompass all language constructs, making it a suitable choice for evaluation. As the Lua test suite

does not provide a Git repository, the experiment for this evaluation case is restricted to the generation and serialisation of the corresponding CM through the extended Lua CMoGS outside of the CIPM pipeline.

In addition, two real-world software projects, "Saghen/blink.cmp" and "apache/apisix", are selected as evaluation cases based on the results from experiment E_{EC}.
The first of these, "Saghen/blink.cmp" [78], is a completion plugin for the vim-based text editor Neovim [89], representing a real-world software project in which Lua is used as an extension language. In contrast, "apache/apisix" [27, 25] is a Lua-based API gateway that provides traffic management features. The selection of these project ensures diversity across software domains.
To assess the extended Lua CMoGS inside CIPM's CI-based update of software models, i.e. its applicability to incoming commits within CI/CD pipelines, five commits from each project are selected for evaluation. The selection of the commits is performed as follows:

- The commit introducing the architectural documentation detected through E_{EC} is identified.
- Starting from this commit, the commit messages of subsequent commits are manually inspected.
- The first set of five commits is selected, for which the commit messages imply the existence of architecturally relevant and architecturally irrelevant commits. To this end, it is assumed that commit messages starting with the keyword "feat" imply architecturally relevant changes by introducing a new feature, whereas commit messages starting with "chore", "fix", or "docs" imply architecturally irrelevant changes.

The resulting commits selected for the real-world evaluation cases are summarised in Tables 5.13 and 5.14, providing the short commit hash, date of the commit, and the commit message.

Commit Hash	Date	Commit Message
9731f51	6.12.2021	docs: Software architecture diagram added (zh-docs)
49762bc	7.12.2021	feat: rocketmq logger
3d5f554	7.12.2021	docs: update README.md
a4b6931	7.12.2021	feat(wasm): allow running in the rewrite phase
b85ebd4	7.12.2021	fix(patch): add global ‘math.randomseed‘ patch support

Table 5.13.: Commits selected for "apache/apisix" for $E_{CM,EC}$.

To enable an initial evaluation of the model updates in the CI-based update of software models, the `FileDetectionStrategy` introduced in Section 4.2.8 is applied as the detection strategy for the components of the real-world evaluation cases.

Commit Hash	Date	Commit Message
a9a0f96	21.12.2024	chore: bump version to 0.8.1
a937edd	21.12.2024	feat: improve auto_show flexibility
4ef6d1e	22.12.2024	fix: auto_show function logic
8620a94	22.12.2024	docs: Add missing window table to set the documentation border
f93af0f	22.12.2024	fix(notifications): add title to notifications

Table 5.14.: Commits selected for "Saghen/blink.cmp" for $E_{CM,EC}$.

For "apache/apisix", the detection strategy is configured according to the provided architectural documentation, shown in Figure 5.5. Based on this documentation, all Lua files in the corresponding repository are assigned to components as specified in Table 5.15.
Unfortunately, in the case of "Saghen/blink.cmp", the directory and file names do not imply any direct correspondance to the provided architectural documentation [78]. Consequently, the corresponding mapping simply specifies one component for each directory inside the project, as well as an additional component for any files at the root of the project (Table 5.16).

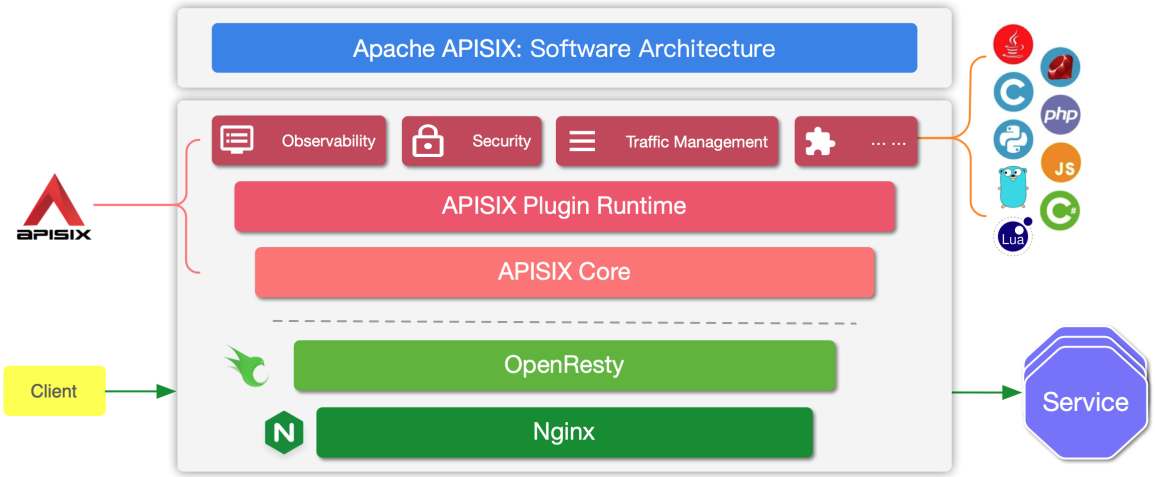


Figure 5.5.: Architecture documentation diagram provided by "apache/apisix" [27].

Measurements for the requirements $REQ1_{CM}$-$REQ1_{CM}$ are collected as follows by conducting $E_{CM,EC}$:

- For all evaluation cases, the Generation-Serialisation Similarity *Sim* is computed for all files of the evaluation case, as described in Section 5.2 (M2.2.1b). This enables the evaluation of the satisfaction of $REQ1_{CM}$ and $REQ2_{CM}$.
  For evaluation cases with multiple commits, this data is computed for each commit (M2.2.2).

Component Name	Corresponding Files
Core	All files in the `core` directory, and the `core.lua` file.
Plugin	All files in the `plugins` directory, and the `plugins.lua` file.
Plugin Runtime	All other files.

Table 5.15.: Mapping from files to components specified for the `FileDetectionStrategy` applied to "apache/apisix".

Component Name	Corresponding Files
Completion	All files in the `completion` directory.
Config	All files in the `config` directory.
Fuzzy	All files in the `fuzzy` directory.
KeyMap	All files in the `keymap` directory.
Lib	All files in the `lib` directory.
Signature	All files in the `signature` directory.
Sources	All files in the `sources` directory.
Other	All other files.

Table 5.16.: Mapping from files to components specified for the `FileDetectionStrategy` applied to "Saghen/blink.cmp".

- The resolution of all references, either non-synthetically or synthetically, is checked for each evaluation case and commit (M2.3.2).
- Data relating to non-synthetically resolved references, critical synthetic references, non-critical synthetic references, as well as the corresponding reference types and causes, is collected for Metrics M2.3.2a-c. As specified in the GQM-Plan (Section 5.2), synthetic references are divided into critical and non-critical, based on the detected cause:
  Critical synthetic references are synthetic references caused by

$$\text{Critical Causes} := \{C_{FR}, C_{DA}, C_U\},$$

  whereas non-critical synthetic references are synthetic references caused by

$$\text{Non-Critical Causes} := \{C_{Ext}, C_{SL}, C_{Arg}\}.$$

  For evaluation cases with multiple commits, this data is aggregated accross all commits (cf. Section 5.2).

- The Jaccard Index between the CM generated by the extended CMoGS and the CM within the Vitruvius V-SUM is computed (M3.1).
- The components within the generated `Repository` models of the PCM for each commit are compared manually to the expected components specified by the corresponding `FileDetectionStrategy` (M3.2.1).

5.4.5. $E_{CM,EC}$: Results

This section presents the results for $E_{CM,EC}$. Unfortunately, due to technical issues, Metrics M3.1 and M3.2.2 could not be measured. The cause of these issues is likely related to the technical adaptations made to the matching strategy, as described in Section 4.2.8.

Similarity & complete reference resolution. The measurements for Generation-Serialisation Similarity (M2.2.1b & M2.2.2) yielded a value of 1 for all evaluated projects and commits, as shown in Table 5.17. Additionally, this table demonstrates the complete resolution of all references for all evaluation cases and commits, through either non-synthetic or synthetic reference resolution (M2.3.2).

Project	Commit Hash	Generation-Serialisation Similarity	Ratio of Resolved References
Lua 5.2 Test Suite	-	1	100%
apache/apisix	9731f51	1	100%
	49762bc	1	100%
	3d5f554	1	100%
	a4b6931	1	100%
	b85ebd4	1	100%
Saghen/blink.cmp	a9a0f96	1	100%
	a937edd	1	100%
	4ef6d1e	1	100%
	8620a94	1	100%
	f93af0f	1	100%

Table 5.17.: Generation-Serialisation Similarity (M2.2.1b & M2.2.2) for all evaluation cases and commits, and ratio of resolved references including synthetic references (M2.3.2).

Reference Distributions. Figures 5.6, 5.7, and 5.8 present the distributions of resolved references, distinguishing between non-synthetic, critical synthetic, and non-critical synthetic references for the evaluation cases Lua 5.2 test suite, "apache/apisix", and "Saghen/blink.cmp", respectively. Each figure consists of two subfigures: (a) depicting the distribution across all resolved references (M2.3.2a) and (b) showing the distributions for each reference type (M2.3.2b), namely Function Call (FC), Table Access (TA), and Root (R).

For the Lua 5.2 Test Suite, the reference resolution mechanism resolved 86.23% of all references non-synthetically, with 11.82% of references being resolved through non-critical synthetic references and 1.95% through critical synthetic references (Figure 5.6a).
Regarding reference types, 33.89% were FC, 7.82% were TA, and 55.29% were R references

(Figure 5.6b). Among FC references, 74.46% were non-synthetic, 24.93% were non-critical synthetic, and 0.62% were critical synthetic references. For TA references, 49.18% were non-synthetic, 25.53% were non-critical synthetic, and 25.29% were critical synthetic references. Finally, 96.03% of all R references were non-synthetic, while 3.97% were non-critical synthetic, with no critical synthetic references.

For "apache/apisix", across all five evaluated commits[1], the reference resolution mechanism resolved 81.97% of all references non-synthetically, while 15.76% were resolved through non-critical synthetic references and 2.27% through critical synthetic references (Figure 5.7a).
With respect to the distribution of reference types, 14.28% were FC, 19.76% were TA, and 65.96% were R references (Figure 5.7b). Among FC references, 57.16% were non-synthetic, 40.73% were non-critical synthetic, and 2.11% were critical synthetic. For TA references, 21.48% were non-synthetic, 65.67% were non-critical synthetic, and 12.85% were critical synthetic. Lastly, 98.64% of all R references were non-synthetic, while 1.36% were non-critical synthetic, with no critical synthetic references.

Finally, across all five evaluated commits for "Saghen/blink.cmp", the reference resolution mechanism resolved 75.69% of all references non-synthetically, while 19.47% were resolved through non-critical synthetic references and 4.84% through critical synthetic references (Figure 5.8a).
With regard to the categorisation of reference types, 16.41% were classified as FC, 20.76% as TA, and 62.83% as R references (Figure 5.8b). Among FC references, 41.79% were non-synthetic, 43.17% were non-critical synthetic, and 15.04% were critical synthetic. For TA references, 25.66% were non-synthetic, 58.38% were non-critical synthetic, and 15.96% were critical synthetic. Lastly, 93.54% of all R references were non-synthetic, while 6.46% were non-critical synthetic. As with the other evaluation cases, no critical synthetic references were present for reference type R.

Cause distributions. Figures 5.9, 5.10, and 5.11 illustrate the distributions of detected causes for synthetic references (M2.3.2c). Again, each figure consists of two subfigures: (a) showing the distribution for Function Call (FC) references, and (b) showing the distribution for Table Access (TA) references. Given that the Cause Detection Algorithm assumes accesses to external libraries (C_{Ext}) as the cause for all synthetically resolved Root (R) references (cf. Section 4.2.6), the only detected cause for R references was C_{Ext} for all evaluation cases. Thus, no separate figures are provided for this reference type.

