Lehrstuhl für Rechnerarchitektur & Parallele Systeme
Fakultät für Informatik
Technische Universität München

Interactive Software Parallelization Based on Hybrid Analysis and Software Architecture Reconstruction

Andreas Johannes Wilhelm

Lehrstuhl für Rechnerarchitektur & Parallele Systeme
Fakultät für Informatik
Technische Universität München

Interactive Software Parallelization Based on Hybrid Analysis and Software Architecture Reconstruction

Andreas Johannes Wilhelm

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr. Bernd Brügge

Prüfende der Dissertation:
1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Felix Wolf,
   Technische Universität Darmstadt

Die Dissertation wurde am 15.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.03.2019 angenommen.

To Rafael, Samuel, and Verena

Zusammenfassung

Die kontinuierliche Entwicklung im Bereich moderner Mehrkernprozessoren stellt die Software Industrie vor große Herausforderungen. Es zwingt die Entwickler bestehende Anwendungen zu parallelisieren, damit sie von der zusätzlichen Leistungsfähigkeit dieser Prozessoren profitieren können. Eine solche Modifikation ist jedoch äußerst aufwendig, da industrielle Bestandssoftware in den meisten Fällen sehr umfangreich und komplex ist. Automatische Ansätze zur Parallelisierung sind hier nicht effektiv, da solche Anwendungen nur wenig reguläre Berechnungen enthalten, welche für diese Ansätze günstig sind. Manuelle Verfahren sind hingegen langwierig und fehleranfällig, da skalierbare Lösungen detailliertes Domänenwissen sowie umfangreiche Kenntnisse der zugrundeliegenden Software Architektur und dessen Verhalten voraussetzen. In dieser Arbeit stellen wir einen interaktiven Ansatz zur Identifikation möglicher Parallelität auf verschiedenen Abstraktionsebenen vor. Zur Unterstützung dieses Ansatzes haben wir Parceive implementiert, ein Werkzeug für die Analyse von Software auf Binärcode-Ebene. Parceive sammelt während der Laufzeit der Programme Daten zu deren Verhalten und kombiniert diese mit strukturellen Informationen bezüglich der zugrundeliegenden Software Architektur. Die resultierenden Modelle werden anschließend mit Hilfe mehrerer für Parallelisierung relevanter Ansichten visualisiert. Die meisten Architekturen und Implementierungen industrieller Anwendungen unterscheiden sich gravierend voneinander. Um dieser Varianz gerecht zu werden, schlagen wir außerdem eine regelbasierte Sprache vor, um verschiedene Sichten der implementierten Architektur zu rekonstruieren. Diese Technologie erlaubt Abhängigkeitsanalysen auf unterschiedlichen Granularitätsstufen zur Identifikation verschiedener Parallelisierungsmuster. Wir demonstrieren den effektiven Nutzen dieser Methoden anhand von zwei Fallbeispielen mit Software aus dem Open-Source Spektrum.

Abstract

Continuous advances in multicore processor technology have placed immense pressure on the software industry. Developers are forced to parallelize their legacy applications to make them scalable. However, such applications are often very large and inherently complex; here, automatic parallelization methods are inappropriate and manual parallelization is tedious and error-prone. A dependable software redesign towards scalable parallelism rather requires an in-depth understanding of the problem domain, the underlying software architecture, and its dynamic behavior. We propose an interactive approach to support identification of parallelization scenarios at various levels of abstraction. Thus, we implemented Parceive, an interactive tool that operates on user applications in binary form. Parceive collects behavior information during run-time and combines it with structural information at architecture level to provide useful visualizations and analyses relative to parallelization. Industrial applications differ widely in terms of architectural and implementation styles; thus, we additionally propose a rule-based language to extract and reconstruct various views on the implemented software architecture. This enables multi-granular and iterative dependency analysis, which is crucial to identify efficient parallelization scenarios. We demonstrate the usefulness of our approach on two case studies with open source applications.

Acknowledgments

First and foremost, I would like to thank my advisor Michael Gerndt who taught me how to do research that matters. I am indebted for his guidance and experience, for pushing me out of my comfort zone and become independent in pursuing my goals.

I would like to thank Felix Wolf for being in my dissertation committee. It is an honor to get the input and criticism of such world-class experts.

I would like to thank Tobias Schüle, my mentor at Siemens Corporate Technology, for all the advice and the support he gave me. I feel very fortunate to having worked with him and I wouldn't have gotten this far without this mentorship. Many ideas in this dissertation developed from our lengthy discussions.

All the work presented here is the result of collaboration with many incredibly bright people. I am grateful to Efe Amadasun, Samya Bagchi, Sharma Bharatkumar, Faris Čakarić, Daniel Hogrebe, Yousef Kandalaft, Ranajoy Malakar, Krishna Prasad, Victor Savu, Adriaan Schmidt, and Marcus Winter for their tenacious help with Parceive. Additionally, I would like to thank the numerous great students who participated in our Parallel Programming classes. They helped me to realize the central issues of parallel programming and to shape effective tools to tackle them.

My PhD experience was defined by the interactions I had with the phenomenal team in the Chair of Computer Architecture and Parallel Systems. I am truly grateful to Andreas Hollmann, Madhura Kumaraswamy, Robert Mijaković, Yury Oleynik, Amir Raoofy, Josef Weidendorfer, and Dai Yang for the inspiring moments and making the Chair a fun place to be.

I thank Siemens for generously supporting my research and their important financial support. I thank TUM for providing me a unique working environment and unparalleled resources to pursue my work.

Contents

List of Figures

List of Listings

1

Introduction

"Forget for a while about computers and computer programming, and think instead about objects in the world around us, which act and interact with us and with each other in accordance with some characteristic pattern of behavior."

– C.A.R. Hoare

1.1 Motivation

Stunning advances in information technology transformed how we work and live. The rapid growth in computer performance enables applications that where unthinkable roughly 70 years ago when the first general-purpose computer was built. Face recognition, natural language processing, or self-driving cars are examples for the computational capability of today's systems. Much of this progress was achieved in the last two decades of the 20th century when single-processor performance improved by a factor of 10,000 - an annual rate of over 50% [55]. That growth stemmed from steadily increasing the number and speed of transistors on a processor chip by shrinking their size. Most programmers have enjoyed the regular performance boost and added further levels of abstraction while preserving a sequential programming model.

The situation has changed since the early 21st century when transistor sizes reached dimensions that hit fundamental limits of power efficiency [31]. As a consequence, the microprocessor industry was forced to use multiple efficient processor cores instead of single inefficient processors. That change in hardware shifted the responsibility of ever-increasing performance characteristics to the programmers. They must design their applications using parallel programming models to fully utilize the potential of multicore processors. The increasingly slowing of *Moore's Law* in recent years further strengthened the need for more software parallelism. To satisfy performance expectations for next-generation applications, new types of heterogeneous architectures and domain-specific processing units appeared [55]. This trend results in computer systems that are naturally parallel; consequently, this must also hold for the programs running on those systems.

Unfortunately, parallel programming is notoriously tedious and error-prone. Despite having a plethora of existing parallel programming models and languages, only few programmers can realize the full power of

parallelism; most programmers only achieve a small fraction of the potential performance gains [104]. The difficulties are related to the inherent complexity of parallel programming and the lack of appropriate tools. The main complexity arises from programmer's experience with sequential problem solving. For decades, professionals were trained to create algorithms built upon consecutive and deterministic instructions. However, designing efficient programs requires parallel thinking that accepts the concurrent, non-deterministic nature of computational problems. The programmer's task is to structure programs with concurrent elements that may operate simultaneously. What is needed - besides education - are appropriate tools to support with parallel thinking by facilitating program comprehension and discovery of parallelization opportunities.

Parallelization of existing legacy software adds an extra level of complexity. Besides concurrency issues, programmers must cope with large and unfamiliar code bases with no or obsolete documentation, missing tests, and situations where software architects have left the company. Typically, industrial applications comprise various components at different levels of abstraction that heavily interact with each other. Examples are client-server applications that process numerous simultaneous requests. These applications usually contain database managers, message queues, central logging facilities, and actual business logic. To exploit parallelism and achieve good scalability, those components must concurrently interact with each other. However, necessary restructuring requires a solid understanding of the problem domain, the overall software architecture, and inherent dependencies at implementation level. Consequently, parallelization of industrial applications is very time-consuming and complicated. Increasing the effectiveness of this process is inevitable for ensuring sustainable software systems. For that purpose, programmers depend upon tools to make informed design decisions about program parallelization.

While some parallelization tools address parallelism discovery, they are rarely designed to analyze architectural aspects of software. This makes their findings hard to understand and less expressive as the used abstractions do not conform with the programmer's mental models. Furthermore, their basic assumption - that substantial parallelism can be recovered from serial implementations - does not generally hold. Solutions designed with the sequential programming model in mind may not provide enough concurrency for efficient parallelism. On the other hand, tools for software architecture reconstruction disclose program structures at high levels of abstraction but rarely combine the resulting models with behavioral information relevant for parallelization. Additionally, those tools are known to suffer from significant inaccuracies, mainly caused by two challenges: first, there is no construct in any major programming language to express architectural elements (e.g., "layer" or "component"). Second, the great diversity of implementation and architectural styles in real-world applications requires sophisticated analyses. In conclusion, there is a serious abstraction gap between the problem and program domain, and existing tools for supporting programmers with parallelization.

In this thesis we propose a tool-based approach for interactive software parallelization. Its objective is to improve programmer's software comprehension, which is required to identify possible parallelization opportunities. The proposed tool collects information about program behavior during run-time and combines it with software architecture information. The user provides a set of architecture rules to automatically extract and recover architectural elements at different levels of abstraction. Analysis results are then used to provide effective visualizations and analyses that facilitate parallel computational thinking. The resulting parallelization scenarios represent a blueprint for potential software restructurings.

1.2 Contributions

The contributions of this thesis can be classified into three categories: methods and techniques, tools, and case studies.

Methods and Techniques

- *Defining the abstraction gap problem* of existing tools for parallelism discovery that operate on diverging abstractions relative to the programmer's mental models.
- *Providing a methodology for interactive parallelization* based on an iterative, pattern-based approach. The methodology emphasizes the importance of analyses and scenarios at different levels of granularity [124].
- *Specifying a meta model for hybrid analysis* to enable precise and versatile execution traces optimized for efficient analysis queries. The meta model is tailored towards data dependency analysis at different levels, ranging from single instructions to architectural elements [124].

Case Studies

- *Presenting two programs as case studies.* The first case study originates in our institute and exemplifies all steps of our approach on a small but for this format manageable scale. The second case study is a medium-size open source application that shows the practical characteristics of our tool-based approach, such as scalability.

Tools

- *Parceive* is a parallelization tool that supports dependency analyses at different granularity through visualization and exploration. We integrated the aforementioned techniques into Parceive and validated them with case studies from open source projects and software systems from our industry partner Siemens.
- *A visualization framework for parallelization* to provide a common interface for accessing large trace files and support a wide variety of visualizations. The framework includes various optimization and abstraction techniques to ensure responsiveness and scalability, e.g., on-demand loading, caching, or interactions across visualizations [123].
- *A technique for semi-automatic extraction of architectural representations* from existing software systems. The technique is based on a new language for specifying architectural rules to steer the extraction process and provide effective filter mechanisms for run-time tracing [122].

1.3 Structure of the Dissertation

This dissertation illustrates software architecture analysis for parallelization by proposing guidelines for a pattern-oriented approach and using them to design and evaluate an interactive parallelization tool, called Parceive. The remainder is structured as follows.

Chapter 2 presents an overview of related work in interactive software parallelization. Although we are not aware of work that treats software architecture as first class entity for parallelization, our approach is closely related to parallelism discovery, software architecture reconstruction, and software visualization. The chapter surveys relevant theory and work in these areas.

Chapter 3 introduces our proposed pattern-based parallelization approach. We give an overview of its workflow before describing the individual steps, and how Parceive supports their typical use cases. The chapter presents our proposed language and technique for software architecture reconstruction together with decisive visualizations and analyses of the tool.

Chapter 4 illustrates important design decisions in Parceive's architecture and implementation. Our objective is to provide enough insights for solid reproducibility of our findings and support effective maintenance of our tool. Thus, we first describe the tool's overall software architecture and highlight essential interactions between major components. Then, we provide more details of the backend, frontend, and the component for software architecture reconstruction, respectively.

Chapter 5 introduces two case studies for interactive parallelization of sequential programs. The first program is a soccer simulator developed at our institute. During the case study we show how Parceive provided active support with the identification and realization of parallelization scenarios. The second program is a static program analysis tool by the open source community. Here, we highlight the benefits of having dependency information at architectural levels.

Chapter 6 concludes our work by discussing our approach and the lessons we learned. We then present a set of research directions that are opened by this thesis.

2

Background and Related Work

"If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice."

– Donald Ervin Knuth

2.1 Introduction

The proposed tool-based approach touches a wide range of research areas, such as parallelization, software architecture reconstruction, and software visualization. We are not aware of existing approaches that cover these research areas collectively; however, several approaches share similar objectives with ours.

2.2 Interactive Parallelization

Parallelization is the process of transforming a sequential program to exploit parallel hardware. This process is based on the distribution of computational problems among multiple processing units, such as CPU or GPU cores. Parallelization results in a parallel program; a specification of execution units (e.g., software entities to execute instructions) that operate simultaneously and communicate among each other to solve the computational problems.

Parallelization can improve many program qualities, such as latency, throughput, or the software design. Reduced latencies lets parallel programs solve their computational problems faster than the sequential counterparts. Parallel software with increased throughput is more scalable, solves larger problems, or produces results with higher resolution. Finally, a parallel software design ideally reflects the inherent parallelism of computational problems. This helps to naturally express the behavior of a program, reduces the accidental complexity of its source code, and improves its understandability.

Unfortunately, parallelization is recognized as a tedious and error-prone activity. It requires a greater effort for the programmers, who must consider the peculiarities of parallel execution environments. Proven techniques for validation, verification, and optimizations are not directly applicable to parallel programming. These techniques depend upon deterministic program execution, which is not given with parallel execution environments. Additionally, existing issues may be exposed a long time after deployment.

Various approaches to help programmers in parallelizing applications have been proposed. For example, automatic parallelization has been considered as the silver bullet [20, 10, 126]. Automatic parallelization approaches allow programmers to use their existing sequential programming models for exploiting the benefits of modern computer architectures. To preserve application semantics, the employed analyses must make conservative decisions; thus, automatic parallelization is only applicable to small code blocks and simple loops without intricate dependencies [99]. Semi-automatic parallelization leverages the programmer's knowledge to enable additional parallelism [16, 46]. Here, user input informs the tools which dependencies can be safely ignored. Unfortunately, identifying such dependencies is not trivial and sometimes impossible without major software redesign toward concurrency. A promising alternative are interactive procedures, which give the developer full control over parallelization. Interactive parallelization essentially requires tools for parallelism discovery, i.e., tools that support programmers with identifying dependencies and estimating the efficiency of parallel software solutions.

Parallelism discovery tools rely on static and dynamic software analysis to detect problematic dependencies and estimate potential speedup. These techniques differ in the time of their data acquisition; static analysis collects information before program execution and dynamic analysis during program execution. Consequently, static methods analyze program specifications (e.g., source code) whereas dynamic analyses record run-time events (e.g., memory accesses). This implies diverging qualities of both methods. Static analyses can efficiently inspect different control flow paths but must rely on imprecise assumptions about potential program states. Those assumptions lead to false-positive results, i.e., static analysis is overly conservative and may miss crucial parallelism opportunities. On the other hand, dynamic analysis records precise program states during execution but is limited to the traversed control flow paths. Thus, those analyses can return false-negative results, i.e., they potentially miss parallelism-inhibiting dependencies and their results must be double-checked. Our proposed tool uses a combination of static and dynamic analysis (a so-called *hybrid analysis*) for parallelism discovery.

In the remainder we summarize important vocabulary and notations relative to interactive parallelization. Afterwards, we introduce generic design patterns before presenting important pattern languages for parallelization.

2.2.1 Vocabulary and Notations

Parallelism discovery is based on two important theories: performance estimation and dependency detection. We now provide relevant foundations used in this thesis.

Parallel Performance

A main purpose of parallelization is to improve a program's performance. Two important performance aspects are *latency* and *throughput*. Latency is important for reactive systems; it describes the unit of time (e.g., execution time) that is necessary to complete a task. Throughput is the rate at which a series of tasks can be completed. It is measured as units of work per unit of time. Parallel programming can improve latency and throughput by computing sub-problems simultaneously. However, some parallel software designs improve the latency of certain computations but decrease the overall throughput, or vice versa.

Relevant performance metrics for parallelization are *speedup*, *efficiency*, and *scalability*. Speedup compares the latency of the sequential program with latencies of the parallel program computing the same problem on P processing units. Assuming T_0 is the latency of the (fastest) sequential program, and T_P is the latency of the parallel program on P processing units, then:

$$S_P = \frac{T_0}{T_P} \qquad \text{(speedup)}$$

Note that the baseline of S_P is derived from the sequential program. This is a fair method since it shows the absolute speedup by parallelization. Sometimes we are more interested in the relative speedup effects caused by synchronization, communication, or load balancing. A workable approach is to execute (or simulate) a serialization of the parallel program and use the resulting latency (denoted as T_1) as the dividend.

Efficiency is speedup divided by the number of processing units P:

$$E_P = \frac{S_P}{P} = \frac{T_0}{PT_P} \qquad \text{(efficiency)}$$

The efficiency of a parallel program measures its utilization of processing units. Optimal efficiency is 1 (mostly reported as 100%), where the speedup is linear to the processing units. Usually, efficiency decreases as the number or processing units increases, i.e., the speedup is sublinear. The main reason for sublinear speedup is parallelism overhead, such as task management, synchronization, or communication. Note that there can be situations that result in super-linear speedup (i.e., $S_P > P$ and $E_P > 1$), such as beneficial cache effects (more processing units have more cache memory) or algorithms that solve inherently parallel problems more effectively.

Before starting a (complex) parallelization task, programmers must estimate the expected speedup. For a fixed problem size and a varying number of processing units P, the speedup can be computed with Amdahl's law [4]. This is sometimes called *strong scaling*. Gene Amdahl's observation was that programs usually contain inherently sequential operations that are non-parallelizable. Hence, the law separates execution time T_0 into two categories: time that is spent doing serial work, and time that is spent doing parallelizable work. Assuming s is the serial fraction of T_0, then:

$$S_P \leq \frac{1}{s + (1-s)/P} \qquad \text{(Amdahl's law)}$$

Amdahl's law has serious consequences for the efficiency of parallel algorithms. For example, assume a program that spends 90% of its execution time with perfectly parallel work. When the parallel program is executed on a system with four processing units, it runs over three times faster and has a notable efficiency of 77% (left part of Figure 2.1). But the situation gets worse when executed on systems with more parallelism. When using 32 processing units, the expected efficiency drops already below 25%. Even if we would assume infinite hardware resources, the achievable speedup is limited to 10.

Note that Amdahl's law assumes the same problem size for the sequential and the parallel program. John Gustafson noted that this assumption does not hold for more complex applications that solve increasingly larger problems [52]. Here, the parallelizable part of the problem frequently grows much faster than the serial part. With increasing work the serial fraction of the execution time gets more insignificant and the speedup raises. Assuming s is the serial fraction of T_P, and the problem size increases linearly with the number of processing units P, then:

$$S_P \leq s + (1-s) * P \qquad \text{(Gustafson's law)}$$

The resulting speedup results are sometimes called *weak scaling*. Using the same ratios as above (i.e., $s = 0.1$), our example program runs 3.9 times faster on a system with four processing units (right part of

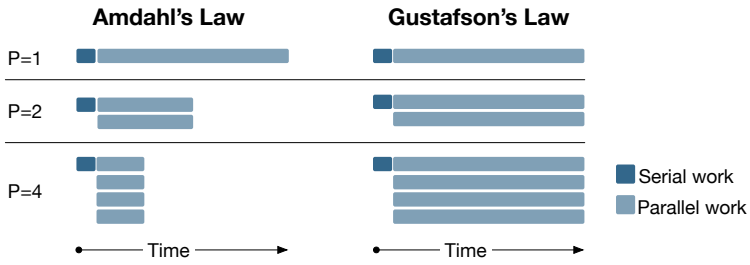


Figure 2.1: Conceptual comparison of Amdahl's law (left) and Gustafson's law (right).

Figure 2.1). This is an efficiency of 98%, resulted by the four times larger (parallel) problem size. Using a parallel system with 32 processing units raises the efficiency even to 99,7%.

Both Amdahl's law and Gustafson's law are valid methods for speedup estimation. It depends on the computational problems and the input data which one is more appropriate. Unfortunately, both methods compute speedups too optimistic since they neglect any parallelism overhead. Although there are more elaborate methods to determine upper (and lower) bounds of speedup (e.g., the work-span model [15]), the presented laws suffice to identify good parallelization opportunities in most cases. We apply the methods to justify parallelization opportunities within the presented case studies (Chapter 5).

Data Dependency

A data dependency is a situation where two code regions access the same memory location. For sequential programs this is no issue; the statements are always executed in the same order, resulting in deterministic memory accesses. However, concurrent programs comprise non-deterministic memory accesses. Their order depend on the relative timing between interfering threads and differs for various program executions. Thus, data dependencies between concurrent code regions can lead to different computational results. These situations are called *race conditions* [95]. They become critical bugs when the order of execution is not intended by programmers, producing issues that are hard to reproduce and debug. Hence, knowing data dependencies is essential for parallelization. Our approach relies on a more formal definition to identify data dependencies.

Assume R_1 and R_2 are code regions; then, R_2 depends on R_1 if one or more of the following conditions hold:

$$O(R_1) \cap I(R_2) \neq \emptyset \qquad \text{(flow dependency)}$$

$$I(R_1) \cap O(R_2) \neq \emptyset \qquad \text{(anti dependency)}$$

$$O(R_1) \cap O(R_2) \neq \emptyset \qquad \text{(output dependency)}$$

where:

- $I(R_i)$ is the set of memory locations read by R_i.
- $O(R_i)$ is the set of memory locations written by R_i.
- There is a feasible execution path from R_1 to R_2.

These conditions are called Bernstein Conditions [13]. They essentially state that R_1 and R_2 can be safely executed in parallel if there is no (data) dependency between them, i.e., if

$$O(R_1) \cap I(R_2) \cup I(R_1) \cap O(R_2) \cup O(R_1) \cap O(R_2) = \emptyset$$

Note that the formalization is not limited to the granularity of code regions. The conditions can be applied to various levels of abstraction, such as single statements or whole components of the software architecture. Our approach relies on this property for analyzing data dependencies between arbitrary software elements.

The dependency types are not equally critical for concurrent programs. In case of anti and output dependencies, the second memory access is a write operation and determines the final value of the memory location. Usually, these dependencies can be resolved by memory duplication. Unfortunately, this approach is not valid in case of flow dependencies, where the second memory access reads the written value of the first memory access. These dependencies must be satisfied in concurrent programs to preserve the results.

2.2.2 Pattern-based Parallelization

Design patterns are textual representations of proven solutions for frequently occurring problems. Originally intended for civil engineering [3], design patterns enjoy great popularity in the object-oriented computing community [42, 22]. They capture expert's knowledge and vocabulary to design dependable software and foster communication among stakeholders. A typical formalism for design patterns contains six parts: a description of (1) the problem, (2) the proposed solution, (3) the problem's context, (4) relevant forces that must be considered, (5) few examples to illustrate the pattern, and (6) a final discussion that draws connections to related patterns.

A *pattern language* describes a structured collection of related design patterns. The objective is to cover a broad range of granularity levels to design complex systems. At each decision point, the designer chooses an appropriate pattern to solve a specific problem. Every design pattern can lead to other patterns at different granularity levels until reaching a final design. In retrospectives, design decisions are often obscured by complex code; pattern languages make these decisions explicit and comprehensible. Additionally, using a web of patterns facilitates communication about various levels of software design by providing a common vocabulary.

Pattern languages for parallel programming describe models on how to create parallel software designs [87, 88, 105]. They comprise patterns at different levels of abstraction, ranging from low-level algorithms to architectural patterns. Those languages aim for composable patterns to apply them at most parallelization tasks. Our tool-based approach facilitates pattern-oriented parallelization by analyses and visualizations that provide enough flexibility to consider arbitrary patterns.

Although most parallel pattern languages focus on creating parallel programs from scratch [85], their proposed patterns are equally suited for parallelization of existing software. However, there are two important differences between working on green-field projects and existing code. First, programmers who conduct parallel programming for new software mostly belong to the original development team whereas parallelization of legacy code is often done by people without detailed knowledge of important design decisions. Missing information about relevant program parts complicates the identification of the right patterns and prolongs validation processes. Second, fresh software designs can be easily adapted to new requirements and parallel computing. On the other hand, existing software architectures may not match the design of parallel patterns. Adapting these architectures often requires restructuring the software or even some tedious re-engineering.

In the remainder we introduce OPL (Our Pattern Language) [1], a pattern language that heavily influenced our approach. It focuses on important programming structures and processes for parallel programming.

Example: OPL

The newest version of the OPL resulted from merging two projects: the PLPP (Pattern Language for Parallel Programming) project, which focuses on a pattern based procedure [87], and a research initiative centered at the UC Berkeley to consider the larger problem of well-engineered parallel software architectures [66].

OPL comprises 56 patterns, organized into five categories: structural, computational, algorithm strategy, implementation, and parallel execution (Figure 2.2). Each category addresses a specific level of the design problems, such that patterns from different categories can be flexibly combined. The intention is to encompass the complete parallel design of software systems, ranging from their overall organization at an architectural level to the low-level details of the parallel algorithm.