For the Lua 5.2 test suite, 78.73% of all synthetic FC references (Figure 5.9a) were caused by calls to the Lua standard library (C_{SL}), 18.4% by calls to external libraries (C_{Ext}) and 0.07% by accesses to argument fields (C_{Arg}), resulting in their categorisation as non-critical. Regarding the synthetic references categorised as critical, 0.39% of all synthetic references were caused by accesses to fields of tables returned by functions (C_{FR}), 0.46% by other accesses to table fields (C_{DA}), and 1.96% by unidentified causes (C_U).

[1] Individual reference resolution results for each commit of the evaluation candidates "apache/apisix" and "Saghen/blink.cmp" are provided in the appendix.

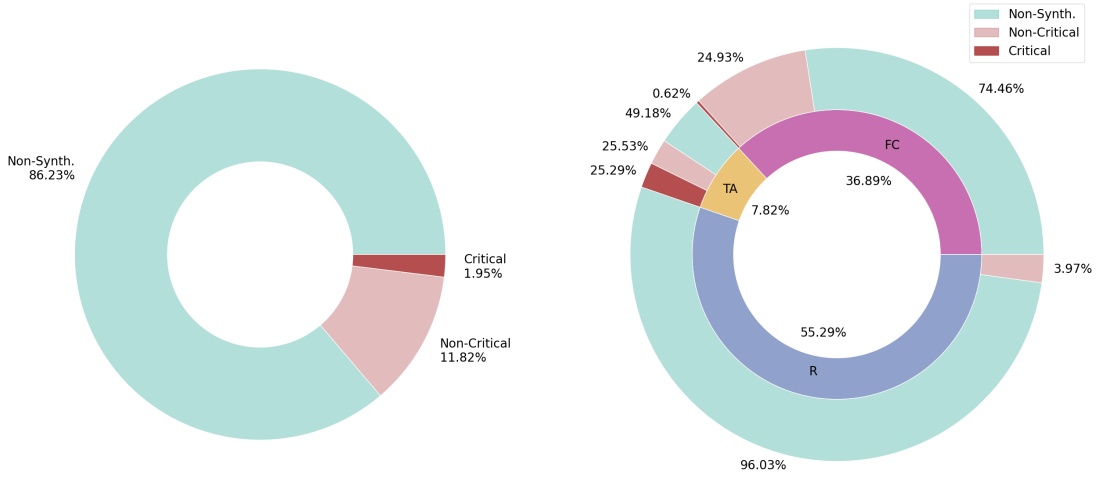


Figure 5.6.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for the Lua 5.2 test suite.

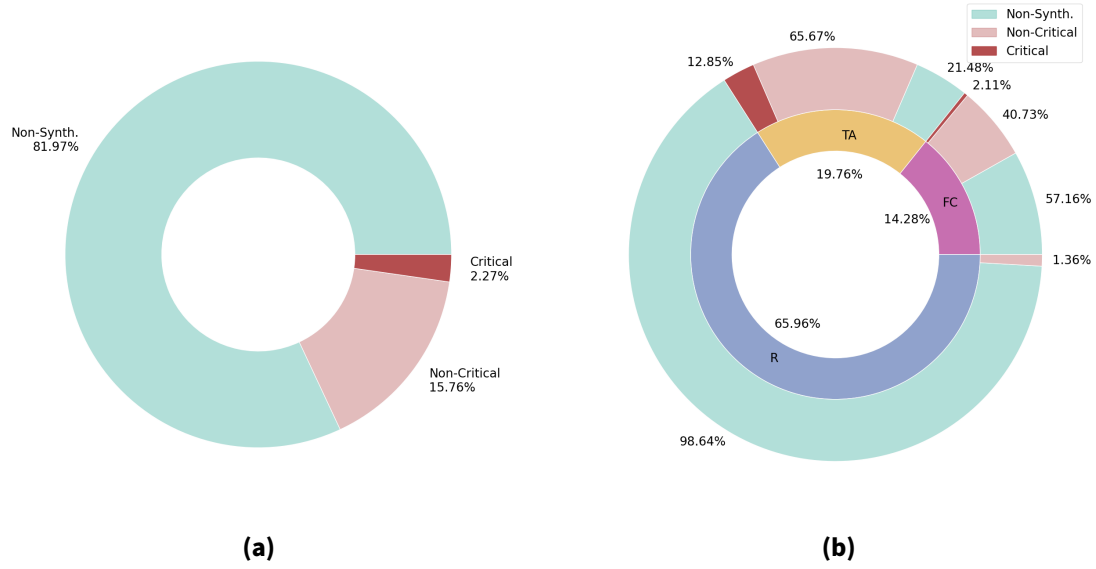


Figure 5.7.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), accross all commits for "apache/apisix".

Regarding TA references (Figure 5.9b), 37.71% were caused by C_{SL}, 2.01% by C_{Ext}, and 8.5% by C_{Arg}, while 5.87% were caused by C_{FR}, 11.59% by C_{DA}, and 34.31% by C_U.

For "apache/apisix", across all five evaluated commits, synthetic FC references (Figure 5.10a) were caused by C_{SL} in 77.65% of cases, C_{Ext} in 12.48%, and C_{Arg} in 4.94%. Additionally, 0.92% were caused by C_{FR}, 0.41% by C_{DA}, and 3.6% by C_U.
For TA references (Figure 5.10b), 35.21% were caused by C_{SL}, 9.99% by C_{Ext}, and 38.43% by C_{Arg}, while 0.65% were caused by C_{FR}, 3.29% by C_{DA}, and 12.42% by C_U.

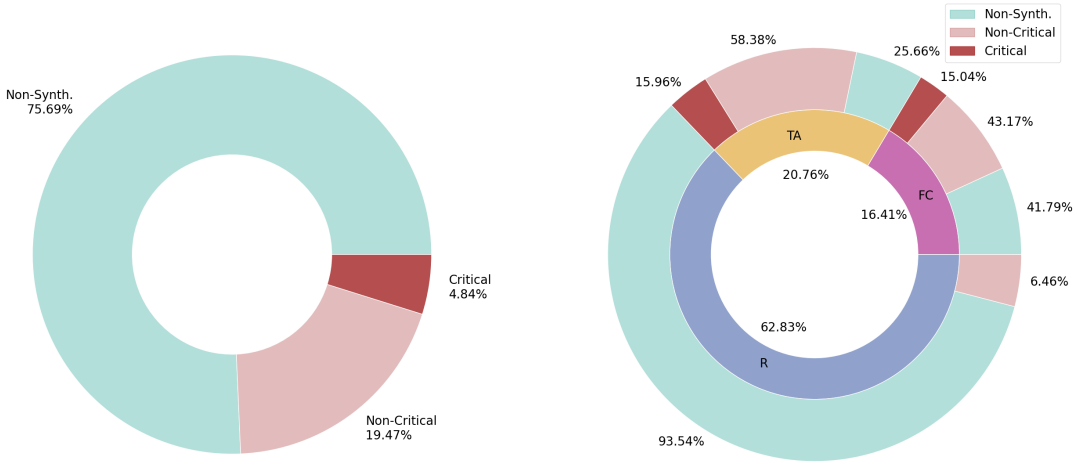


Figure 5.8.: Total Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), accross all commits for "Saghen/blink.cmp" .

Lastly, across all five commits for "Saghen/blink.cmp", synthetic FC references (Figure 5.11a) were caused by C_{SL} in 22.29% of cases, C_{Ext} in 40.67%, and C_{Arg} in 10.79%. In addition, 11.28% were caused by C_{FR}, 1.42% by C_{DA}, and 13.56% by C_U. For TA references (Figure 5.11b), 5.81% were caused by C_{SL}, 29.63% by C_{Ext}, and 43.09% by C_{Arg}, while 10.32% were caused by C_{FR}, 4.66% by C_{DA}, and 6.49% by C_U.

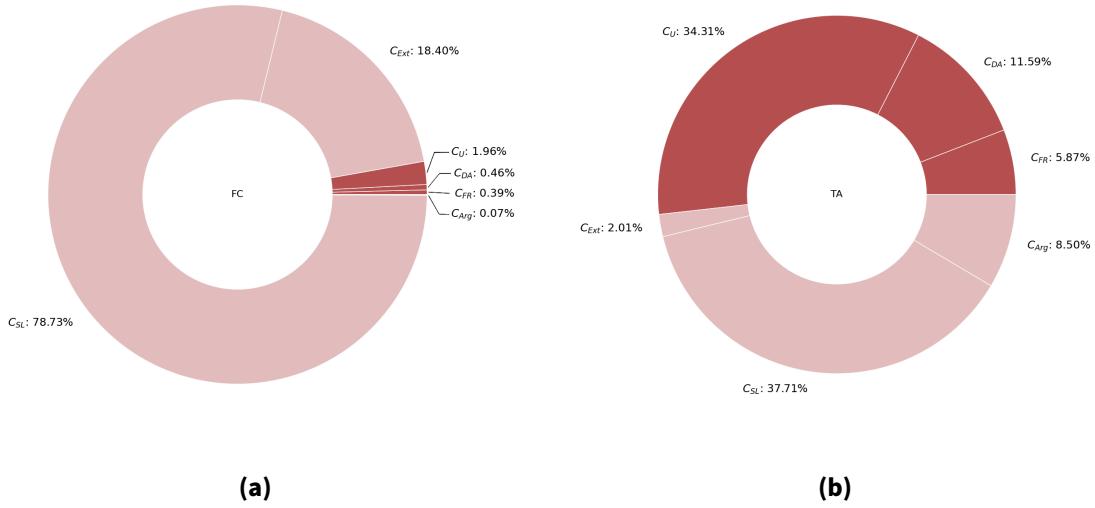


Figure 5.9.: Causes for synthetic reference resolution of Function Calls FC (a), and Table Accesses TA (b) for the Lua 5.2 test suite.

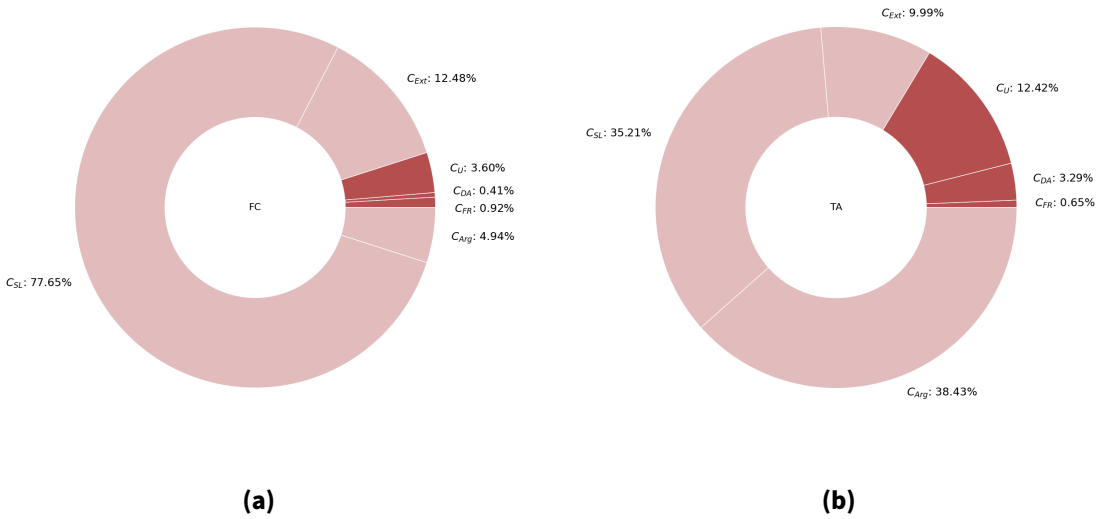


Figure 5.10.: Causes for synthetic reference resolution of Function Calls FC (a), and Table Accesses TA (b) for "apache/apisix" accross five commits.