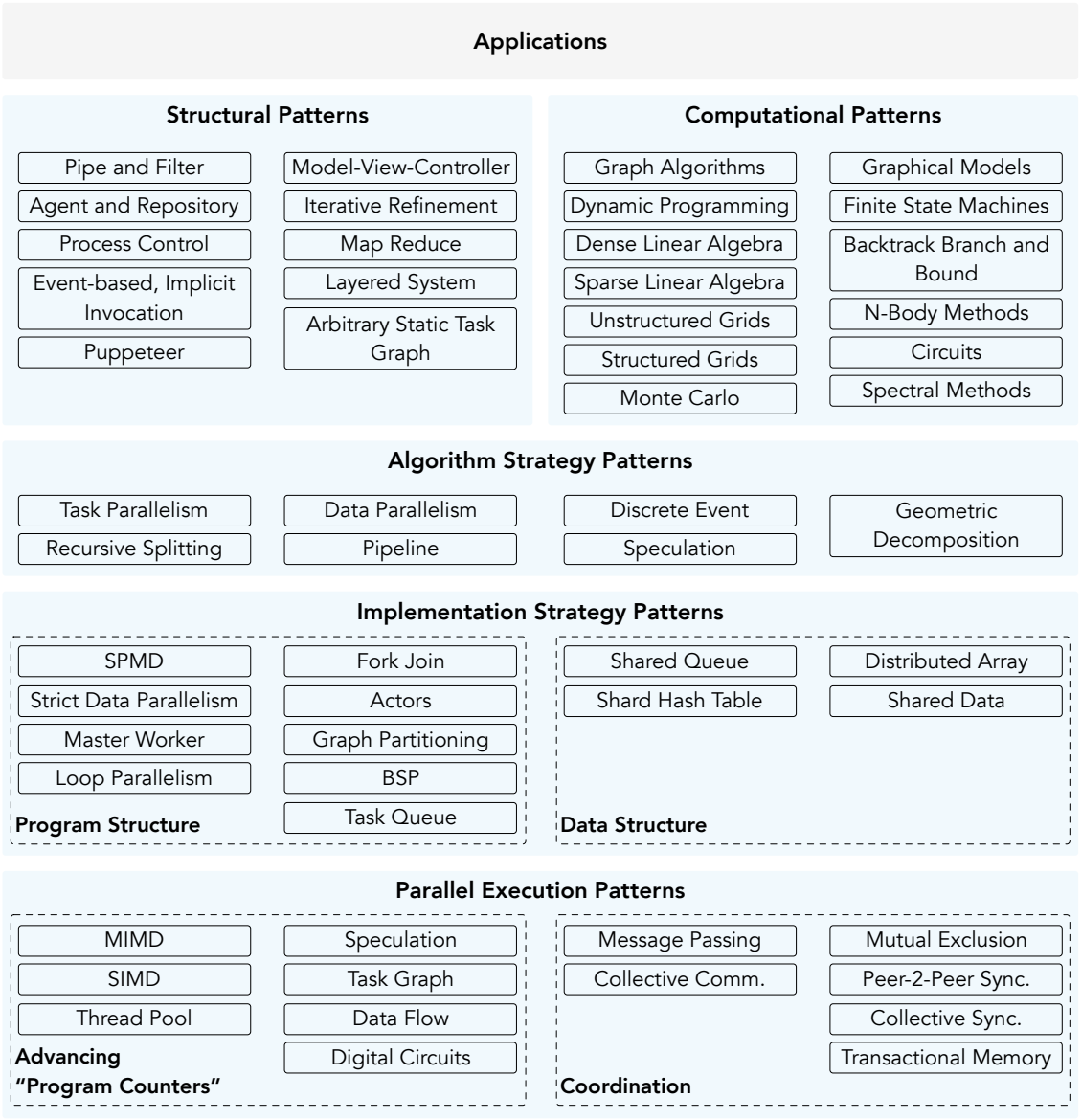


Figure 2.2: Parallel patterns language OPL [1].

1. *Structural Patterns* describe the overall structure of a program at software architecture level. Inspired by architectural styles [107, 44], those patterns define a set of high level components and their interactions. Patterns construct a supportive framework for the program's computational elements. Parallelism is not explicitly integrated into the resulting software architectures; rather, the architectures support and simplify parallelization implied by patterns from lower categories. For example, layered systems restrict interactions between specific abstraction levels, leading to software architectures that are relatively easy to understand and maintain. These qualities ease identification of appropriate com-

putational patterns and relevant dependencies. Maintainers may iteratively adjust these patterns and the layered system until a proper design is found.

2. *Computational Patterns* describe proven solutions for important computational problems. They originate from previous work that identified essential problems which benefit from parallelism [6]. Solutions for different computational problems may be composed to form structural patterns. Furthermore, computational patterns may itself comprise lower-level patterns. As with structural patterns, parallelism is not directly expressed within this category. However, the computational structure proposed by those patterns fosters essential task and data decomposition for parallel execution. This consequently requires strategy and execution patterns from OPL's lower levels. Examples are Monte Carlo simulations, which run independent experiments to statistically sample the solution space. Possible solutions build upon the structural *Map Reduce* pattern [29], where different experiments are conducted by separate tasks in the map phase and the results are accumulated in the reduce phase.

3. *Algorithm Strategy Patterns* specifies high-level strategies to solve individual computational problems. Here, the programmer's key issue is to find most appropriate parallelization strategies for the given problem and software structures. Besides efficiency, scalability, simplicity, and portability, good algorithm strategies must consider the organization of the problem, i.e., if the problem is decomposed into tasks, data, or streams of data. Most effective strategies may combine different parallelization strategies at different hierarchy levels. For example, algorithms that perform subsequent computations on data streams can be parallelized using a combination of the *Pipeline* [120] and *Task Parallelism* patterns. Independent computations operating on different streams are performed by multiple pipeline stages simultaneously. Additionally, operations of single computations may run in parallel by multiple tasks. For each parallel algorithm there may be multiple implementation strategies that may fit, which are concerned by lower-level patterns.

4. *Implementation Strategy Patterns* describe how individual processes and threads execute the chosen algorithm designs. They can be divided into patterns for program structure and data structure. Program structure patterns define processing elements that are executed in parallel, their relations and interactions. For example, the *Master Worker* pattern provides good load balancing for multiple (mostly independent) tasks. It comprises two types of processing elements, a master that distributes work packages, and workers that operate on these work packages and collect their results after completion. On the other hand, data structure patterns are concerned with the mechanics of sharing data between processes or threads. They describe concurrent data containers including their interfaces, suitable contexts, and their progress guarantees (e.g., blocking, lock-free, or wait-free [56]). For example, the *Shared Queue* pattern specifies a queue container that provides safe thread management. Often, multiple program and data structure patterns can be implemented or combined; thus, programmers must accurately weigh their concrete advantages and disadvantages. The patterns also inevitably express a bias towards particular programming models. For example, the SPMD pattern is mostly implemented with a message passing framework, such as MPI. Hence, the decisions made during this level highly influences the parallel programming model.

5. *Parallel Execution patterns* define how software elements are mapped to hardware for exploiting parallelism. At this level, programmers can estimate (and measure) the resulting performance of different realizations. The patterns are divided into two categories: process and thread control, and coordination. The former category comprises patterns for advancing control flows of concurrent program executions. For example, thread pools avoid time-consuming operations to create and destroy threads. Whenever a new thread is needed, one is used from the pool (if available). Whenever a thread gets destroyed, it is returned to the pool. As opposed to the execution of threads and processes, the second pattern category is concerned with their coordination. The comprised patterns define different

mechanics for communication and synchronization of concurrent software elements. Message passing and mutual exclusion are typical examples for these two categories, respectively. The former lets different processes coordinate their work by explicit communication messages passed over interconnection networks. The latter synchronizes accesses to shared resources (e.g., memory locations) by restricting simultaneous execution of containing code regions to single threads or processes. Most of the parallel execution patterns are included within major parallel programming models, such as OpenMP, or MPI. Some patterns are native to specific programming models but foreign to others.

Besides the described patterns, OPL includes a procedure model to support parallel programmers through the entire development process. Here, designers iteratively move between different design spaces to find patterns that constitute the best parallel software architecture. The procedure model originates from the PLPP pattern language [87], which is organized into four design spaces (Figure 2.3):

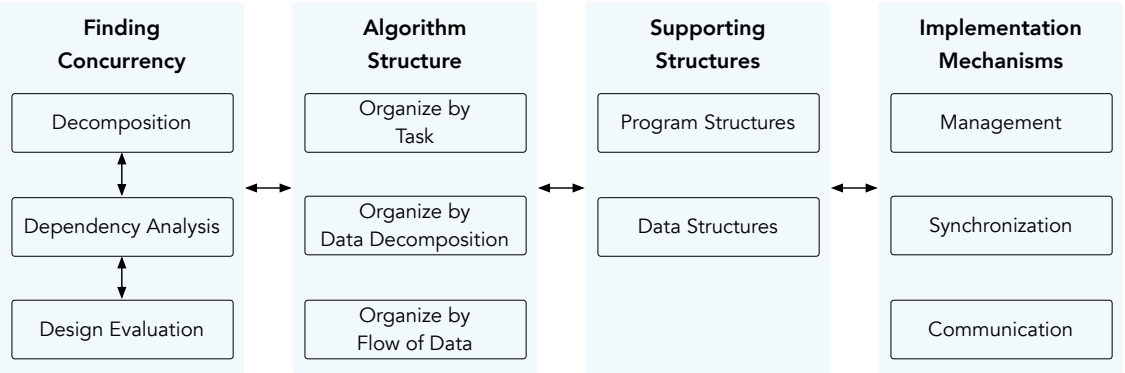


Figure 2.3: Design spaces of the PLPP pattern language [87].

1. *Finding Concurrency.* This design space is concerned with analyzing and structuring computational problems to exploit concurrency. Here, designers aim to understand relevant key features in the problem domain and justify the effort for parallel programming. To support this process, the design space contains several patterns organized into three related categories: decomposition, dependency analysis, and design evaluation. The decomposition patterns are intended to split the problem into multiple sub-problems that may execute simultaneously. Dependency analysis patterns help to group tasks and analyze dependencies among them. Design evaluation compares different decompositions relative to different qualities, such as suitability (for the target platform), flexibility, efficiency, or simplicity.

   Note that the design space abstractly describes parallel computations and is not concerned with algorithms or implementation details. We use the same distinction in our approach to define parallelization scenarios (Section 3.3).

2. *Algorithm Structure.* Here, the intended software design is refined towards a parallel program by mapping concurrency onto multiple units of executions. This mapping can be done in numerous ways; however, related patterns mostly follow one of three major organizing principles: organization by task, organization by data decomposition, or organization by data flow. The patterns of this design space are similar to Algorithm Structure patterns from the OPL An organization by task is recommended if the execution of tasks constitute the best algorithmic principle. An organization by data decomposition is the best choice if data is the major principle in understanding the concurrency. Finally, if the flow of data imposes an ordering or grouping of tasks, an organization by data flow is recommended.

   Our parallelization approach considers algorithmic strategy patterns during the identification of parallelization scenarios and a (potential) software restructuring (Section 3.3).

3. *Supporting Structures.* This step represents an intermediate stage between the problem-oriented patterns of the Finding Concurrency design space and the programming techniques in the Implementation Mechanisms design space. The contained patterns correspond to OPL's Implementation Strategy patterns and are grouped into program and data structuring patterns. Patterns in the first group describe code structures to define concurrent software elements. Examples are SPMD, Master/Worker, Loop Parallelism, etc. Patterns in the second group have to do with managing data dependencies between multiple tasks. The Shared Data pattern deals with the generic case, the other patterns describe other frequently used data structures (e.g., queues or hash tables).

   The design space exclusively corresponds to the restructuring stage of our approach (Section 3.3). Depending on the selected patterns, existing programs require substantial restructuring to integrate the program and data structures. For example, the SPMD pattern requires dividing loop-based algorithms that operate on single data structures into parallel computations that solely communicate via explicit messages.

4. *Implementation Mechanisms* are concerned with program's source code and low-level operations to manage parallelism. The contained patterns are principally comparable to OPL's Parallel Execution patterns; however, these patterns are categorized differently. The distinction is made between management, synchronization, and communication of execution units. OPL combines the latter two categories into a group called Coordination.

   We do not explicitly consider those implementation mechanisms within our approach since they are implied by major parallel programming models.

2.3 Tools for Dynamic Parallelism Discovery

Tools for identifying exploitable parallelism have been developed in academia and industry. These tools heavily rely on dependency analysis techniques to detect independent software elements. Purely static approaches for these analyses are overly conservative; hence, recent approaches have increasingly relied on dynamic information gathered during program execution. In this section we cover two categories of tools that are most closely related to Parceive: quasi-automatic and annotation-based parallelism discovery.

2.3.1 Automatic Parallelism Discovery Tools

Tools within this category automatically detect and evaluate potential parallelization opportunites in sequential programs. The user is responsible to select and implement appropriate suggestions, based on detailed feedback from those tools about potential speedup and detected obstacles for parallelization. The required information is typically acquainted by profiling and tracing to capture run times and dependencies during program execution, respectively.

DiscoPoP

DiscoPop [75] is a tool to assist programmers with identifying and exploiting parallelism. The tool discovers parallelization opportunities and rank them according to their parallelism potential. Unlike conservative approaches [126, 33], the method is not limited to loop parallelism and enables identification of parallel tasks that are not well aligned with the source code structure.

DiscoPoP's workflow for parallelism discovery comprises two major steps: a hybrid analysis to build a dependency graph, and a step for discovering and ranking of parallelization opportunities (Figure 2.4).

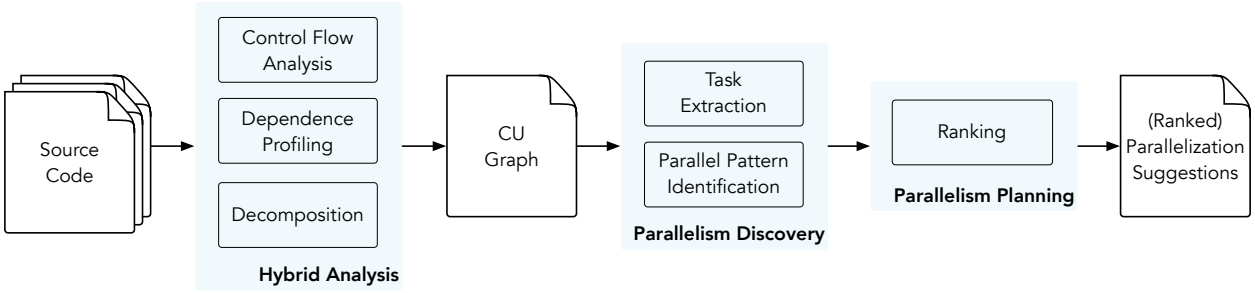


Figure 2.4: DiscoPoP's workflow for parallelism discovery [75, 64].

In the first step, control flow analysis is used to determine boundaries of control regions (e.g., loops) within a program's source code[1]. Then, the code is instrumented and executed to obtain its control and data dependencies. The final step of the hybrid analysis decomposes the profiled program into a graph of *computational units* (CUs), the author's basic blocks for building parallel programs. CUs are collections of instructions following a read-compute-write pattern. A CU graph represents a program with CUs as nodes and dependencies between them as edges.

During the second step, the tool searches for potential parallelism in the user program by merging CUs and partitioning the CU graph. Task extraction and pattern identification detect the following parallel patterns in the CU graphs: DOALL loops, DOACROSS loops, SPMD tasks, and MPMD tasks. After finding pattern occurrences, those are then ranked and suggested to the user according to their resulting speedup and parallelism-inhibiting dependencies [61].

DiscoPoP's parallelism discovery abilities exceed Parceive's features. Their automatic pattern detection and ranking algorithms alleviate the user from manually identifying concurrent software elements. Some extensions (not shown in Figure 2.4) transform user code into parallel versions [129] and validate them relative to concurrency issues [62, 63]. Finally, data dependency profiling in DiscoPoP is more efficient than our used models by relying on a signature algorithm, dependency merging, and static instrumentation. However, the latter technique limits dependency analyses to software modules with available source code. Hence, data accesses within many third-party libraries (e.g., container libraries) - which are often essential for parallelization - are not captured.

Kremlin

Kremlin [43] discovers parallelism by detecting the (hierarchical) critical path for all code regions of a program. The tool uses this information to compute the *self-parallelism* of code regions, a metric to rate the inherent parallelism potential. The tool returns a *parallelism plan*, an ordered list of code regions that are favorable for parallelization (e.g., regions with loop parallelism or task parallelism). The programmer is responsible to determine how to expose the underlying parallelism detected by Kremlin.

The tool's workflow comprises three phases for parallelism discovery: static analysis, dynamic analysis, and parallelism planning (Figure 2.5). During the static analysis phase, Kremlin inserts specific callback functions into program's source code by performing critical path instrumentation and region instrumentation. The first step establishes a profiling infrastructure for critical path analysis while the second step instruments and extracts the region structure to localize parallelism. During the dynamic analysis phase, the tool executes the instrumented binary (linked with the KremLib instrumentation library) with a sample input to produce parallelism profiles for each program region. These profiles contain the total amount of work

[1] more concretely: the resulting LLVM IR [73] generated from program's source code.

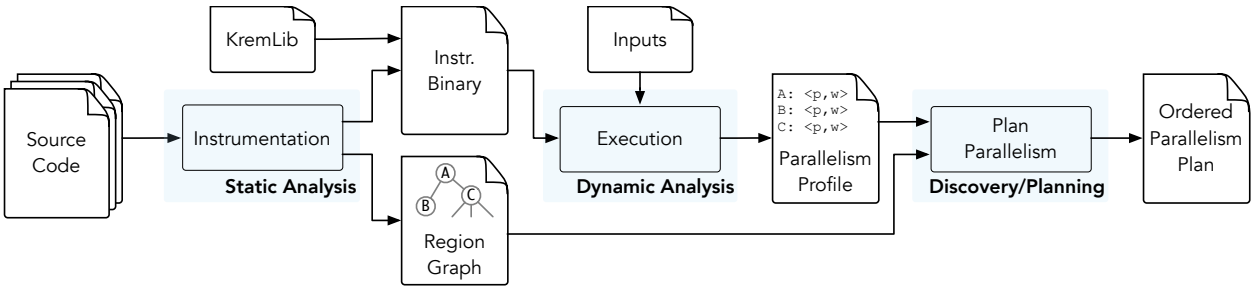


Figure 2.5: Kremlin's workflow for parallelism discovery [43].

and the computed parallelism excluding any from subregions. During the final phase, Kremlin computes a parallelism plan by relying on the extracted region structure and the parallelism profiles. The tool therefore considers target-specific (e.g., OpenMP) and machine-specific parameters (e.g., the number of CPU cores) to estimate speedup results based on Amdahl's law.

Similar to DiscoPoP and Parceive, Kremlin performs dependency analyses to detect (true) data and control dependencies. However, the collected information is exclusively used for implicit parallelism planning; Kremlin does not explicitly highlight those parallelization obstacles for the suggested parallelization opportunities. Thus, programmers are forced to discover these dependencies manually when exploiting the parallelism.

SLX

SLX is a "multicore development tool" [48]; a tool suite assisting with software parallelization tailored for embedded platforms, such as a multiprocessor system on chip (MPSoC) or FPGAs. The tool suite is developed by Silexica, a start-up company that origins from the MAPS research project [24]. MAPS aims at parallelizing C applications for MPSoC platforms.

SLX belongs to the class of quasi-automatic dynamic parallelism discovery tools since it provides *parallelization hints*: identified parallelization opportunities that are ordered by their relative and absolute speedup and by detected dependencies. For that purpose, SLX uses a pattern-based framework to identify parallelism at task level, pipeline level, and data level.

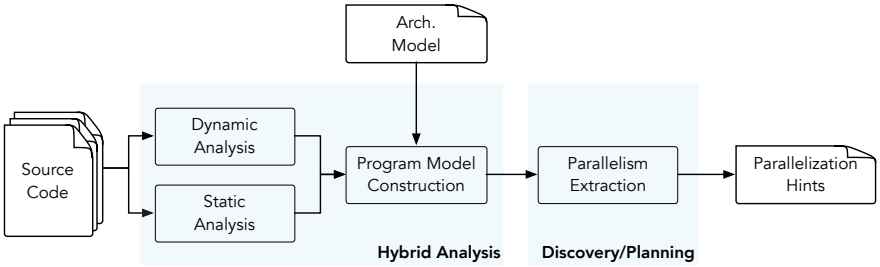


Figure 2.6: SLX's workflow for parallelism discovery.

The workflow expects sequential source code as input and applies a set of static and dynamic analyses based on compiler-based instrumentation (Figure 2.6). Then, the gained information (e.g., structural information and trace data) and a given hardware architecture model is used to generate a program model. That hardware architecture model contains relevant information of the intended target platform for speedup estimations, such as the number of CPU cores, execution costs for the instruction set and for communication, or the memory architecture. Finally, several visualizations present the simulation results, such as a graphical call graph and a source code visualization with annotated speedup results and dependency information.

Alchemist

Alchemist [128] uses dynamic binary instrumentation for its analyses to unveil parallelization opportunities across loops, function calls, and if-statements. The tool detects code constructs that can run asynchronously with its context, similar to *futures* [9]. Alchemist thus profiles the duration of executed code constructs and between conflicting memory references from inside these constructs to their continuation. This is conducted by an online algorithm that manages an index tree for currently active code constructs to detect all data dependencies between them. After completion, the tool provides a ranked list of code constructs together with an estimation of the required work for manual parallelization (i.e., the number and type of found data dependencies).

Similar to Parceive, Alchemist is not restricted to specific parallelization patterns or programming models. This implies more effort for the developer yet it allows more flexibility in the design of parallel software.

2.3.2 Tools for Assisting Parallelism Discovery

Tools within this category rely on the knowledge of programmers and architects to evaluate their program's parallelism potential. As opposed to quasi-automatic approaches, identification of parallelization opportunities is conducted by the user. The tools assist this procedure with profiling results, pattern detection, and detailed information about potential implications, such as estimated speedup, problematic dependencies, or required transformations.

SUIF Explorer

The SUIF Explorer [76] is among the first parallelization tools that incorporate user interaction. It actively guides programmers to decide on proper program transformations for loop parallelization. The tool combines analysis results from an own compiler with dynamic program profiles and represents them with interactive visualization techniques.

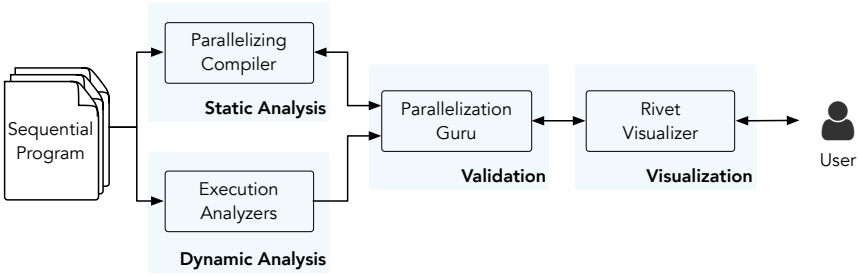


Figure 2.7: Major components of the SUIF Explorer [76].

During the parallelizing process, the tool first invokes a compiler [126] that automatically parallelizes loops without loop-carried dependencies (Figure 2.7). Then, the parallel program is instrumented and executed to collect profiling information and data dependencies. The Parallelization Guru module analyzes static and dynamic information to identify long-running sequential loops that would benefit from parallelization. The user is involved in the parallelization process by the Rivet visualization. It pinpoints the identified loops and the dependency information concisely presented by program slicing techniques. By adding annotations to the source code (or manually transforming source code), the compiler can consequently parallelize loops.

The SUIF Explorer enables - as Parceive - dependency analyses across arbitrary deep levels of function calls. However, the author's system perform these analyses strictly on loops, which excludes other higher-level

programming models. Due to the underlying compiler infrastructure, the SUIF Explorer provides many useful features which might improve Parceive's effectiveness. For example, the compiler uses interprocedural analysis and alias analysis to statically detect data dependencies. Although static analysis potentially returns false-positive results, the gained information would favorably extend the dynamic data dependencies found by Parceive.

Intel® Threading Advisor

The Advisor XE [27] is a well-established proprietary tool suite for parallelism discovery. Besides abilities for vectorization optimization and flow graph analysis, the suite contains a prototyping tool to model threaded software designs called Threading Advisor. The two main features of this tool are detection of performance bottlenecks using the roofline model [125] and what-if analysis for predicting data sharing problems among user-selected code regions.

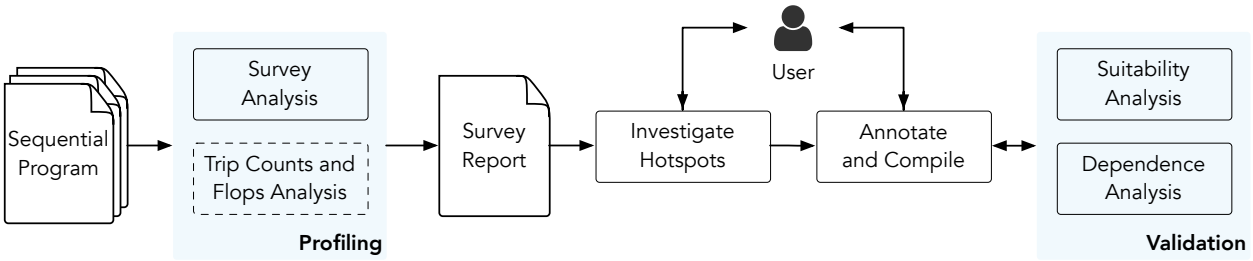


Figure 2.8: Parallelization workflow of the Intel® Threading Advisor [27].

The tool's workflow comprises three phases: profiling, investigation and annotation, and parallelism validation (Figure 2.8). During the profiling phase, a Survey Analysis identifies loops and functions with a substantial run time and aggregates the loop's trip counts. The result is a Survey Report, a visualization of beneficial code regions for data parallelism, loop parallelism, and task parallelism. In the second phase, the user annotates selected code regions based on the profiling report (we give more details on these annotations below). During the final phase, the Suitability Analysis executes the annotated program binary to estimate the parallel performance characteristics of annotated regions, i.e., scalability ranges for various target models specifying the processor types, the number of cores, or the programming model. Additionally, the Advisor performs deep dependency analyses within the annotated parallel regions based on dynamic binary instrumentation.

The Advisor's approach shares many aspects of Parceive; most notable is its idea of annotation-based dependency analysis. As opposed to Parceive's visualization-based parallel region selection, the tool from Intel accepts the following types of source code annotations:

1. *Parallel site and task annotations.* Parallel sites are named code regions containing at least one task that executes concurrently with other tasks of the same site. The tasks are itself code regions, e.g., loops or a set of consecutive instructions. Parceive's users can perform equivalent analyses on call and loop level by selecting the respective nodes in the CCT View (Section 3.3). Our tool does not support instruction-based dependency analysis.
2. *Lock annotations.* These are explicit synchronization annotations that are hints for the dependency analysis to ignore specific data dependencies. This feature is not supported by Parceive since we do not provide multi-pass analyses.
3. *Pause and resume annotations.* These annotations let the user pause and resume the collection of data accesses within parallel regions during analysis runs. This improves the analysis performance

and reduces the amount of shown (incidental) data races. Parceive provides similar functionality on function level (and above) through static and dynamic filter rules (Section 4.7).

4. *Special-purpose annotations.* Recently, the Threading Advisor has been extended by interesting annotations for special purposes. The underlying use cases partly follow the same insights for parallelization as we identified. For example, a particular annotation let programmers identify all accesses to a specific (heap) memory region. Parceive provides an equivalent feature on call and architecture level (Section 3.3).

As opposed to Parceive, the Advisor uses a multi-pass analysis, i.e., the tool performs separate executions for profiling and dependency analysis. This reduces the overhead incurred by dynamic binary instrumentation frameworks; however, it requires a recompilation due to source code annotations, which may be infeasible in some industrial settings. Additionally, the savings in slowdown may be negligible for programs with substantial parallelism. More run time spent in parallel regions requires more overall analysis time.

Tareador

Tareador [8] is a framework to support parallelization that was developed for educational purposes. It provides an application programming interface (API) to annotate user tasks similar to the approach from the Intel® Threading Advisor. Annotated task regions are used by the author's framework to estimate the resulting parallelism and highlight problematic data dependencies. For speedup estimations, Tareador relies on the OpenMP programming model with explicit support for loop level parallelism. Tareador also offers a set of trace visualizations which allow the programmer to understand the actual behavior of the parallel decomposition.

As opposed to the Advisor's two-pass methodology, Tareador relies on a single pass workflow for parallelism discovery (Figure 2.9). The programmer starts with annotating the source code with three types of functions: scoping the analysis, defining task boundaries, and hints for data dependency analysis. The annotated code is executed sequentially and the tasks are executed in the invocation order. During this execution, Tareador collects an execution trace with data regions accessed by each potential task. The backend component reads this trace and feeds three different visualizations. First, a usage visualization of the data accessed by each task. Second, a task dependency graph with all tasks represented hierarchically. Third, time-plots showing the potential parallel execution produced by a parallelism simulator.