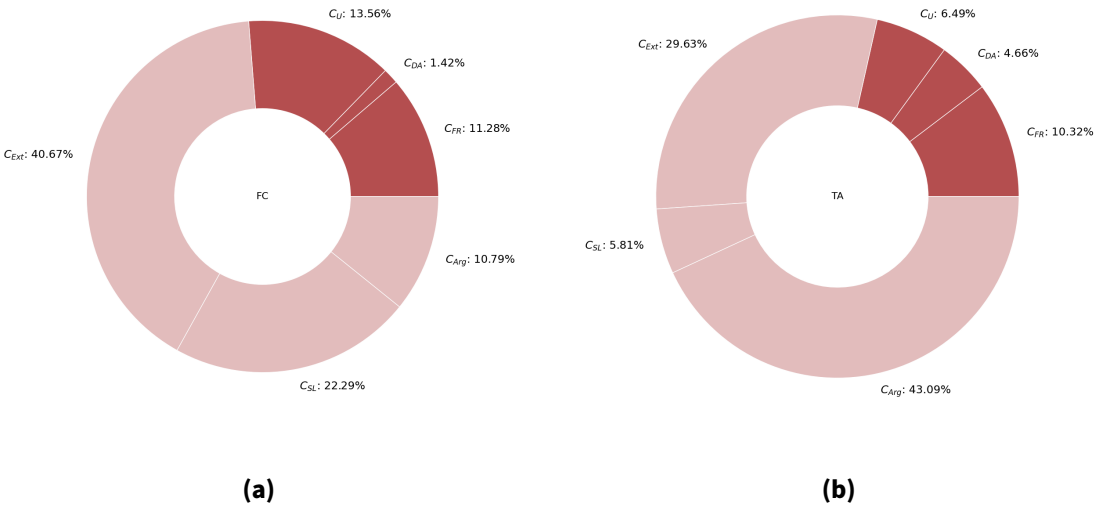


Figure 5.11.: Causes for synthetic reference resolution of Function Calls FC (a), and Table Accesses TA (b) for "Saghen/blink.cmp" accross five commits.

The final metric measured in this experiment assesses the presence of all expected components across all evaluated commits for the evaluation cases "apache/apisix" and "Saghen/blink.cmp". As shown in Table 5.18, the PCM `Repository` models generated for all commits contained the expected components. An example for the resulting `Repository` models is provided in Figure 5.12. In addition, the resulting `Repository` models show the generation of `OperationInterfaces` based on the initial technical extensions to the original Lua state-based model updates.

Project	Commit Hash	`Repository` Model contains Expected Components
apache/apisix	9731f51	Yes
	49762bc	Yes
	3d5f554	Yes
	a4b6931	Yes
	b85ebd4	Yes
Saghen/blink.cmp	a9a0f96	Yes
	a937edd	Yes
	4ef6d1e	Yes
	8620a94	Yes
	f93af0f	Yes

Table 5.18.: Results of manual comparisons of components in the generated PCM `Repository` models with the expected components (cf. Tables 5.15 and 5.16).

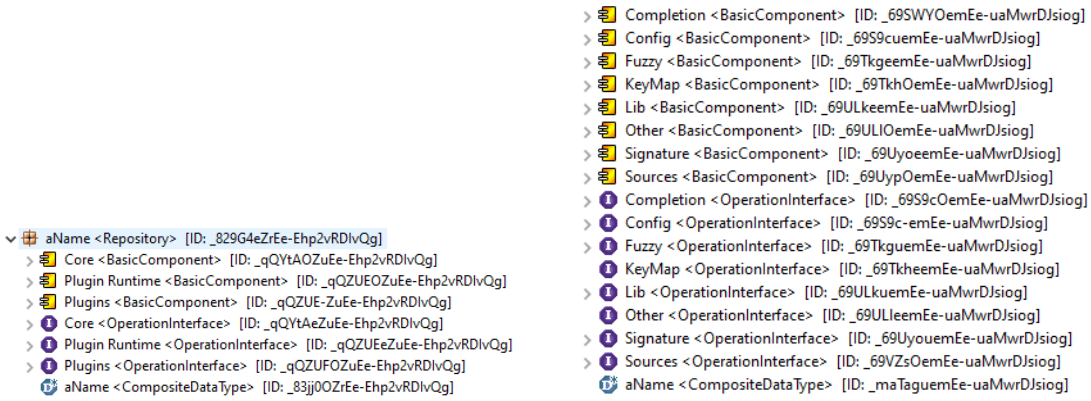


Figure 5.12.: Examples of PCM `Repository` models for the commits b85ebd4 for "apache/apisix" (a) and f93af0f for "Saghen/blink.cmp" (b).

5.4.6. $E_{CM,EC}$: Discussion

In this section, the results from experiment $E_{CM,EC}$ are discussed in relation to the questions Q2.1-Q3.2.2, and their implications for the satisfaction of **G2** and **G3**. In addition, the pending discussion of the implications resulting from the application of evaluation cases resulting from E_{EC} to these goals for G1 are discussed at the end of this section.

Ecore meta-model, generation and serialisation. The findings from $E_{CM,T}$ already suggested that the extended Lua CMoGS satisfies requirements $REQ1_{CM}$ and $REQ2_{CM}$.

These indications are further reinforced by the results from $E_{CM,EC}$, which show that the automatic generation and serialisation of Lua CMs results in a Generation-Serialisation Similarity of 1 across all evaluation cases. This confirms the satisfaction of $REQ2_{CM}$.
By extension, these findings also indicate that the provided Ecore meta-model is complete, as required by $REQ1_{CM}$[2]. Additionally, the inclusion of the Lua 5.2 test suite as an evaluation case, along with the evaluation conducted on multiple commits from real-world software projects, suggests that these requirements are met for general Lua projects.

Static reference resolution. The results show that the majority of references are resolved non-synthetically across all evaluated projects, indicating the suitability of the extended Lua CMoGS for application within the CIPM approach for Lua-based projects. However, the distributions of non-synthetic, non-critical synthetic, and critical synthetic references across different reference types suggest that various projects present distinct challenges for reference resolution.
For instance, Function Call (FC) references – the key reference type requiring resolution in the CIPM approach – is resolved non-synthetically 74.46% of cases for the Lua 5.2 test suite, 57.16% for "apache/apisix", and 41.79% of cases for the evaluation case "Saghen/blink.cmp". Additionally, the results highlight variations in the causes of synthetic reference resolution across different software projects. For example, while Function Resolution (C_{FR}) accounts for only 0.92% of synthetically resolved FC references in "apache/apisix", it is responsible for 11.28% of synthetic FC references in "Saghen/blink.cmp".
These observations lead to the hypothesis that CIPM's CI-based update of software models is influenced not only by a project's domain, the technologies it utilises, and its programming language, but also by specific coding conventions among projects that share the same programming language.

Integration into CIPM's CI-based update of software models. Unfortunately, due to the technical issues referenced in the preceding section, a thorough evaluation of the integration of the extended Lua CMoGS into CIPM's CI-based update of software models could not be conducted.
However, the results that are available do provide valuable indications regarding key aspects of the integration. Specifically, the findings suggest the effectiveness of the proposed `FileDetectionStrategy`, as it successfully identified the expected components across all evaluated software projects and commits. Furthermore, the results indicate that interfaces for the PCM Repository Model can be generated based on the detected components by leveraging non-synthetic FC reference resolution.

Implications for G2. The results demonstrate that the extended Lua CMoGS fulfils the requirements $REQ1_{CM}$-$REQ3_{CM}$ for CIPM CMoGS. The generation of CMs and their subsequent serialisation into source code for multiple real-world Lua projects produces source code that closely resembles the original indicating the satisfaction or $REQ1_{CM}$ and $REQ2_{CM}$. In addition, all references are either resolved non-synthetically or synthetically,

[2]As mentioned in Section 5.1.1, ensuring a meaningful underlying language meta-model necessitates an examination of the corresponding model. This inspection was conducted by-design. The Lua meta-model is provided along with all other artefacts pertaining to this thesis via the KIT GitLab.

confirming the satisfaction of $REQ3_{CM}$. Consequently, these findings support the overall satisfaction of G2.

Implications for G3. The initial evidence regarding the applicability of CIPM's CI-based update of software models to real-world, open-source Lua projects is promising, supporting the satisfaction of G3. The proposed extensions to the CPRs for Lua lead to the expected updates of the PCM `Repository` Model. However, further investigation is needed to fully assess the correctness of these model updates. Specifically, a more in-depth analysis of the required extensions to the existing Lua CPRs, as well as a more comprehensive evaluation of the integration of the extended Lua CMoGS into the CIPM pipeline, is necessary to establish conclusive findings.

Final implications for G1. The outcomes of the experiments for **G2** and **G3** indicate efficacy of the evaluation cases selected using the approach to Semi-Automated CIPM Evaluation Case Selection proposed in this thesis in investigations regarding the generalisability of the CIPM approach. Both **G2** and **G3** were evaluated based on these evaluation cases, thereby demonstrating the practical suitability of these software projects to CIPM generalisability research.

5.4.7. Threats to Validity

A potential threat to internal validity is that the correctness of the reference resolution mechanisms provided by the extended Lua CMoGS was evaluated exclusively on benchmark tests. While these tests provide insights into the mechanisms' functionality, they may not fully capture their accuracy in more complex real-world scenarios. Furthermore, the evaluation does not assess whether the accuracy of non-synthetically resolved references is sufficient to enable accurate performance predictions using the aPM generated by the CIPM approach. Additional validation including the integration of the Lua CMoGS into the complete CIPM pipeline and evaluation of the resulting accuracy of performance prediction for real-world software projects would be required to determine the practical impact of the resolution accuracy.

Regarding external validity, one limitation is that the scalability of the reference resolution mechanisms was not evaluated. In addition, while multiple real-world projects were analysed, the extent to which the findings generalise to Lua projects with different coding conventions, technologies, or domains remains requires further investigations. The evaluation cases employed in this thesis may not fully represent the diversity of Lua-based software, and differences in these characteristics could impact the accuracy of the reference resolution mechanisms. Further investigations on larger sets of evaluation candidates are necessary to confirm the generalisability of the results.

5.5. Summary

This section summarises the findings of the evaluation with regard to the research questions addressed in this thesis.

How can openly accessible software projects for CIPM evaluation be selected systematically? This thesis presents the approach to Semi-Automated CIPM Evaluation Case Selection as a method for systematically selecting real-world, open-source software projects as evaluation cases for investigations into the generalisability of the CIPM approach. By specifying a set of Evaluation Case Requirements for CIPM and introducing a process to reduce the manual effort involved in generating evaluation case sets, this approach facilitates the identification of openly accessible projects suitable for assessing the generalisability of the CIPM approach.

Can the CIPM approach be applied to dynamic programming languages? This question was investigated by examining the generalisability of CIPM's CM generation and serialisation using Lua as a representative dynamically typed programming language. The identification and analysis of the requirements for CIPM CMoGS in this thesis revealed challenges relating to the satisfaction of these requirements posed by dynamically typed programming languages.
On the one hand, the application of frameworks such as Xtext for the definition of a Lua meta-model is particularly challenging in the case of dynamically typed programming languages, as evidenced by the limitations of the original Lua CMoGS. However, these challenges mainly pertain to the technical complexities in specifying an Xtext grammar capable of representing the complete Lua language, for instance due to left-recursive nature of this grammar, and are resolved by the extensions to the Lua CMoGS performed in this thesis.
On the other hand, the static reference resolution mechanisms required for CIPM CMoGS face particular challenges in the case of dynamically typed languages. Unlike statically typed languages, where type information can be leveraged for reference resolution, dynamically typed languages lack such information at compile time. This makes static resolution inherently approximate and necessitates fallback mechanisms, such as trivial recovery, to prevent unresolved references in the resulting CMs.
Despite these challenges, the evaluation of the extended Lua CMoGS demonstrates that the majority of references can still be resolved non-synthetically across multiple real-world, open-source Lua projects, suggesting the successful applicability of the CIPM approach to dynamically typed programming languages.

In conclusion, the findings of this thesis provide promising evidence for the generalisability of the CIPM approach to other programming languages. Although dynamically typed languages introduce specific challenges to the generation and serialisation of CMs, the results demonstrate that these challenges can be addressed. However, the complexities and effort-intensive nature of designing CIPM CMoGS for new programming languages poses a challenge for the application of CIPM to a broad range of programming languages. Moreover, while this thesis addressed the generalisability of CMoGS for CIPM, further investigation

is needed to assess the applicability of the complete CIPM pipeline to dynamically typed languages such as Lua.

6. Related Work

This section provides insights into the literature related to this thesis. Section 6.1 provides an overview of MSR-based tools to generate datasets consisting of software projects, followed by related work regarding static reference resolution in dynamically typed programming languages in Section 6.2 .

6.1. Mining Software Repositories

There is a vast body of research spanning the field of MSR, including various surveys analysing and categorising MSR approaches [49, 81, 11, 95]. In order to provide a concise impression of the related work, this section presents a brief selection of MSR-based tools related to the Approach to Semi-Automated CIPM Evaluation Case Selection developed in this thesis. The interested reader is directed to the aforementioned surveys, in which further MSR approaches are discussed.

One example of an MSR-based tool is *BUDGET*, developed by Santos et al. [79] to automate the creation of training data for supervised learning models in software architecture traceability research. It utilises automated web mining and big data analysis techniques to extract datasets from software repositories. Additionally, it allows users to configure various filters, such as restricting analyses to specific programming languages, selecting relevant technical libraries for web-based mining, and specifying repositories for dataset generation. The evaluation conducted in the corresponding study demonstrated BUDGET's ability to generate good quality initial datasets. While originally developed for research in software system traceability, the authors suggest that BUDGET can also be leveraged for mining software repositories and architectures. Although the tool was initially made publicly available online, it no longer appears to be accessible.

Bajracharya, Ossher, and Lopes [9, 8] designed *Sourcerer* as an infrastructure for the large-scale collection, indexing and analysis of Java source code. Sourcerer collects Java projects from sources such as version control systems, public web sites and open-source repositories. The source code of these projects is parsed, analysed, and then stored in three abstractions: (1) A Managed Repository containing an original copy of the source code, (2) a database, SourcererDB, containing relational models of the project's code, and (3) a Serach Index to enable keyword searches in the Managed Repository and SourcererDB. Leveraging these abstractions, Sourcerer provides services such as keyword analyses and structural analysis, for instance through cross-project dependency information and entity dependency slicing (i.e., retrieving a collection of entities a given entity depends on). Similarly to BUDGET,

Sourcerer was originally publicly available online, but does not seem to be accessible any more.

To facilitate large-scale mining of software repositories, Dyer et al. introduced *Boa* [19], a DSL and corresponding infrastructure designed to simplify the analysis of ultra-large-scale software repositories. Boa provides researchers with a high-level language tailored for MSR tasks, abstracting the complexities associated with the implementation of data retrieval and processing. It offers predefined functions for common mining goals, while also allowing for the definition of user-defined queries. Boa's infrastructure leverages distributed computing to efficiently process extensive datasets, enabling large-scale repository analysis. The evaluation conducted in this work demonstrates the applicability of Boa to the analysis of software repositories based on programming languages, project management practices and legal aspects, while reducing implementation effort and maintaining efficiency. The authors provide a web-based interface to Boa's infrastructure, making it accessible for researchers interested in analysing large-scale software repositories.

While these approaches facilitate software repository analyses, they are not specifically aimed at generating a data basis for CIPM generalisability research. In contrast, the approach to Semi-Automated CIPM Evaluation Case Selection proposed in this thesis systematically assesses software projects based on nine identified requirements, ensuring that selected projects are directly relevant to the empirical validation of the generalisability of the CIPM approach.

6.2. Static Reference Resolution

This section presents research related to the challenges posed by dynamically typed programming languages for static analysis, particularly for static reference resolution as employed by the extended Lua CMoGS developed in this thesis.

Madsen's PhD thesis [59] presents several papers addressing different challenges in the static analysis of dynamically typed languages. In the paper "Sparse Dataflow Analysis with Pointers and Reachability" [61, 59], Madsen and Møller present an approach for sparse data flow analysis in JavaScript. Their approach identifies def-use edges in a Control Flow Graph (CFG), which link binding occurrences (referred to as "definition sites") to applied occurrences (referred to as "use sites"). They address the challenges of static reference resolution in dynamically typed programming languages by leveraging sparse dataflow analysis on CFGs in static single assignment (SSA) form. The utilisation of sparse data flow analysis is aimed at reducing redundant computations by propagating dataflow information only where necessary, instead of across every program statement. SSA represents the CFG in a form where every applied occurrence has exactly one dominating binding occurrence in the graph.
To handle the challenge that constructing SSA requires def-use information, Madsen and Møller employ an on-the-fly computation strategy that incrementally builds the SSA representation while performing dataflow analysis. This approach eliminates the need for a

separate pre-analysis of def-use chains, improving efficiency. Their evaluation on benchmark JavaScript programs demonstrates the performance benefits of their approach compared to traditional dense dataflow analysis.

In another paper presented in Madsen's PhD thesis [59], "String Analysis for Dynamic Field Access" [60, 59], Madsen and Andreasen address the challenges posed by dynamic property accesses in JavaScript. The authors introduce seven novel string abstractions to perform string analysis on property accesses. They also discuss five existing abstractions and design a hybrid string abstraction denoted as $\mathcal{H}$. To evaluate their approach, they integrate $\mathcal{H}$ into a dataflow analysis of points-to relations and compare it to an analysis using string literals. Their results indicate significant improvements in the precision of point-to relations and performance when applying $\mathcal{H}$, compared to dynamic field resolution constrained to string literals.

While these approaches address static analysis in dynamically typed programming languages, they do not consider the specifics of the Lua language. Furthermore, they do not explore the challenges and applicability of static reference resolution for automated model updates within the CIPM framework, as this thesis does.

7. Future Work

This chapter explores potential areas for future work, building on the findings of this thesis, with a focus on further investigations into the generalisability of the CIPM approach.

7.1. Generating a Data Basis for CIPM Research

While the proposed Approach to Semi-Automated CIPM Evaluation Case Selection has demonstrated efficiency in generating evaluation case sets for assessing the generalisability of the CIPM approach, several avenues for future research remain.

Broadening Evaluation Criteria. The current Set of Evaluation Case Requirements could be expanded to encompass a wider array of research objectives. Conducting evaluations on more diverse sets of initial candidates and aligning them with varied research goals would enhance the robustness and applicability of the approach.

Integrating Advanced MSR Tools. Exploring the application of more sophisticated Mining Software Repositories (MSR) tools, such as Boa and Sourcerer, could further automate and enhance the dataset generation process for CIPM-related research. Investigating how these tools can be leveraged to compile comprehensive collections of software projects tailored to various CIPM research objectives would be a valuable extension of this work.

By pursuing these directions, future research can build upon the foundation laid in this thesis, contributing to a more comprehensive and versatile framework for the generation of datasets for the evaluation of the CIPM approach.

7.2. Towards Generalised Consistency Preservation Rules

The findings regarding the extensions to the Lua CPRs in this thesis indicate that components and interfaces for the PCM Repository Model can be extracted using the `FileDetectionStrategy` and the reference resolution performed by the Lua CMoGS. Building upon these findings, future work could explore the generalisability of CPRs through the following steps:

1. *Analysing the applicability of existing architectural reverse engineering approaches*: Investigate the applicability of methods such as those recently discussed by Qayum et al. [72] to determine mappings from source code files to architectural components for the CIPM approach. These mappings could inform the `FileDetectionStrategy`,

facilitating the reconstruction of corresponding components within the CI-based update of software models.

2. *Extending Lua CPRs for PCM Repository Model updates*: Enhance existing Lua CPRs to include the reconstruction of RDSEFFs by leveraging information provided by the reference resolution mechanism of the extended Lua CMoGS.
3. *Evaluating the complete CI-based update of software models*: Assess the effectiveness of the extended CI-based update process, incorporating the extensions for CPR generalisability from the previous steps.

Such investigations could reduce the dependency of CIPM's CPRs on a project's domain and employed technologies. By generalising the component detection strategy and PCM `Repository` Model updates based on architectural reverse engineering approaches and static reference resolution, it may be possible to eliminate the need for defining specific CPRs for individual software projects.

8. Conclusion

The Continuous Integration of architectural Performance Models (CIPM) approach addresses the challenge of integrating automated architectural Performance Model (aPM) updates into CI/CD pipelines, enabling efficient and accurate Architecture-based Performance Predictions (AbPP). While CIPM has been primarily evaluated on Java-based projects, prior research has provided initial insights into its generalisability to the Lua programming language.

A key prerequisite for investigating the generalisability of CIPM is establishing a suitable data basis in the form of software projects for evaluation. To address this, this thesis introduces the approach to Semi-Automated CIPM Evaluation Case Selection, aimed at reducing the manual effort required to identify suitable evaluation cases.
In addition, this thesis builds upon previous work by further investigating the generalisability of the CIPM approach to dynamically typed programming languages, using Lua as a representative case. The analysis of the challenges posed by dynamically typed languages for the Code Model (CM) generation and serialisation in CIPM revealed limitations in the existing Lua Code Model Generator and Serialiser (CMoGS). These limitations were addressed through targeted extensions developed in this work.

The evaluation results indicate that the approach to Semi-Automated CIPM Evaluation Case Selection effectively generates a data basis for assessing the generalisability of CIPM, significantly reducing the required manual effort. Additionally, findings suggest that the extended Lua CMoGS satisfies the requirements for CIPM CMoGS in general Lua projects.

In conclusion, this thesis provides foundational insights into the generalisability of CIPM to dynamically typed programming languages. While challenges remain, for instance concerning the generalisability of Consistency Preservation Rules (CPRs) employed in the CIPM approach, the findings contribute to expanding the applicability of the CIPM approach across various programming languages.

Bibliography

[1] Wasim Alsaqaf, Maya Daneva, and Roel Wieringa. "Quality requirements challenges in the context of large-scale distributed agile: An empirical study". In: *Information and software technology* 110 (2019), pp. 39–55.

[2] Suzana Andova et al. "MDE Basics with a DSL Focus". In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2012, pp. 21–57.

[3] Hendrik van Antwerpen et al. "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–30.

[4] Martin Armbruster. "Commit-Based Continuous Integration of Performance Models". MA thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2021.

[5] Martin Armbruster. *Parsing and Printing Java 7-15 by Extending an Existing Metamodel*. Tech. rep. Tech. rep. July 28, 2022.

[6] Martin Armbruster, Manar Mazkatli, and Anne Koziolek. "Recovering Missing Dependencies in Java Models". In: (2023).

[7] Felix Bachmann et al. "Designing software architectures to achieve quality attribute requirements". In: *IEE Proceedings-Software* 152.4 (2005), pp. 153–165.

[8] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code". In: *Science of Computer Programming* 79 (2014), pp. 241–259.

[9] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. "Sourcerer: An internet-scale software repository". In: *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE. 2009, pp. 1–4.

[10] Simonetta Balsamo et al. "Model-based performance prediction in software development: A survey". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310.

[11] Gabriele Bavota. "Mining unstructured data in software repositories: Current and future trends". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. IEEE. 2016, pp. 1–12.

[12] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio Component Model for Model-driven Performance Prediction". In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066.

[13] Floris van Beers et al. “Deep neural networks with intersection over union loss for binary image segmentation”. In: *Proceedings of the 8th international conference on pattern recognition applications and methods*. SciTePress. 2019, pp. 438–445.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[15] Lukas Burgey. “Continuous Integration of Performance Models for Lua-Based Sensor Applications”. MA thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2023.

[16] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[17] Andrei Costin. “Lua code: security overview and practical approaches to static analysis”. In: *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2017, pp. 132–142.

[18] Liliana Dobrica and Eila Niemela. “A survey on software architecture analysis methods”. In: *IEEE Transactions on software Engineering* 28.7 (2002), pp. 638–653.