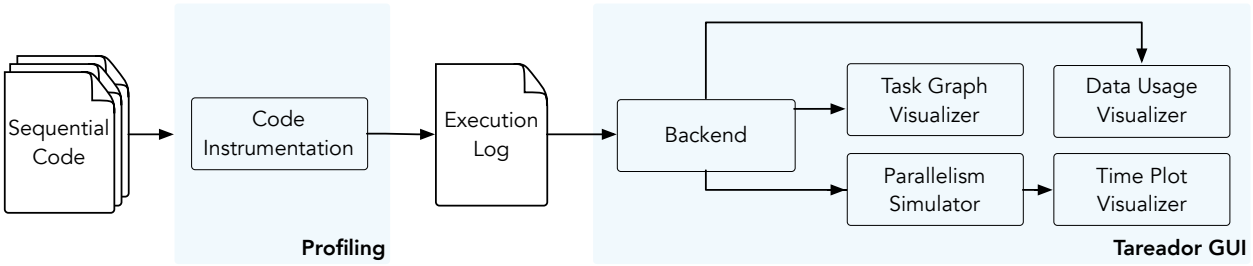


Figure 2.9: Tareador's workflow for interactive parallelization [8].

Although the functionality of Tareador is below the one of Advisor (e.g., the specialized annotations for locking or dependency analysis), the framework facilitates useful features for parallelism comprehension. For example, the task graph visualization supports the user with identification of execution patterns, such as recursive graph algorithms. This information may be very insightful to elaborate more efficient parallelization opportunities or consider specialized parallel programming models. Parceive shares this idea of visual pattern recognition by its provided visualizations.

2.3.3 Summary

In this section we have introduced state-of-the-art tools for dynamic parallelism discovery. Based on available information from published research papers and documentation, we now summarize the most notable findings.

The presented tools differ widely in the concerned parallelism types and the targeted levels of abstraction[2] (Table 2.1). As parallelism on loop and function levels is relatively easy to detect, it is supported by all parallelism discovery approaches. Parallelism that is unaligned with the given source code structure requires different models for detection; appropriate tools must consider concurrency in non-continuous code regions. Parceive supports parallelism discovery at various levels, ranging from call to architectural levels.

	DiscoPoP	*Kremlin*	*SLX*	*Alchemist*	*SUIF Explorer*	*Advisor*	*Tareador*	**Parceive**
Architecture Level	-	-	-	-	-	-	-	✓
Design Pattern Level	✓	✓	✓	-	-	-	✓	✓
Loop Level	✓	✓	✓	✓	✓	✓	✓	✓
Function Level	✓	✓	✓	✓	✓	✓	✓	✓
Instruction Level	✓	✓	✓	✓	-	✓	-	-
Unaligned	✓	-	-	✓	-	-	-	✓

Table 2.1: Comparison of parallelization tools by parallelism type.

As opposed to existing parallelization tools, Parceive considers architectural structures as first class elements for parallelism discovery. This entails additional flexibility to identify and validate parallel patterns without the limitation on single programming models. Parceive's provided visualizations enable users to explore their software systems (Section 3.2). This facilitates program comprehension and software restructuring as it gives detailed insights at different perspectives of the software.

Note that many features of the presented related work can be usefully integrated into Parceive. For example, parallel pattern detection would perfectly match with the Architecture View (Section 3.3) to highlight potential parallelization scenarios at high levels of abstraction.

2.4 Software Architecture Reconstruction

Before presenting state-of-the-art approaches for software architecture reconstruction (SAR), we must define software architecture. In this thesis we follow IEEE's definition of software architecture as '*the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*" [57]. We sometimes abbreviate software architecture as *architecture* (as opposed to *hardware architecture*).

Software architecture is most usefully thought as a shared mental model to facilitate communication and understanding of software systems [59]. Having therefore a clear and current representation of a software

[2]We categorize data and task level parallelism as loop level, function level and instruction level, respectively

architecture is a prerequisite for complex development and maintenance tasks, particularly for parallelization. Unfortunately, architecture documentation is rare in practice. Either the architecture has never been recorded, or the documentation got out of sync as software systems evolve. A solution is to obtain representations of the software architecture from existing systems, which is the aim of software architecture reconstruction.

Software architecture reconstruction is a process of reverse engineering. It aims to build, maintain, and understand a representation of an existing software architecture in form of architectural views [32]. The literature uses several other terms to refer to SAR: reverse architecting, architecture mining, extraction, recovery, or discovery.

SAR is an interactive, interpretive, and iterative procedure that heavily relies on tool support [12]. Those tools extract information about the system, typically by inspecting software artifacts (e.g., source code, build scripts, or traces from running systems). The main difficulty for SAR tools stems from the large diversity of software systems and their architectures. Different software systems have different architectural styles; even if two systems have the same style, they probably differ in their implementation mechanisms. To get dependable results, knowledge of requirements, reference architectures, or design constraints is important. This requires attention, skilled reverse engineers, and (in the best case) original architects or programmers.

2.4.1 SAR Processes

SAR follows the same principles as proposed by theories of program comprehension. An early theory argues that programmers use a bottom-up process to comprehend software. Low-level information, such as source code, is progressively aggregated to form architectural models until a high-level understanding is reached [108, 11]. An opposite theory proposes a top-down process. Developers start with high-level knowledge, such as requirements or reference architectures, to generate hypotheses on the architecture and verify them with the implementation [19, 93]. Finally, hybrid theories combine and alternate bottom-up and top-down processes to build a mental model of the software [74, 111]. Low-level information gets aggregated to form architectural elements, high-level knowledge is refined and mapped against these elements.

Top-Down Processes

SAR tools that are based on top-down processes match conceptual hypotheses of a software system with program artifacts to discover its software architecture [19]. These hypotheses are formulated from deriving high-level knowledge, such as requirements or architectural styles.

The reflexion model is a typical example for top-down processes [93]. It matches conceptual views of a software architecture with concrete views to highlight convergences, divergences, and absences (Figure 2.10). Programmers derive conceptual views (in the form of architecture rules) from their architecture hypotheses, which include elements and their relationships. The concrete views are extracted and abstracted from program artifacts (e.g., semi-automatically [68, 96] or automatically [25]). Finally, the views are compared to create a reflexion model that shows their differences. Here, convergences locate elements present in both views, divergences are elements that are only in the concrete view, and absences are elements that are only in the conceptual view. The process intends an iterative refinement of the software system (or the hypotheses) until the conceptual and the concrete software architecture converge. This works well if applications are sufficiently modular to detect coding patterns in architectural elements [12]. Unfortunately, most techniques are limited in their applicability due to an inherent mismatch between the static code-base and run-time aspects.

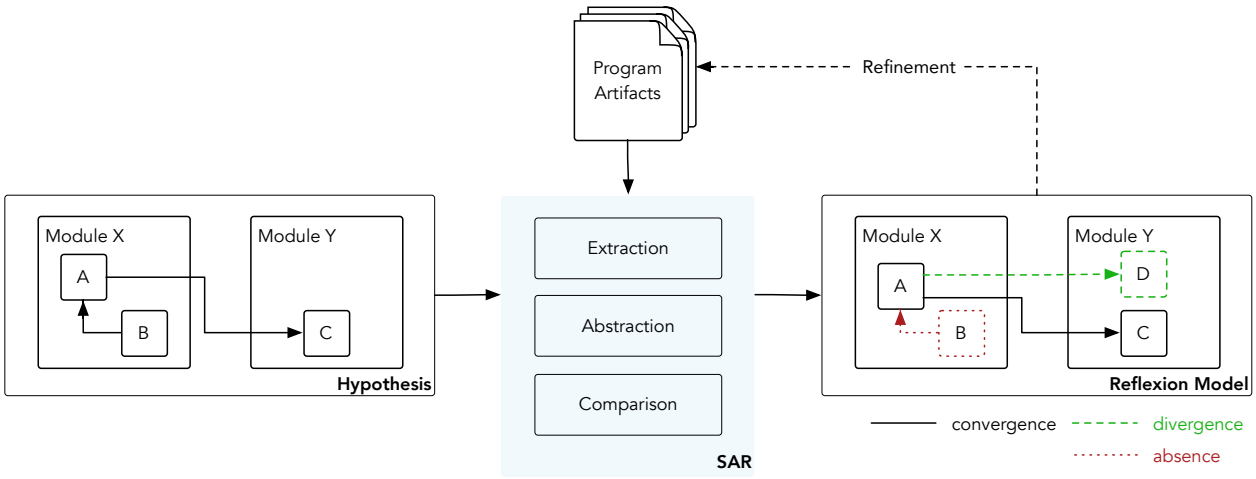


Figure 2.10: The Reflexion Model, a top-down process for SAR.

Bottom-Up Processes

Tools for SAR that follow bottom-up processes start with low-level artifacts of a software system and progressively reduce them by filtering and abstraction until a high-level view is reached. These tools are closely related to the extract-abstract-present cycle for reverse engineering (Figure 2.11).

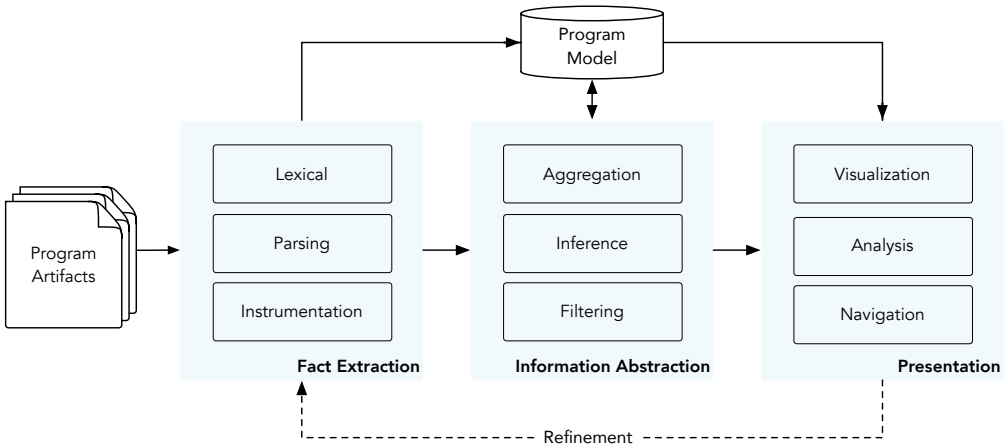


Figure 2.11: The extract-abstract-present process of SAR approaches.

Techniques used for fact extraction include static analysis (e.g., parsing), dynamic analysis (e.g., tracing), and informal data extraction (e.g., interviewing). For program comprehension, extracted data must be put into a representation that reflects the user's perspective of system characteristics and permits efficient analysis of its artifacts and relationships. This representation is usually captured into a program model (e.g., a repository) that stores relevant static and dynamic system properties.

Information abstraction is a fundamental conceptual tool for managing the complexity of software systems. SAR tools have to handle two complexities: (1) the number of elements and relations in large software systems (and their resulting models) can grow very large; and (2) major programming languages do not explicitly support architectural elements, such as layers or connectors. Hence, even tool-supported program comprehension approaches can be unmanageable without proper organization principles. Aggregation, fil-

tering, and inference techniques are required to structure and reduce the model data for representation. Typically, this step is interactively performed by the programmer [32, 92, 65, 53, 80].

The presentation of program models "holds the key to program comprehension" [115]. Presentation's main purpose is information exploration through navigation, analysis, and visualization. Navigation enables users to traverse the program model relative to multi-dimensional relationships of software artifacts. Examples for such relationships are component hierarchies, data and control flow, or memory accesses. Analysis derives, extracts, and refines information from the program model that is not explicitly represented. The analysis results produce insightful and focused views that supports the programmer's understanding of software systems. Finally, visualizations offer a wide variety of views and features to represent architectural aspects [32], ranging from architectures as boxes [65, 69] to software cities in virtual reality [35].

Although Parceive's design is appropriate for hybrid SAR processes, it focuses on parallelization rather than conformance checking of software architecture. Hence, our tool follows a bottom-up process for reconstructing architectural views. During the fact extraction step it constructs a program model by combining structural information and execution traces. The former is extracted by parsing debug information enclosed by object files. Opposed to approaches that solely rely on source code, our tool obtains detailed post-compilation information even for third-party libraries. Execution traces are obtained by instrumenting binary instructions and writing behavioral information during run-time. Parceive relies on dynamic information to analyze relationships between architectural elements that are most relevant for parallelization. Aggregation, inference, and filtering of the program model is performed during the extraction, during post-processing, and while users interact with the visualization. The tool provides multiple orthogonal views to visualize, navigate, and analyze model data.

2.4.2 Techniques

Techniques for software architecture extraction are orthogonal to the SAR process itself. Fully automatic techniques do not exist; hence, experienced reverse engineers are important for the quality of reconstruction results [32]. The literature categorizes extraction techniques according to their automation level, ranging from slightly assisted to mostly automated techniques.

The first category of extraction techniques require most user interaction. Reverse engineers are expected to manually identify architectural elements from initial model representations. Depending on the underlying SAR process, these representations depict low-level elements or high-level components. In the former case, users manually abstract the elements by using interactive and expressive visualization tools [71]. In the latter case, the tools provide architectural views and let users progressively explore them, starting from the highest level artifacts [80].