[19] Robert Dyer et al. “Boa: Ultra-large-scale software repository and source-code mining”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25.1 (2015), pp. 1–34.

[20] Sven Efftinge and Markus Völter. “oAW xText: A framework for textual DSLs”. In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32. 118. 2006.

[21] Björn Engelmann, Ernst-Rüdiger Olderog, and Frank S De Boer. “Techniques for the verification of dynamically typed programs”. PhD thesis. Carl von Ossietzky Universität Oldenburg, 2017.

[22] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 2010, pp. 307–309.

[23] Alexander Ferrein and Gerald Steinbauer. “On the way to high-level programming for resource-limited embedded systems with GOLOG”. In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer. 2010, pp. 229–240.

[24] Andrew Forward and Timothy C Lethbridge. “Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals”. In: *Proceedings of the 2008 international workshop on Models in software engineering*. 2008, pp. 27–32.

[25] The Apache Software Foundation. *Apache Apisix – Cloud-Native API Gateway*. 2024. URL: https://apisix.apache.org/ (visited on 06/24/2024).

[26] The Apache Software Foundation. *apache/apisix at 5a085bcd0719b7e2817c54e6f2fe5418d6efb164*. 2024. URL: https://github.com/apache/apisix/tree/5a085bcd0719b7e2817c54e6f2fe5418d6efb164 (visited on 01/31/2025).

[27] The Apache Software Foundation. *GitHub - apache/apisix: The Cloud-Native API Gateway*. 2024. URL: https://github.com/apache/apisix (visited on 07/02/2024).

[28] The Eclipse Foundation. *Eclipse Modeling Project | The Eclipse Foundation*. 2024. URL: https://eclipse.dev/modeling/emf/ (visited on 07/23/2024).

[29] The Eclipse Foundation. *EMF Compare | Home*. 2024. URL: https://eclipse.dev/emf/compare/ (visited on 07/23/2024).

[30] The Eclipse Foundation. *Xtext - Integration with EMF*. 2024. URL: https://eclipse.dev/Xtext/documentation/308_emf_integration.html (visited on 07/24/2024).

[31] The Eclipse Foundation. *Xtext - Language Implementation*. 2024. URL: https://eclipse.dev/Xtext/documentation/303_runtime_concepts.html (visited on 07/01/2025).

[32] The Eclipse Foundation. *Xtext - The Grammar Language*. 2024. URL: https://eclipse.dev/Xtext/documentation/301_grammarlanguage.html (visited on 07/01/2025).

[33] *GitHub*. 2024. URL: https://github.com/ (visited on 01/13/2025).

[34] *GitHub REST API documentation - GitHub Docs*. 2024. URL: https://docs.github.com/en/rest (visited on 01/13/2025).

[35] *Github-Ranking/Top100/Java.md at master · EvanLi/Github-Ranking*. 2024. URL: https://github.com/EvanLi/Github-Ranking/blob/master/Top100/Java.md (visited on 01/27/2025).

[36] *Github-Ranking/Top100/Lua.md at master · EvanLi/Github-Ranking*. 2024. URL: https://github.com/EvanLi/Github-Ranking/blob/master/Top100/Lua.md (visited on 01/14/2025).

[37] Martin Glinz. "A glossary of requirements engineering terminology". In: *Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version* 1 (2011), p. 56.

[38] Martin Glinz. "On non-functional requirements". In: *15th IEEE international requirements engineering conference (RE 2007)*. IEEE. 2007, pp. 21–26.

[39] Object Management Group. *OMG | Object Management Group*. 2024. URL: https://www.omg.org/index.htm (visited on 07/23/2024).

[40] Maria Haigh. "Software quality, non-functional software requirements and IT-business alignment". In: *Software Quality Journal* 18 (2010), pp. 361–385.

[41] Ahmed E Hassan. "The road ahead for mining software repositories". In: *2008 frontiers of software maintenance*. IEEE. 2008, pp. 48–57.

[42] Ahmed E Hassan, Richard C Holt, and Audris Mockus. "MSR 2004: International workshop on mining software repositories". In: *Proceedings. 26th International Conference on Software Engineering*. IEEE Computer Society. 2004, pp. 770–771.

[43] Florian Heidenreich et al. "Closing the gap between modelling and java". In: *Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers 2*. Springer. 2010, pp. 374–383.

[44] Florian Heidenreich et al. “Derivation and refinement of textual syntax for models”. In: *Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings 5*. Springer. 2009, pp. 114–129.

[45] Florian Heidenreich et al. *JaMoPP: The Java Model Parser and Printer*. Tech. rep. 2009.

[46] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. “The evolution of Lua”. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 2–1.

[47] Woosung Jung, Eunjoo Lee, and Chisu Wu. “A survey on mining software repositories”. In: *IEICE TRANSACTIONS on Information and Systems* 95.5 (2012), pp. 1384–1406.

[48] *JUnit 5*. 2024. URL: https://junit.org/junit5/ (visited on 01/13/2025).

[49] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. “A survey and taxonomy of approaches for mining software repositories in the context of software evolution”. In: *Journal of software maintenance and evolution: Research and practice* 19.2 (2007), pp. 77–131.

[50] Heiko Klare et al. “Enabling consistency in view-based system development—the vitruvius approach”. In: *Journal of Systems and Software* 171 (2021), p. 110815.

[51] Heiko Koziolek, Steffen Becker, and Jens Happe. “Predicting the Performance of Component-Based Software Architectures with Different Usage Profiles”. In: vol. 4880. July 2007, pp. 145–163. ISBN: 978-3-540-77617-8. DOI: 10.1007/978-3-540-77619-2_9.

[52] Max E Kramer, Erik Burger, and Michael Langhammer. “View-centric engineering with synchronized heterogeneous models”. In: *Proceedings of the 1st workshop on view-based, aspect-oriented and orthographic software modelling*. 2013, pp. 1–6.

[53] William Landi. “Undecidability of static analysis”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

[54] Michael Langhammer. “Automated Coevolution of Source Code and Software Architecture Models”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 259 pp. DOI: 10.5445/IR/1000069366.

[55] *Licensing a repository - GitHub Docs*. 2024. URL: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository (visited on 01/14/2025).

[56] *Lua: test suites*. 2024. URL: https://www.lua.org/tests/ (visited on 08/09/2024).

[57] *Lua: version history*. 2024. URL: https://www.lua.org/versions.html (visited on 01/20/2025).

[58] *LuaLS/lua-language-server: A language server that offers Lua language support - programmed in Lua*. 2024. URL: https://github.com/LuaLS/lua-language-server (visited on 01/08/2025).

[59] Magnus Madsen. “Static analysis of dynamic languages”. PhD thesis. 2015.

[60] Magnus Madsen and Esben Andreasen. “String analysis for dynamic field access”. In: *International Conference on Compiler Construction*. Springer. 2014, pp. 197–217.

[61] Magnus Madsen and Anders Møller. “Sparse dataflow analysis with pointers and reachability”. In: *International Static Analysis Symposium*. Springer. 2014, pp. 201–218.

[62] Manar Mazkatli, Martin Armbruster, and Anne Koziolek. “Towards Continuous Integration of Performance Models for Lua-Based Sensor Applications”. In: (2023).

[63] Manar Mazkatli et al. “Continuous Integration of Architectural Performance Models with Parametric Dependencies–The CIPM Approach”. In: *Available at SSRN 4232759* (2022).

[64] Manar Mazkatli et al. “Incremental calibration of architectural performance models with parametric dependencies”. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2020, pp. 23–34.

[65] Melange. *The Melange Language Workbench - Melange*. 2024. URL: http://melange.inria.fr/ (visited on 07/16/2024).

[66] modelpractice. *General Model Theory by Stachowiak | modelpractice*. 2024. URL: https://modelpractice.wordpress.com/2012/07/04/model-stachowiak/ (visited on 08/08/2024).

[67] David Monschein. “Enabling Consistency between Software Artefacts for Software Adaption and Evolution”. MA thesis. Karlsruher Institut für Technologie (KIT), 2020. DOI: 10.5445/IR/1000166031.

[68] Pierre Néron et al. “A theory of name resolution”. In: *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer. 2015, pp. 205–231.

[69] *ntop/ntopng at bc373e778687e03c3702433d41a15440a9db3dd7*. 2024. URL: https://github.com/ntop/ntopng/tree/bc373e778687e03c3702433d41a15440a9db3dd7 (visited on 01/31/2025).

[70] Terence Parr and Kathleen Fisher. “LL (*) the foundation of the ANTLR parser generator”. In: *ACM Sigplan Notices* 46.6 (2011), pp. 425–436.

[71] PUC-Rio. *Lua 5.2 Reference Manual*. 2024. URL: https://www.lua.org/manual/5.2/manual.html (visited on 12/28/2024).

[72] Abdul Qayum et al. “A Framework and Taxonomy for Characterizing the Applicability of Software Architecture Recovery Approaches: A Tertiary-Mapping Study”. In: *Software: Practice and Experience* 55.1 (2025), pp. 100–132.

[73] Celina Ramjoué. “Towards open science: The vision of the European Commission”. In: *Information Services & Use* 35.3 (2015), pp. 167–170.

[74] Ralf H Reussner, Heinz W Schmidt, and Iman H Poernomo. “Reliability prediction for component-based software architectures”. In: *Journal of systems and software* 66.3 (2003), pp. 241–252.

[75] Ralf H Reussner et al. *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.

[76] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[77] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14 (2009), pp. 131–164.

[78] *Saghen/blink.cmp at f2710804fd17211c18468100367ce2d3a230ed2e*. 2024. URL: https://github.com/Saghen/blink.cmp/tree/f2710804fd17211c18468100367ce2d3a230ed2e (visited on 01/31/2025).

[79] Joanna CS Santos et al. "BUDGET: A tool for supporting software architecture traceability research". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE. 2016, pp. 303–306.

[80] Michael Scott. *Programming language pragmatics*. Morgan Kaufmann, 2009.

[81] Tamanna Siddiqui and Ausaf Ahmad. "Data mining tools and techniques for mining software repositories: A systematic review". In: *Big Data Analytics: Proceedings of CSI 2015* (2018), pp. 717–726.

[82] KIT chair of Software Design and Quality (SDQ). *CIPM - SDQ Wiki*. 2024. URL: https://sdq.kastel.kit.edu/wiki/CIPM (visited on 07/02/2024).

[83] KIT chair of Software Design and Quality (SDQ). *Goal Question Metric - SDQ Wiki*. 2024. URL: https://sdq.kastel.kit.edu/wiki/Goal_Question_Metric (visited on 07/04/2024).

[84] KIT chair of Software Design and Quality (SDQ). *PRICoBE - SDQ Wiki*. 2024. URL: https://sdq.kastel.kit.edu/wiki/PRICoBE (visited on 07/02/2024).

[85] Mallku Soldevila, Beta Ziliani, and Bruno Silvestre. "From specification to testing: semantics engineering for Lua 5.2". In: *Journal of Automated Reasoning* 66.4 (2022), pp. 905–952.

[86] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[87] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[88] Richard Berntsson Svensson et al. "Quality requirements in industrial practice—an extended interview study at eleven companies". In: *IEEE transactions on software engineering* 38.4 (2011), pp. 923–935.

[89] Neovim Team. *Neovim*. 2025. URL: https://neovim.io/ (visited on 02/16/2025).

[90] Dharmesh Thakkar et al. "A framework for measurement based performance modeling". In: *Proceedings of the 7th International Workshop on Software and Performance*. 2008, pp. 55–66.

[91] *The most-comprehensive AI-powered DevSecOps platform*. 2024. URL: https://about.gitlab.com/ (visited on 01/13/2025).

[92] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. “Kieker: A framework for application performance monitoring and dynamic software analysis”. In: *Proceedings of the 3rd ACM/SPEC international conference on performance engineering*. 2012, pp. 247–248.

[93] SICK Vertriebs-GmbH. *SICK AppSpace · GitLab*. 2024. URL: https://gitlab.com/sick-appspace (visited on 01/02/2025).