Another category of extraction techniques in SAR tools automate recurring steps during refinement and abstraction. Besides implicitly encoded abstraction rules within SAR tools, the reverse engineer can steer these steps by specifying reusable abstraction rules and execute them automatically. Some approaches define these rules using relational queries to abstract data from entity-relationship databases. Here, languages like SQL [65], relational algebra [58], or Tcl [92] are utilized to query, group, decompose, or transform program models. Other approaches rely on expressive logic queries to reconstruct architectural views from static (and dynamic) facts [101]. Other approaches are directly based on the lexical and structural information in the source code. Examples include pattern specification languages to identify architectural patterns in software systems [97], or support multiple inputs (files, programs, Acme information) to present various views [53]. Finally, there are SAR tools that rely on investigation-based approaches, such as pattern recognizer, graph pattern matching, state engines, or mapping scripts.

The last category of extraction techniques are mostly automated approaches. These include formal concept analyses, a lattice theory used to identify design patterns, features or modules [116]. Another automated abstraction technique is clustering algorithms that group similar elements according to some classifications. Examples are naming conventions, cohesion metrics, or object interactions [69]. Finally, to detect layers and cycles in large programs others have used the Dependency Structure Matrix (DSM) representation [112]. This matrix concisely shows dependencies between source code entities.

Parceive combines rule-based architecture extraction, implicit abstraction algorithms, and interactive exploration techniques. Users steer the extraction procedure by declaring a set of reusable architecture rules based on our specific extraction language. For example, reverse engineers may use the rules to decompose the software architecture according to the recursive directory structure. The rules are then applied on the raw software artifacts collected while parsing program's debug symbols. Afterwards, Parceive refines and abstracts the high-level architecture model according to a series of abstraction algorithms. For example, one algorithm groups methods and member variables to their respective class artifacts. Finally, the architecture visualization lets users interactively explore the resulting program model in a top-down fashion.

2.4.3 Software Architecture Visualization

Visual representation of software architecture belongs to the field of software visualization [82]. Architectural views are used for various activities, such as testing, performance tuning, program comprehension, conformance checking, or software analysis. We focus on three decisive classifications of software architecture visualizations: scope, method, and user interactions.

Visualization Scope

The scope of architecture visualization includes representations of static, dynamic, and hierarchical information [41].

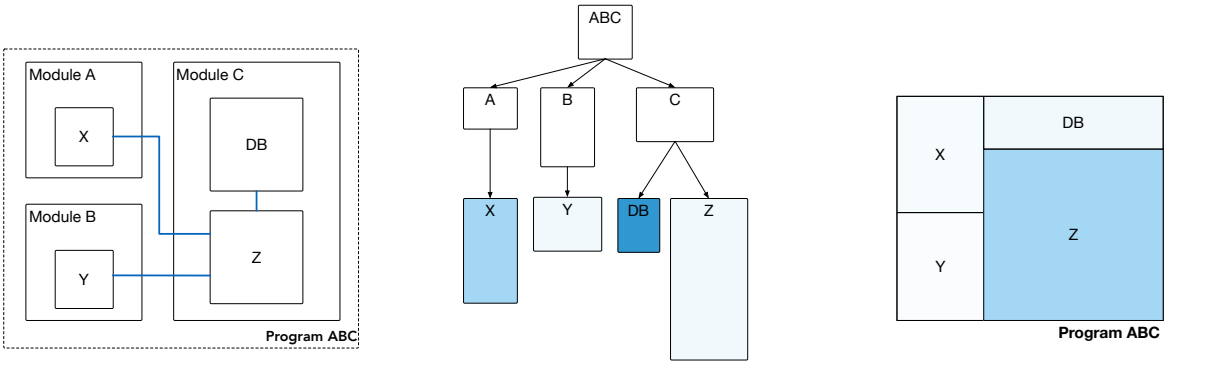
Static information is the dominant source for software architecture representation as it gives trusted and complete facts [106]. Static information is extracted by analysis of static program artifacts, such as source code, debug symbols, or documentation files. Respective visualizations can operate on different levels of abstraction, ranging from variables and classes at design level to components and subsystems at architecture level. Besides these architectural elements, visualizations can show static relationships between them, such as inheritance, association, or usage dependencies.

Static information suffers from limited insights into the run-time nature of programs. Some architecture visualizations therefore utilize dynamic information to represent behavioral aspects. Dynamic information is collected during program execution by annotating, instrumenting, or monitoring software. These procedures determine the form of resulting run-time models, such as execution traces, profiling data, or logs. Visualizations annotate [51] or extend [127, 121] architectural representations to show dynamic elements. Typical run-time events that are relevant for visualization are function invocations or object creation.

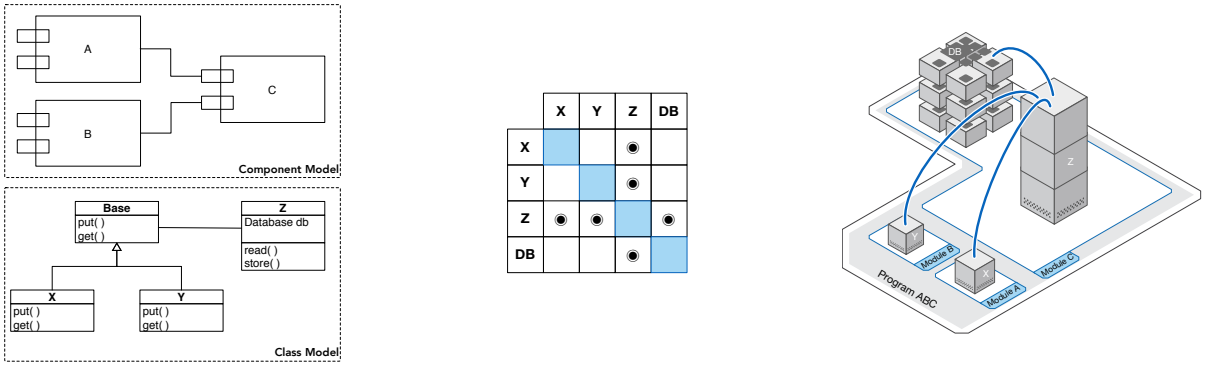
Architectural representations may incorporate historical information to support program comprehension and maintenance. Related views expose how the architectural components evolved across different versions of the software. The basis for historical information is the widespread use of versioning control systems. Visualizations show architectural changes between different versions at various levels of abstraction [45, 60]. These changes indicate architectural elements that are subject of interest for maintenance.

Visualization Methods

Many visualizations approaches have been proposed to facilitate user's comprehension by showing architectural aspects of programs [89]. Most of these approaches support multiple viewpoints to depict different perspectives of the software systems at various levels of abstraction. Some viewpoints are shown by single views; some are combined into single views. For clarity, we focus on four individual techniques to visualize architectural concerns: graph-based visualization, notation-based visualization, matrix-based visualization, and metaphor-based visualization.



a. Nested node-link visualization. b. Polymetric view visualization. c. Treemap visualization.



d. Notation-based visualization. e. Matrix-based visualization. f. Metaphor-based visualization.

Figure 2.12: Software architecture visualizations of a simple software system.

Graph-based visualization is the most popular technique for depicting software architectures [106]. It uses nodes and links to represent architectural elements and their relationships, respectively. Those node-link diagrams can utilize different layouts, such as trees, (nested) boxes, or geometric stacking. Nested node-link graphs show the hierarchy of components and their relationships at different abstraction levels (Figure 2.12 a) [110, 92]. Another node-link visualization are polymetric views, which rely on a tree layout to represent architecture hierarchies (Figure 2.12 b). Instead of having uniform tree nodes, a polymetric view uses rectangular nodes with different widths, heights, and colors to express various metrics (e.g., code size, invocations, or execution time) [72]. Although showing no links between nodes, treemaps can be counted as graph-based visualizations (Figure 2.12 c). They are clean and compact representations of software architectures that size (nested) components according to a single metric (e.g., the number of sub-components). Treemaps are often combined with other visualization layouts, such as polymetric views [80].

Notation-based visualization uses formally defined modeling languages to represent architectural elements. Typical modeling languages are UML (unified modeling language) [118], its extension SysML (systems modeling language) [54], or other specific notations [117]. The languages provide a set of notations for different views of software systems at various levels of abstraction. For example, UML's component and class diagrams use node-link visualizations to show architecture- and design-level aspects, respectively (Figure 2.12 d) [98].

An effective visualization approach to quantify dependencies between architectural elements are matrix visualizations (Figure 2.12 e) [79]. The most prominent examples are Dependency Structure Matrixes (DSM), adapted from the domain of process management [112]. The example highlights the dependencies between `Z` and the components `X, Y` and `DB`.

Metaphor-based approaches provide hierarchical and multi-level visualizations familiar from physical contexts, such as landscapes or cities. The usage of metaphors make the visualizations intuitive (1). For example, visualizations based on the city metaphor represent program's classes or functions as houses with 2D, 3D or even VR (Figure 2.12 f). Different shapes and dimensions of the houses are used for various software metrics to quickly locate interesting program regions. The hierarchy of nested components are represented by layered boxes underneath the houses. Streets (or power lines) with different dimensions indicate relationships between the components.

User Interaction

Visualizations involve the user in the process of navigation, exploration, and analysis of the program representation with different degrees of automation. Most of the approaches require interactions between different views to capture multiple viewpoints of the software systems [106]. Reverse engineers vertically navigate between levels of abstraction and explore their programs relative to structural and behavioral aspects [80].

Some SAR tools rely on analysis frameworks to assist stakeholders in the decision-making process. An example for analyses provided by those frameworks are conformance checks, which identify discrepancies between the conceptual and the concrete architecture. Other approaches support architectural analysis methods, such as SAAM (2) or ATAM (3) to rate architecture's quality. Finally, few tools offer analyses relevant for parallelization and deal with threads, synchronization, or performance [32].

2.5 Related Tools for SAR

Tools that support software architecture reconstruction vary in their goals, processes, inputs, techniques, and outputs [32]. In this section we focus on existing research that is related to Parceive relative to two decisive categories: tools that use an exploratory approach for SAR, and tools that incorporate dynamic information to answer behavioral questions at software architecture level. The common characteristic of these tools is their visualization-based approach for comprehending software architectures. None of the presented approaches are designed towards concurrency-related issues; however, we see similarities with our approach relative to the comprehension-based analyses.

2.5.1 Exploratory SAR Tools

These tools present reverse engineers architectural views to explore structural elements of their software. Typically, these tools use hybrid approaches for data collection and guide users through the program representations starting at the highest level artifacts.

Softwarenaut

Softwarenaut [80] is an SAR tool that supports the "overview first, zoom and filter, and details on demand" principle of software visualization, which we also adapted for Parceive. The tool support the discovery of architectural views of a software system by extracting low-level facts from its source code and aggregating them. The aggregation process uses a decomposition that assumes a hierarchical organization, such as packages in Java and directories in C++. In case of missing hierarchies, Softwarenaut uses its capability for automatic entity clustering.

The process for reverse engineering in Softwarenaut follows the extract-abstract-present cycle (Figure 2.13). In the first step, a code repository is analyzed and data at project and implementation level is extracted into a platform-independent model [114]. Then, a set of automatic analyses apply some metrics and build structural clusters. These analyses perform tasks like collaboration detection to identify programmer relationships, or static code analyses. The reverse engineer interactively explores the resulting architecture representations at different viewpoints. If the generated model is not accurate, the user can amend or filter architectural elements and forward the changes to the model repository. The architecture recovery step involves the user to recover architectural views from the system in a top-down fashion. These views are immediately depicted by Softwarenaut's visualization.

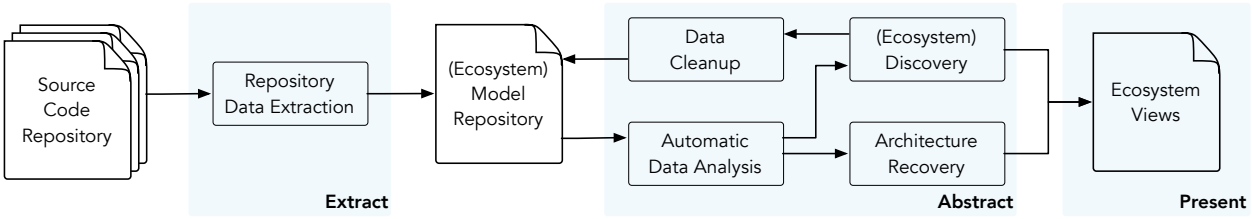


Figure 2.13: Softwarenaut's process for reverse engineering [80].

Softwarenaut focuses on software architecture exploration with architectural views. It provides several features that go beyond the scope of Parceive. For example, the author's tool uses a novel combination of architecture extraction and software ecosystem analysis that incorporates different versions of one or multiple software projects. Although it can reveal different types of dependencies between software components, the information is exclusively obtained by static analysis. Considered data accesses are limited to the level of instance variable accesses for detecting object relationships. On the other hand, Parceive collects data access information during execution for locating data dependencies.

ArchView

ArchView relies on a bottom-up approach for SAR to aggregate low-level data into high-level views. Compared to other approaches, it provides evolutionary analysis and techniques to visualize multivariate data of multiple software releases. Additionally to source code, ArchView accepts modification and problem report data from bug reporting systems. A set of evolution metrics are computed and visualized using polymetric views that provide insights into implementation and evolution of a software system.