[94] SICK Vertriebs-GmbH. *SICK AppSpace developers | SICK*. 2024. URL: https://www.sick.com/de/de/sick-appspace-developers/s/sas-developers (visited on 01/02/2025).

[95] Melina Vidoni. “A systematic process for Mining Software Repositories: Results from a systematic literature review”. In: *Information and Software Technology* 144 (2022), p. 106791.

[96] Markus Völter et al. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[97] Fanjun Weng. “Generalization of Consistency Rules between Architectural and Code Models”. Master’s Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2025.

[98] Mark D Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific data* 3.1 (2016), pp. 1–9.

[99] *xmake-io/xmake at 1c95d287f8bf7b80afc30e68ceb7f438afcccb7a*. 2024. URL: https://github.com/xmake-io/xmake/tree/1c95d287f8bf7b80afc30e68ceb7f438afcccb7a (visited on 01/31/2025).

A. Appendix

A.1. Reference Resolution Results for Single Commits

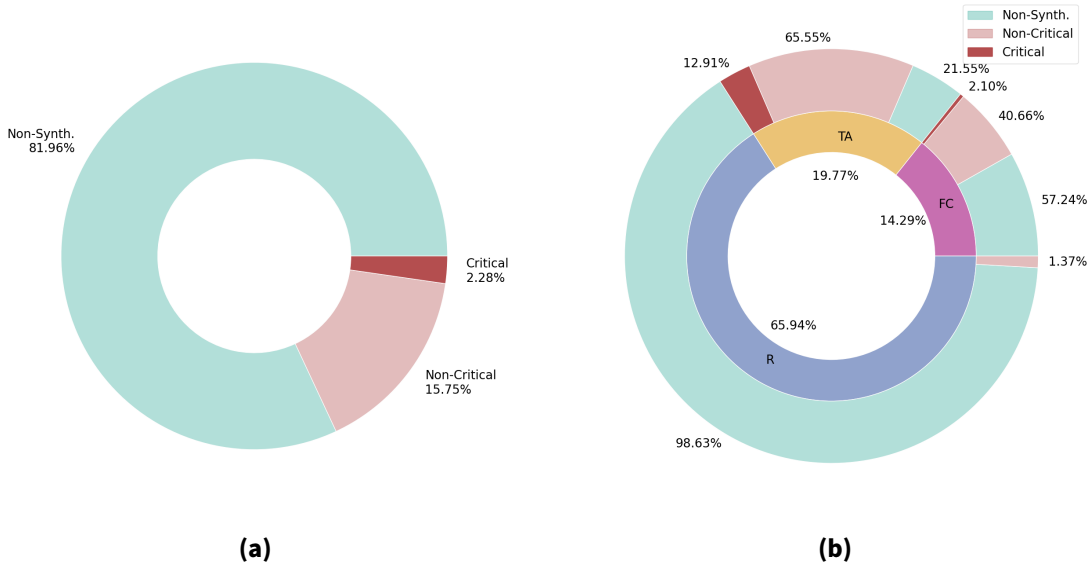
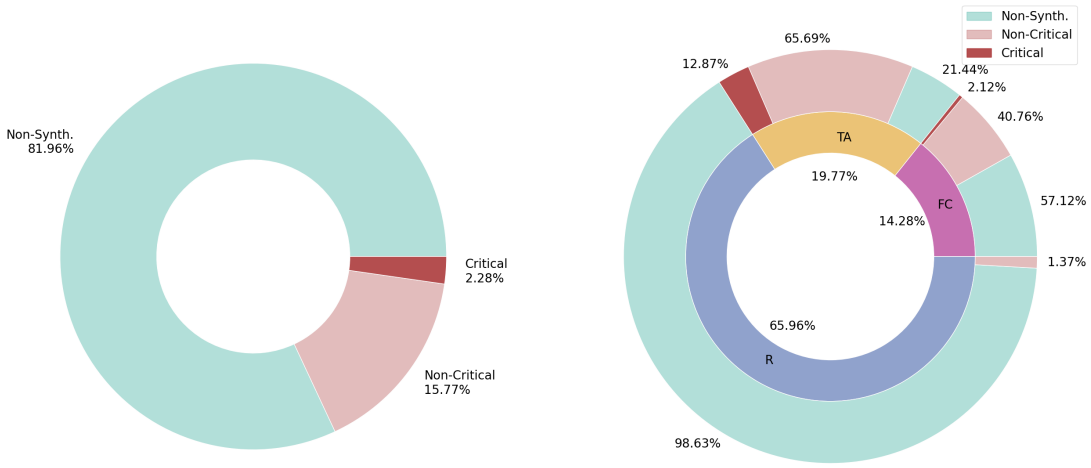
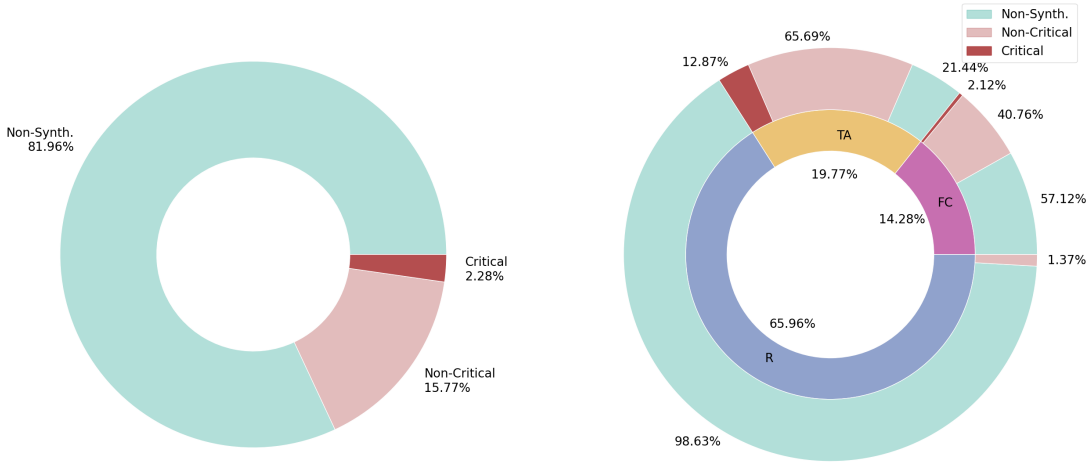


Figure A.1.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "apache/apisix" commit 9731f51.



(a) **(b)**

Figure A.2.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "apache/apisix" commit 49762bc.



(a) **(b)**

Figure A.3.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "apache/apisix" commit 3d5f554.

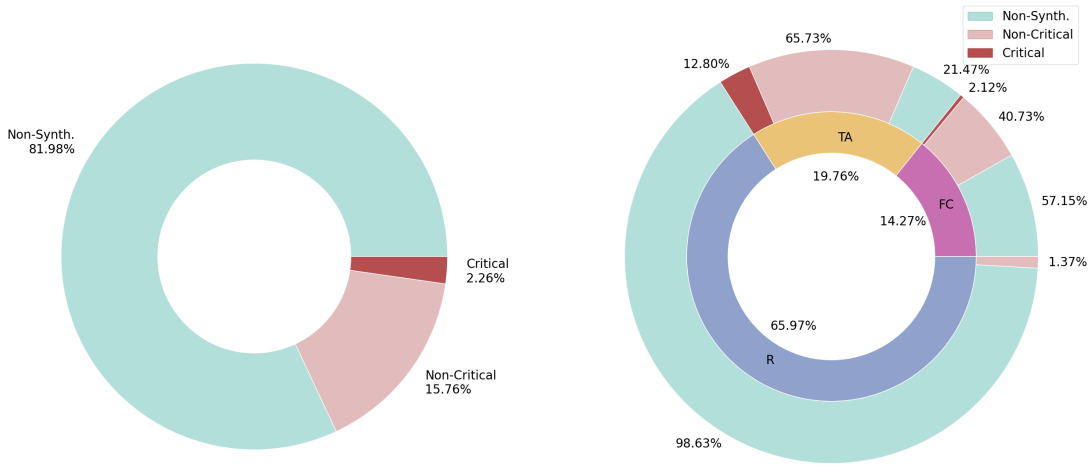


Figure A.4.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "apache/apisix" commit a4b6931.

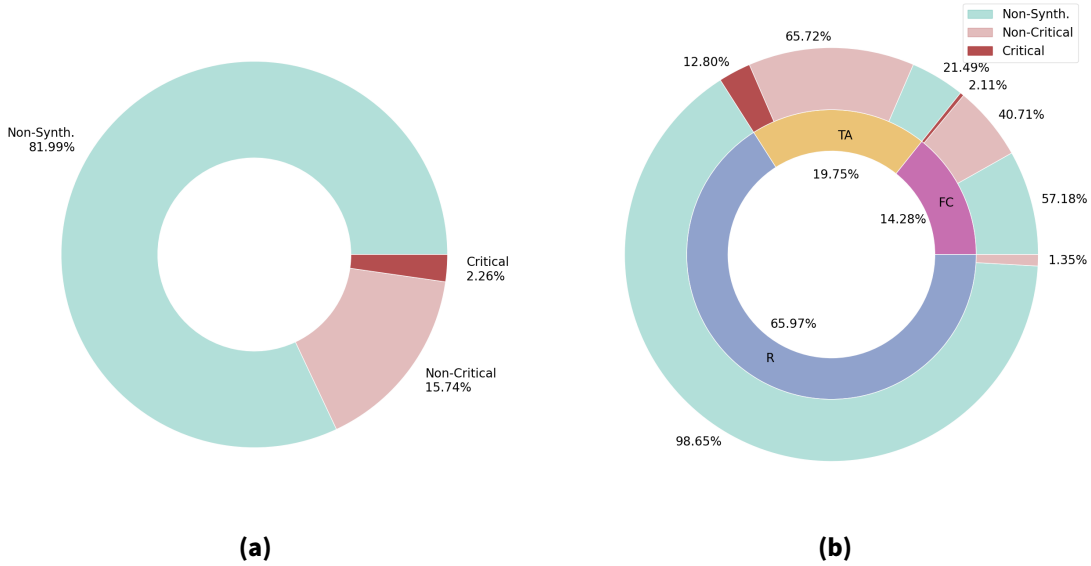
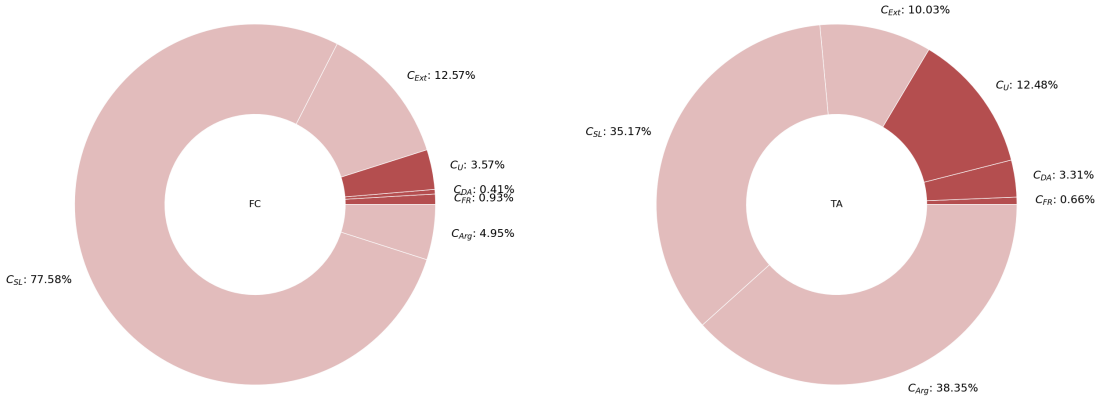
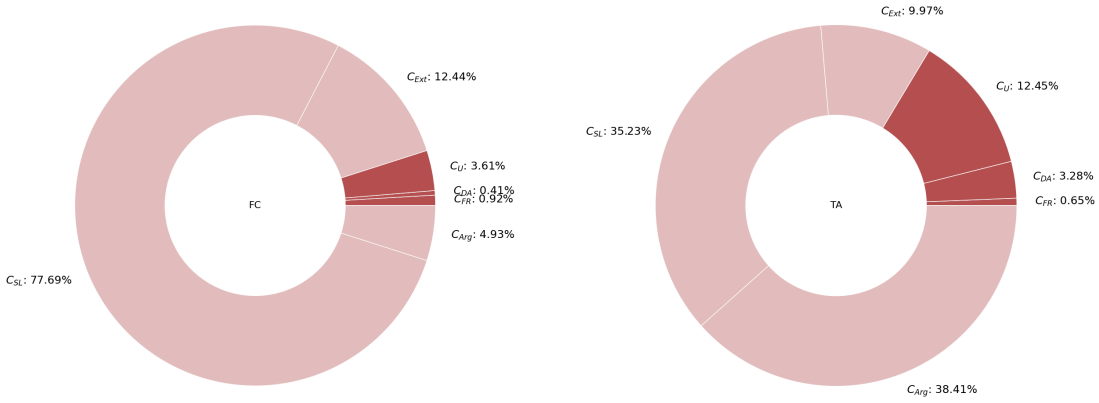


Figure A.5.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "apache/apisix" commit b85ebd4.



(a) **(b)**

Figure A.6.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "apache/apisix" commit 9731f51.



(a) **(b)**

Figure A.7.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "apache/apisix" commit 49762bc.

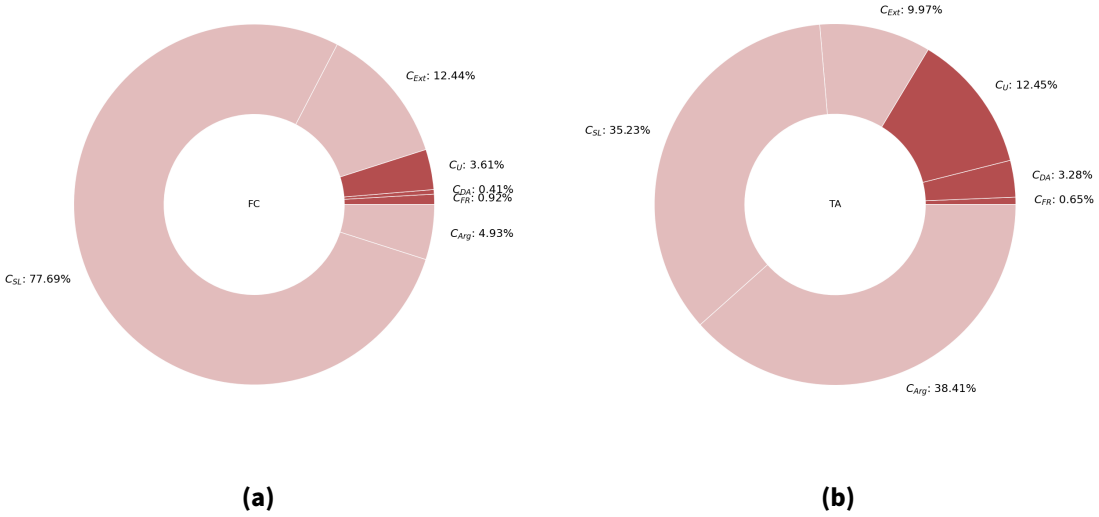


Figure A.8.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "apache/apisix" commit 3d5f554.

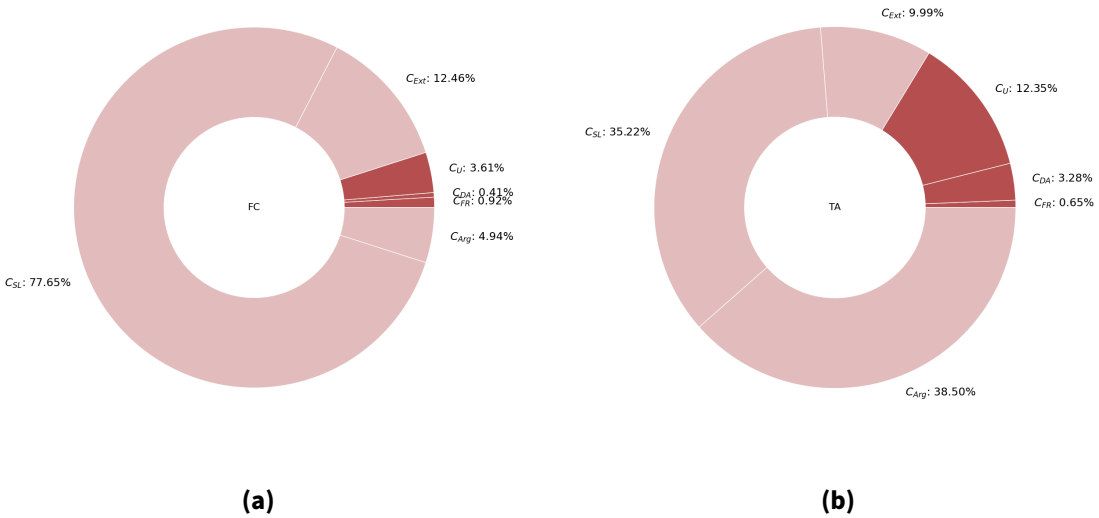


Figure A.9.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "apache/apisix" commit a4b6931.

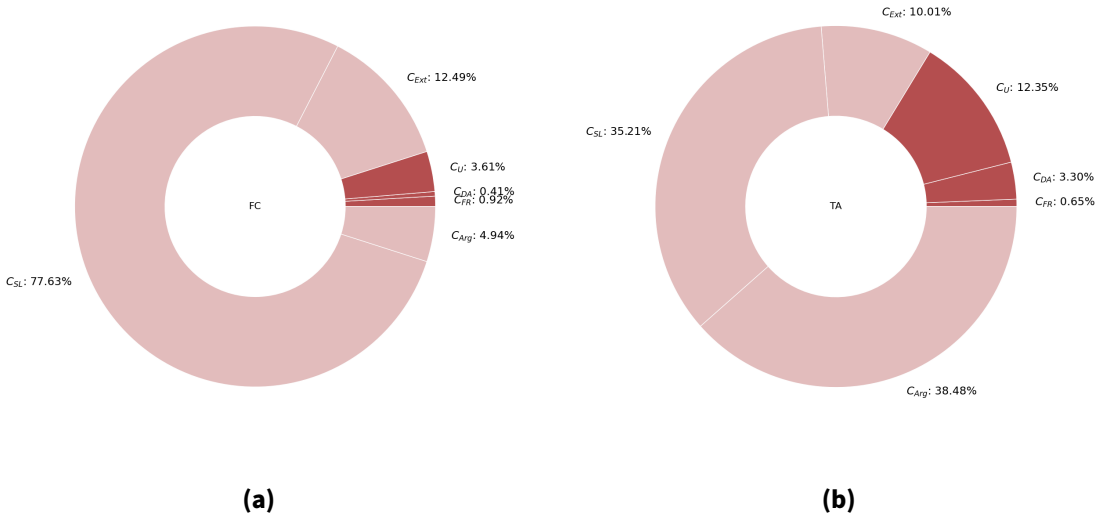


Figure A.10.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "apache/apisix" commit b85ebd4.

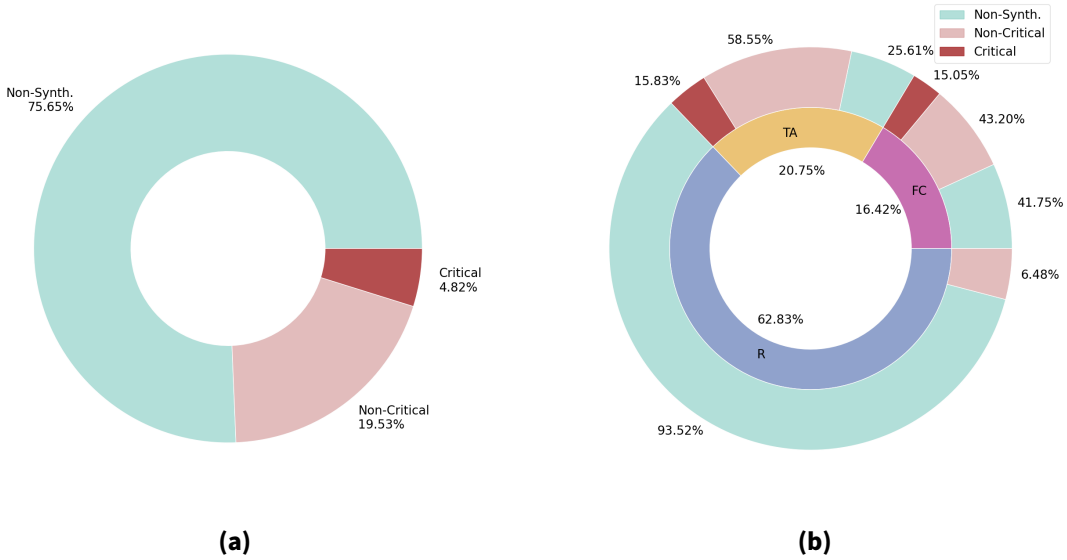


Figure A.11.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "Saghen/blink.cmp" commit a9a0f96.

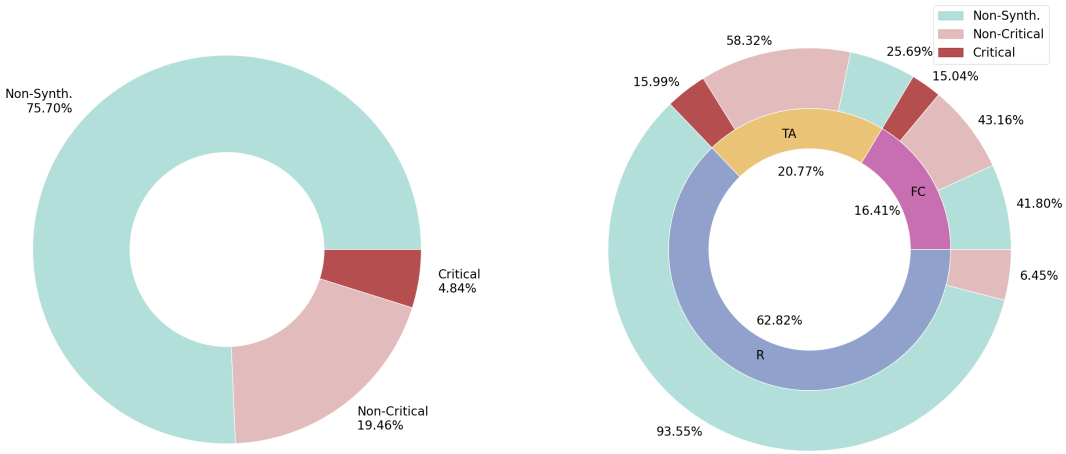


Figure A.12.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "Saghen/blink.cmp" commit a937edd.