The approach follows an interactive and iterative SAR process that is separated into four phases (Figure 2.14). During the initial fact extraction phase ArchView collects information from different source code releases, code modifications (CVS data), and problem report data. The task of the data integration phase is to create the model repository by considering the input data and architecture information from the following architecture analysis. In this phase, low-level information is abstracted to higher level views relative to software modules and source files. The abstraction is controlled by the user to iteratively refine the

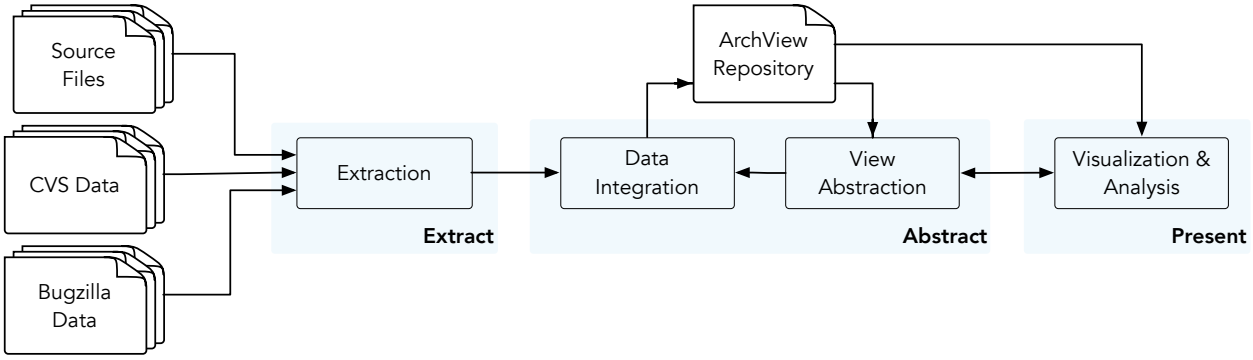


Figure 2.14: ArchView's process for reverse engineering [40].

views with appropriate visualizations. As with Parceive, these visualizations are also used to apply specific analyses on the program representations.

ArchView builds architecture hierarchies according to a set of metrics at different levels of abstraction, e.g., containment relations for architectural components, or invocations and inheritance for implementation level. A methodology for user-specified clustering - as with Parceive - is not supported. Additionally, the lack of dynamic information disables the usage of behavior-related visualization elements.

2.5.2 Dynamic SAR Tools

In this section we present tools for SAR that incorporate data collected during program execution. This data is combined with static information to provide insights into a program's behavior.

Nimeta

Nimeta is also a view-based architecture reconstruction tool that uses a hybrid approach. Similar to Parceive, the author's tool is based on binary relational algebra to define viewpoints and specify abstraction rules. Furthermore, programmers use the formalism to specify architecture conformance checks for validating the extracted model against a conceptual one. The tool supports six different visualization frameworks to depict the viewpoints; examples are Rigi [92] or a custom web interface.

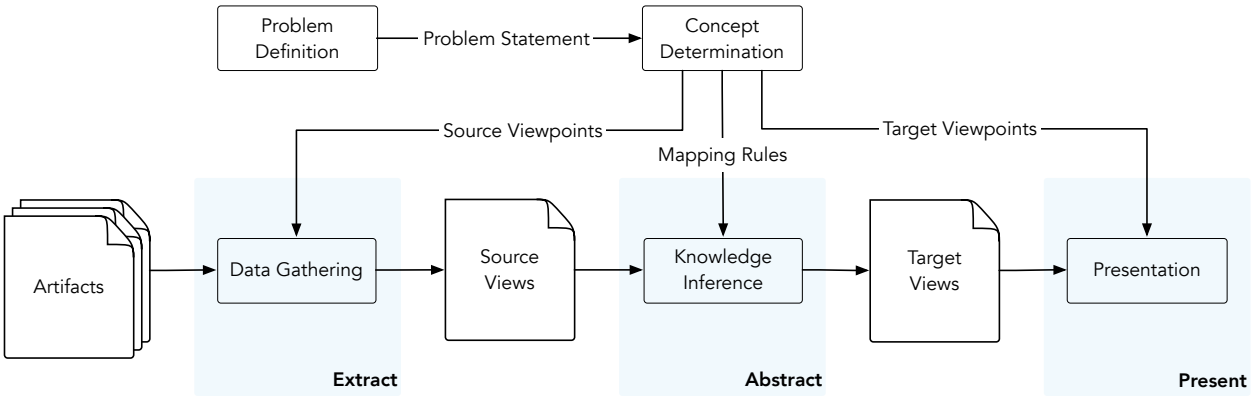


Figure 2.15: Nimeta's process for reverse engineering [102].

The SAR process in Nimeta involves two main actors: stakeholders and system experts (Figure 2.15). At the beginning, information from stakeholders is used to define the problem and determine a reconstruction

concept, i.e., the source and target viewpoints for the reconstruction, and mappings to the implementation. Data gathering is performed on a variety of artifacts (e.g., documentation, source code, or execution traces) according to the target viewpoints. The extraction methodologies are unspecified; however, the author's case studies rely on automatic source code extraction, tracing, and manual specification by system experts. Similar to Parceive, Nimeta uses a set of mapping rules to refine and abstract the extracted model towards a software architecture representation. After this knowledge inference, the visualizations of target views are interpreted with support from stakeholders. Typical activities are conformance checking, architecture assessment, or re-documentation.

Nimeta shares similarities with Parceive's SAR approach, such as the formalism and operations for architecture model refinement or the type of architectural visualizations. However, close integration of Parceive's rule processing and its fact extraction enables a different quality of information refinement. For example, our tool supports filtering and aggregation of run-time events relative to specific architectural elements. Although Nimeta provides dynamic analysis techniques, it is not intended for data dependency detection at the level of detail we require for parallelization support.

ExplorViz

The interesting aspect of the ExplorViz [36] tool is the combination of online trace visualization at application level and components at enterprise application level. The tool aims at supporting system and program comprehension for industrial projects. Possible scenarios include the discovery of communication between components on both the system and application landscape level. The provided visualizations share similarities with UML's activity diagrams for behavior representation and the city metaphor for hierarchy relationships.

ExploreViz's SAR process comprises five major steps that follow the extract-abstract-present cycle (Figure 2.16). In the first step, the tool monitors existing applications in software landscapes with Kieker [119], a third-party monitoring framework that supports control flow tracing (e.g., by manual instrumentation or aspect oriented programming) and resource monitoring (e.g., CPU utilization or memory usage). The data is preprocessed in the second step to enable online processing using trace reduction techniques, resulting in minimized trace data. The abstraction step performs data aggregation and transformation to create a model representation of the software landscape and to transform this representation to a visualization model, respectively. During the final step, the user visually explores the system and landscape model.

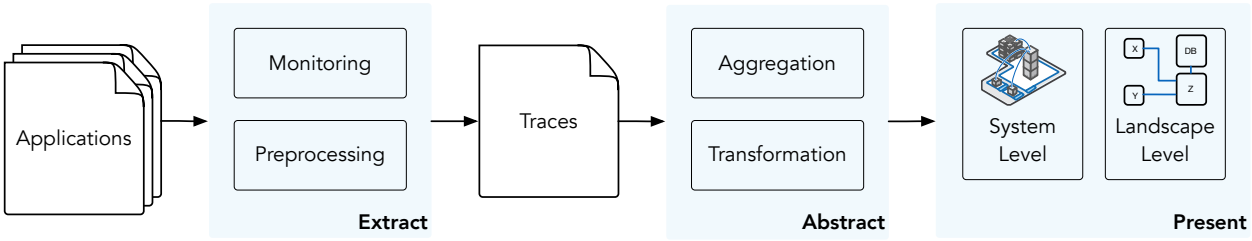


Figure 2.16: ExplorViz's process for reverse engineering [36].

Compared to Parceive, the described tool provides SAR not only for single programs but extends the analysis scope to enterprise architectures. The used visualizations are flexible enough to represent the resulting types of components and their relationships. However, ExplorViz assumes pre-defined containment relations to infer architecture hierarchies, such as packages or object orientation. Here, Parceive is more flexible by its user-guided extraction process.

2.6 Conclusion

In this chapter we presented a selection of important technologies and tools related to Parceive. The decisive methodologies are dynamic data dependency analysis and software architecture reconstruction. To the best of our knowledge, we are not aware of any other tool that combines these two areas for supporting parallelization.

Existing tools for parallelism discovery are closest to our approach as they rely on interactive processes to identify parallelization scenarios at different granularity levels. However, these tools use abstractions that are hardly aligned with mental models of programmers or architects. Parceive forwards the choice of abstractions to the programmers.

On the other hand, tools that support architecture recovery are mainly focused on structural software aspects. If these SAR tools use dynamic analysis methodologies, the behavioral information is limited to conformance checking, architecture assessment, or re-documentation. Parallelization support, however, requires data dependency analysis at a fine-grained instruction level.

3

Approach

"The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstracts away its essence."

– Frederick P. Brooks

3.1 Introduction

In this chapter we present our software parallelization approach, which aims to facilitate the discovery of exploitable parallelism in industrial applications. The approach is actively supported by Parceive, our analysis tool that fosters program comprehension relative to parallelism. Parceive lets users analyze and interactively explore their software by providing a set of different (orthogonal) visualizations. Those represent a combination of run-time data and software architecture information. By relying on Parceive's analyses and visualizations, users find starting points for parallelization and incrementally gain insights to identify potential opportunities for parallel computations.

We believe local parallelization efforts to mitigate single "hot spots" are insufficient for achieving scalable and portable solutions; rather, a comprehensive restructuring at different levels of abstraction is required. Most existing software follows a sequential programming model comprising sequences of instructions with deterministic operational semantics. This simplifies the implementation, composition, and verification of algorithms; however, it limits their parallelism. The (sometimes unnecessary) dependencies between operations induced by the instruction order and the data flow prevent concurrent execution. On higher levels, the software architecture of existing applications regularly obey Conway's Law, i.e., the software architecture follows the communication structures of organizations [26]. This strategy enables efficient engineering of software components across different teams, but entails inappropriate architectures for exploiting parallelism. Approaches that miss altering software designs both at algorithmic and architectural levels leave many sequential computations in user code. According to Amdahl's Law, these sequential parts limit the efficiency of parallel software. As an alternative, we suggest to "think parallel" and restructure software to be inherently concurrent.

Programmer's comprehension of the problem and program domain is key for manual parallelization approaches, like ours. In the problem domain one can reason about the problems being solved at the level of domain-specific design elements, i.e., without the need to consider technical constraints relative to programming or hardware architecture. Proper domain knowledge enables engineers to decompose given problems into individually solvable tasks. Successful parallelization transforms these tasks into software elements within the program domain, relative to selected parallel patterns. Examples are algorithms, data structures, or higher-level components of complex software architectures. The unique challenge is to identify and manage possible dependencies between different tasks. Neglecting the proper execution order of the task's operations - determined by their dependencies - may cause nondeterministic behavior. Working with existing and (partially) unfamiliar software considerably impedes such dependency analysis. Developers not only have to cope with the inherent complexity of the problem domain and its realization in the program domain, but also with accidental complexity induced by inappropriate software abstractions.

To alleviate this tedious and error-prone task, we tailored Parceive towards structured methodologies for designing parallel programs. Most programming problems have several parallel solutions. Finding the best solution is challenging, especially when it profoundly differs from the encoded sequential solution. Structured design methodologies disseminate expert knowledge to make informed design decisions and find the best parallel solutions. Two examples are the PCAM method [37] and pattern-based parallelization [87, 88]. The former presents an exploratory approach that structures the design process into four distinct stages: partitioning, communication, agglomeration, and mapping. Parceive mainly supports with the first two stages, which focus on concurrency and scalability. The third and fourth stage deal with locality and other performance related issues, where other tools (e.g., profilers) are appropriate. Patterns for parallel programming provide a "catalog of good solutions, an expanded vocabulary, and a methodology for the design of parallel programs" [87]. In an iterative process, programmers refine their software structures at various design spaces, using patterns with different granularity and functionality. Parceive supports the design spaces intended for finding concurrency and validating parallel software designs.

In Section 3.2 we present our semi-automatic method for SAR based on user-specified architecture rules. We define an algebraic formalization to describe software architectures, the provided rules, and the operations on the architectural elements. Afterwards, we describe the syntax and semantics of the rule language and demonstrate it on small code fractions. Architecture reconstruction is an essential part of our parallelization approach, which we describe in Section 3.3. For each step, we show how Parceive's analyses and visualizations help with the distinct challenges. Section 3.4 concludes the chapter.

3.2 Software Architecture Reconstruction

Writing dependable and high quality parallel software essentially relies on the development of a robust software design. This applies to the overall architecture of a program, but also for the lower layers in a software system where concurrency is defined. Programmers require in-depth knowledge of their program's structure and behavior at all granularity levels. This task is exceedingly challenging without tools that systematically analyze the software design to rate their strengths and weaknesses for potential parallelism.

Industrial software projects diverge in terms of architectural and implementation styles. Selecting the concrete software architecture largely depends on the experience and preferences of project members. Analysis tools - such as Parceive - must be able to handle such high variability. Thus, we consider software architecture as a model comprising multiple user-defined viewpoints of the program's software entities. Each viewpoint represents a related set of software entities and their relations at different levels of abstraction, i.e., at architecture level, design level, and implementation level (Figure 3.1). The actual views are visualized by

a collection of software artifacts subject to custom architecture rules. These rules map the knowledge of the architects and programmers to extract a concrete perspective of the software architecture as implemented.