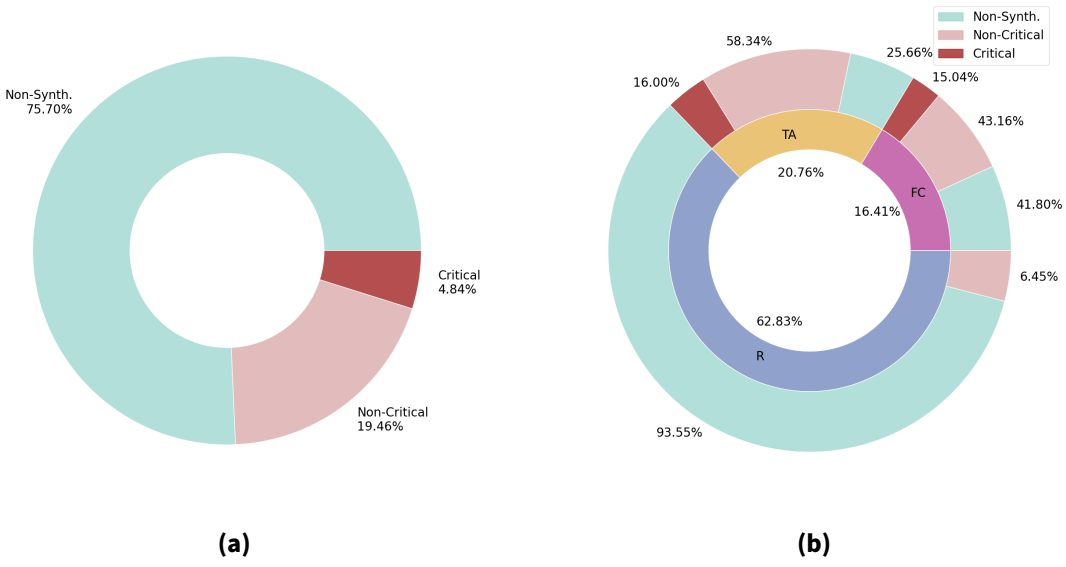
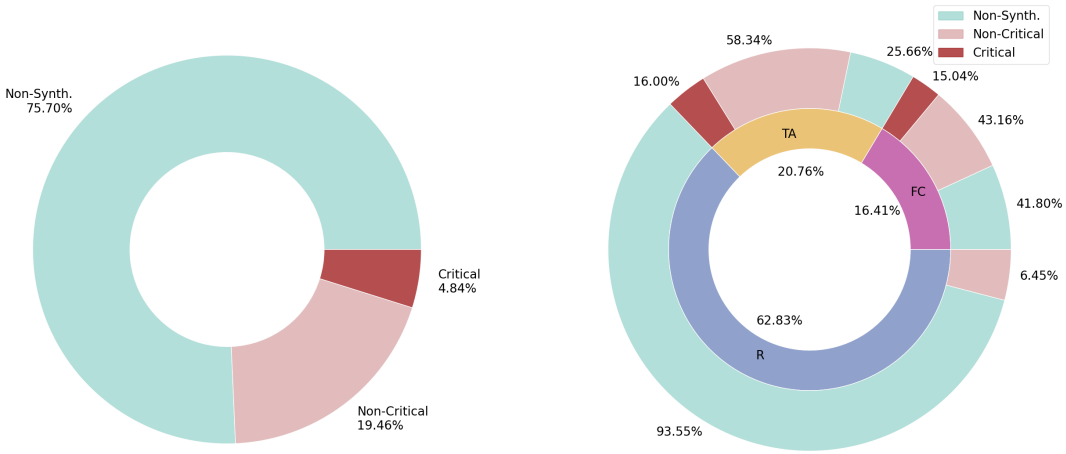
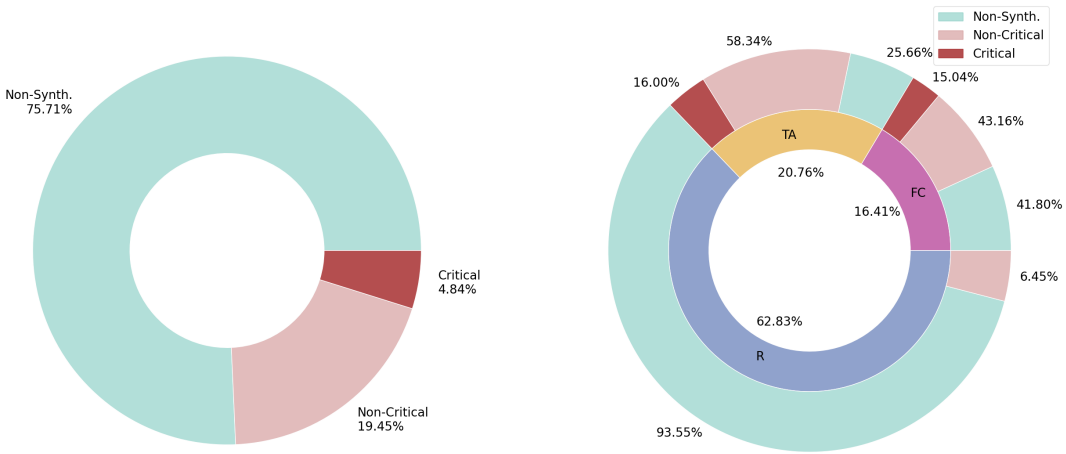


Figure A.13.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "Saghen/blink.cmp" commit 4ef6d1e.



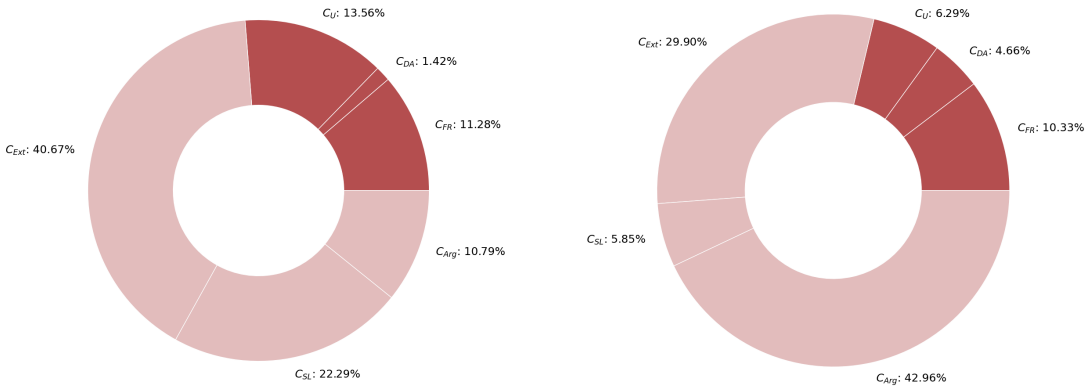
(a) **(b)**

Figure A.14.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "Saghen/blink.cmp" commit 8620a94.



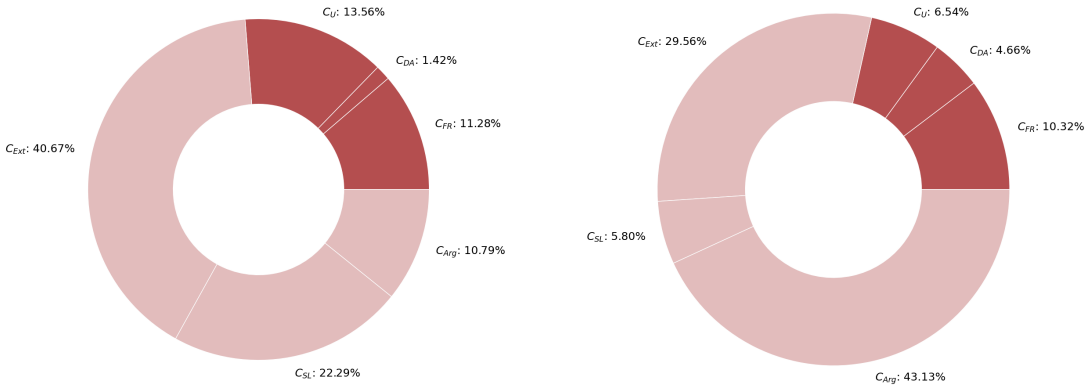
(a) **(b)**

Figure A.15.: Distribution of non-synthetic, critical synthetic, and non-critical synthetic references across all resolved references (a), and each reference type (b), for "Saghen/blink.cmp" commit f93af0f.



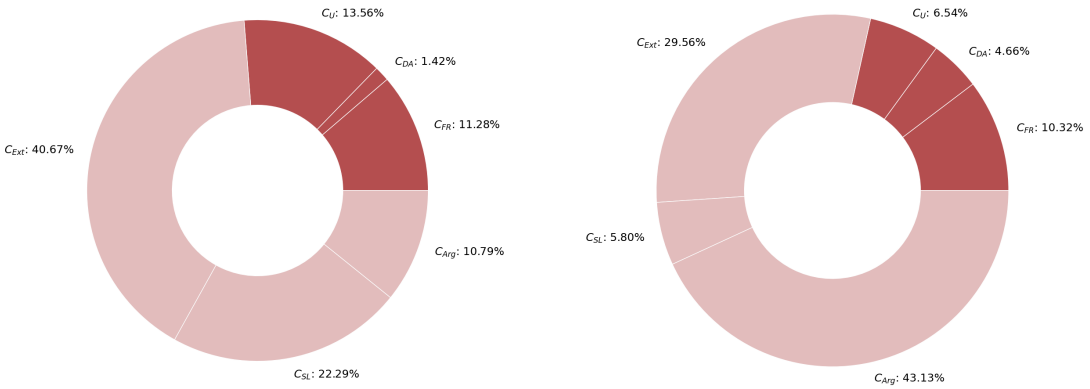
(a) **(b)**

Figure A.16.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "Saghen/blink.cmp" commit a9a0f96.



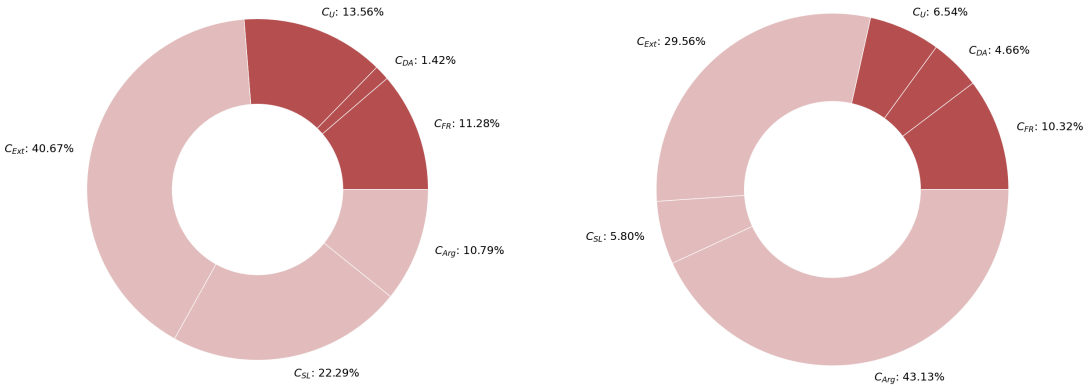
(a) **(b)**

Figure A.17.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "Saghen/blink.cmp" commit a937edd.



(a) **(b)**

Figure A.18.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "Saghen/blink.cmp" commit 4ef6d1e.



(a) **(b)**

Figure A.19.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "Saghen/blink.cmp" commit 8620a94.

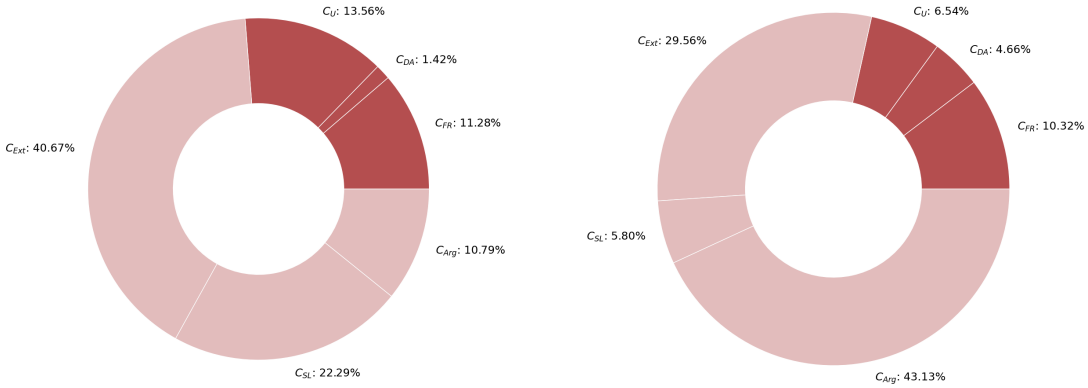


Figure A.20.: Causes for synthetic reference resultion of Function Calls FC (a), and Table Accesses TA (b) for "Saghen/blink.cmp" commit f93af0f.