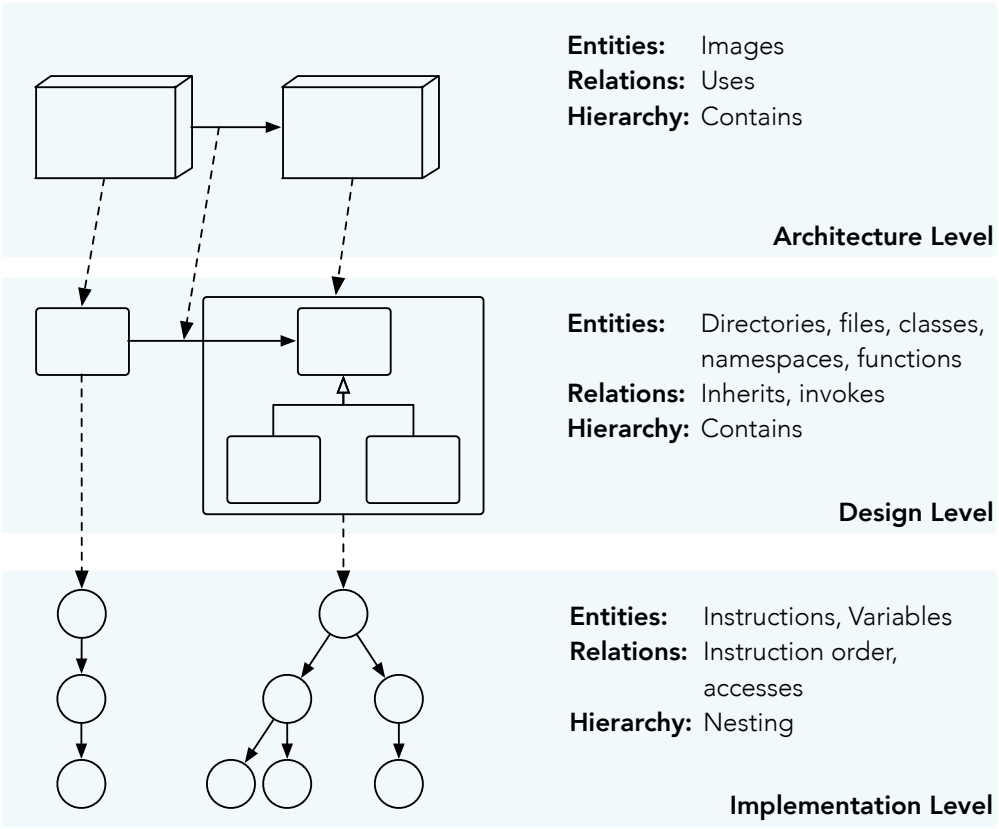


Figure 3.1: Abstraction levels with corresponding entity and relation types. Dotted arrows indicate mappings between entities and relations from different abstraction levels.

We extract software artifacts by using Parceive's ability to parse debug symbols. Those include data about various software entities and their relationships. Once the tool identified the entities, it maps them to new software artifacts according to the given architecture rules. The entity hierarchy is maintained by allowing artifacts to contain sub-artifacts. In addition, artifacts can be grouped together to form higher-level artifacts. Using the debug symbols as an information source has two major advantages. First, it allows the reconstruction of software architectures without access to the entire source code, which is a common requirement in industrial environments that use many third-party libraries. Second, the symbol data contains data generated during compilation. This avoids filtered artifacts from compiler optimizations or it adds artifacts introduced by generic programming techniques (e.g., template instantiation).

In this section we present our methodology for rule-based architecture recovery. We start with a description of the used algebraic formalism. Then, we show the operations and semantics of the proposed rules before presenting examples.

3.2.1 Binary Relational Algebra

We formalize our declarative expression language by Tarski's binary relational algebra [113]. The algebra centers on three elements: sets, binary relations, and typed graphs. We assume the reader is familiar with

the basic concepts and notations of set theory, but we will describe the latter two elements. Binary relational algebra matches favorably with software architecture extraction and transformation tasks due to its concise formalism for graph representations [58]. The presented models are derived from the models presented by Riva [102] and Kazman [12].

Binary Relations

A *binary relation* R, or simply a *relation*, from X to Y is a subset of the Cartesian product $X \times Y$. We define the relation by enumerating its elements, e.g., $R = \{(f,g),(g,h)\}$.

A relation can be represented as a directed graph. Assuming V is a set of elements, called *vertices*, and $R \subseteq V \times V$ is a relation. Then, we define $G = (V,R)$ as the directed graph of R. The elements in R are called *edges* and are directed from x to y for $(x,y) \in R$.

The operations for *union*, *intersection*, and *subtraction* for binary relations are inherited from set theory:

$$R_1 \cup R_2 = R_1 + R_2 := \{(x,y) | (x,y) \in R_1 \vee (x,y) \in R_2\} \quad \text{(union)}$$
$$R_1 \cap R_2 = R_1 \wedge R_2 := \{(x,y) | (x,y) \in R_1 \wedge (x,y) \in R_2\} \quad \text{(intersection)}$$
$$R_1 \setminus R_2 = R_1 - R_2 := \{(x,y) | (x,y) \in R_1 \wedge (x,y) \notin R_2\} \quad \text{(subtraction)}$$

The *inverse* of a relation R, denoted R^{-1}, is obtained by reversing its tuples:

$$R^{-1} := \{(y,x) | (x,y) \in R\} \quad \text{(inverse)}$$

We define the operations for selecting the *domain* and *range* of a relation R:

$$dom(R) := \{x | (x,y) \in R\} \quad \text{(domain)}$$
$$ran(R) := \{y | (x,y) \in R\} \quad \text{(range)}$$

The *relational composition* of R_1 and R_2, denoted by $R_1 \circ R_2$, enables the subsequent composition of different relations. Given the edges of $R_1 + R_2$, the relation $R_1 \circ R_2$ is the set of all edges that can be drawn by following an edge from R_1 and an edge from R_2. The definition is:

$$R_1 \circ R_2 := \{(x,y) | \exists z : (x,z) \in R_1 \wedge (z,y) \in R_2\} \quad \text{(relational composition)}$$

Relational composition is associative, and it is possible to compose a relation n times: $R \circ \cdots \circ R$ (i.e., R^n).

The *identity relation* of a set X, denoted by Id_X, is the relation:

$$Id_X := \{(x,x) | x \in X\} \quad \text{(identity relation)}$$

Considering the directed graph $G = (V,R)$, we assume that the identity relation is calculated on the set of vertices V and we simply write Id. By definition, we have $R^0 = Id$. The identity relation of a graph represents the set of all edges that loop directly back to the vertices. The identity relation has the property $R \circ Id = Id \circ R = R$.

Next, we define the *transitive closure* (*reflexive transitive closure*) of a relation R, denoted by R^+ (R^*) as:

$$R^+ := \bigcup_{i=1}^{\infty} R^i \qquad \text{(transitive closure)}$$

$$R^* := R^0 + R^+ = Id + R^+ \qquad \text{(reflexive transitive closure)}$$

Given a directed graph $G = (V,R)$, the transitive closure is the set of all edges that connect the vertices that are reachable by following the edges in R.

We also define a set projection, which effectively transmits the elements of a set S through a relation R to determine another set:

$$S \circ R = \{y | \exists x \in S : (x,y) \in R\} \qquad \text{(forward projection)}$$

$$R \circ S = \{x | \exists y \in S : (x,y) \in R\} \qquad \text{(backward projection)}$$

Typed Graphs

A *typed graph* is a graph with distinguishable types of edges. Given a set of vertices V, and the relations $R_1, R_2, \cdots, R_n$, we can define a typed graph as the set $G = (V, R_1, R_2, \cdots, R_n)$. The different types of edges are often visualized with different colors or line types. For a typed graph, we have the following properties:

$$dom(T) = V$$
$$\forall i : R_i \subseteq V \times V$$

A *hierarchical typed graph* is a typed graph with a *containment relation* C. Similar to the above definition of a typed graph, we can define a typed graph as the set $H = (V, C, R_1, R_2, \cdots, R_3)$, or simply $H = (V, C, R)$.

Given a hierarchical typed graph $H = (V, C, R)$, we define several relations:

$$P(C) = C^{-1} \qquad \text{(parent)}$$

$$S(C) = P(C) \circ C - Id \qquad \text{(sibling)}$$

$$D(C) = C^+ \qquad \text{(descendant)}$$

$$D_0(C) = C^* \qquad \text{(reflexive descendant)}$$

$$A(H) = P(C)^+ \qquad \text{(ancestor)}$$

$$A_0(C) = P(C)^* \qquad \text{(reflexive ancestor)}$$

Given a hierarchical typed graph $H = (V, C, R)$, we define the *lifting* operation, denoted by $R \uparrow C$, and its inverse operation *lowering*, denoted by $R \downarrow C$, as follows:

$$R \uparrow C = \left\{(x,y) | \exists a,b : (a,b) \in R \wedge (x,a) \in C^+ \wedge (y,b) \in C^+ \wedge x \neq y\right\} \qquad \text{(lifting)}$$

$$R \downarrow C = \left\{(x,y) | \exists a,b : (a,b) \in R \wedge (a,x) \in C^+ \wedge (b,y) \in C^+ \wedge x \neq y\right\} \qquad \text{(lowering)}$$

The lifting operator specifies the set of edges between two vertices that are abstracted from relations between any of their respective descendants. We also define the operations of *left lifting* and *right lifting*, denoted as $R \upharpoonleft C$ and $R \upharpoonright C$, respectively. They are defined as follows:

$$R \upharpoonleft C = \left\{(x,y) | \exists a : (a,y) \in R \wedge (x,a) \in C^+ \wedge x \neq y\right\} \qquad \text{(left lifting)}$$

$$R \upharpoonright C = \left\{(x,y) | \exists b : (x,b) \in R \wedge (y,b) \in C^+ \wedge x \neq y\right\} \qquad \text{(right lifting)}$$

We implemented left and right lifting into our software architecture visualization to abstract and refine dependency relations for various software structures.

To provide examples for the operator definitions, we use a hierarchical typed graph $H = (V,C,F,D)$ comprising the vertices $V = \{p,a,b,f,g,h,o,v,c\}$ and the following relations as edges (Figure 3.2):

$C = \{(p,a),(p,b),(a,f),(a,g),(b,h),(b,o),(o,c),(o,v)\}$
$F = \{(f,g),(f,c),(g,h)\}$
$D = \{(h,v),(o,v)\}$

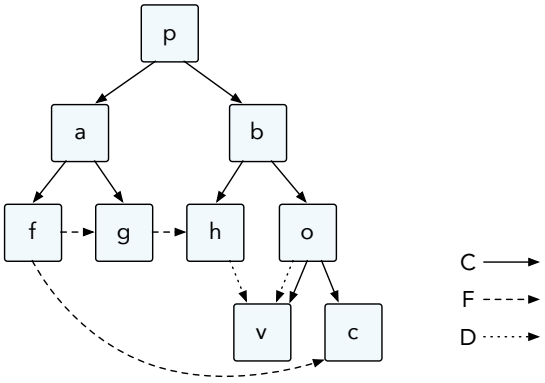


Figure 3.2: An example for a typed graph represented as containment tree [58].

We provide an example of various relations on the graph $H = (V,C,F,D)$:

$$
\begin{aligned}
F + D &= \{(f,g),(f,c),(g,h),(h,v),(o,v)\} \\
D \wedge C &= \{(o,v)\} \\
C - D &= \{(p,a),(p,b),(a,f),(a,g),(b,h),(b,o),(o,c)\} \\
F \circ D &= \{(g,v)\} \\
F^{+} &= F + \{(f,h)\} \\
F^{*} &= F^{+} + \{(f,f),(g,g),(h,h)\} \\
P(C) &= \{(a,p),(b,p),(f,a),(g,a),(h,b),(o,b),(v,o),(c,o)\} \\
S(C) &= \{(a,b),(f,g),(h,o),(v,c)\} \\
D(C) &= C + \{(p,f),(p,g),(p,h),(p,o),(p,v),(p,c),(b,v),(b,c)\} \\
F \uparrow C &= \{(a,b)\} \\
F \upharpoonright C &= \{(a,c),(p,c),(a,h),(p,h)\} \\
F \restriction C &= \{(f,o),(f,b),(f,p),(g,b),(g,p)\} \\
(\{(a,b)\} \downarrow C) \cap F &= \{(f,c),(g,h)\}
\end{aligned}
$$

3.2.2 Formalization of Software Architectures

Before we present the syntax of our rule-based language, we introduce a framework for formalizing software architectures. This Framework is based on the presented binary relational algebra. We define a *software architecture*, or simply an *architecture A* as a non-empty, finite set of vertices V, a containment relation C,

and a set of relations $R = (R_1, R_2, \cdots R_N)$:

$$A = (V, C, R) \quad \text{(architecture)}$$

where $C \subseteq V \times V$ and $R_i \subseteq V \times V (i \in \mathbb{N}_0)$.

From now, we call the vertices in an architecture *artifacts* to emphasize the representative character of extracted software elements. For our purposes, we specify the artifacts of an architecture as

$$V = V_V \cup V_F \cup V_C \cup V_N \cup V_{Fi} \cup V_I, \text{ where}$$

- V_V represents variables (e.g., local stack variables or heap variables).
- V_F represents functions (e.g., global functions or class methods).
- V_C represents classes or structures (including nested classes).
- V_N represents namespaces (including nested namespaces).
- V_{Fi} represents source files.
- V_I represents compiled image files (e.g., executables or shared library files).

Additionally, we use the relation

$$R = (R_I, R_A, R_C, R_T, R_D), \text{ where}$$

- $R_I \subseteq V_C \times V_C$ represents inheritance between classes.
- $R_A \subseteq V \times V_V$ represents variable accesses.
- $R_C \subseteq V_F \times V_F$ represents function invocations.
- $R_T \subseteq V \times V$ represents relations between instantiated software elements from the same template.
- $R_D \subseteq V \times V$ represents relations between duplicated software elements.

3.2.3 Rules for SAR

We now present our provided rules to extract and construct software architecture representations. These rules are based on regular expressions, i.e., the formal language $\mathscr{L}(\texttt{regex})$ defined over characters of alphabet Σ. $\mathscr{L}(\texttt{regex})$ is specified as

1. $\mathscr{L}(\varnothing) = \emptyset$
2. $\forall a_i \in \Sigma : \mathscr{L}(a_i) = \{a_i\}$
3. Let x and y be regular expressions, then:

$$\mathscr{L}(x|y) = \mathscr{L}(x) \cup \mathscr{L}(y)$$
$$\mathscr{L}(xy) = \{\alpha\beta | \alpha \in \mathscr{L}(x) \wedge \beta \in \mathscr{L}(y)\}$$
$$\mathscr{L}(x^*) = \bigcup_{i \geq 0} \mathscr{L}^i(x)$$

The extraction rules rely on the function $\mu(V) \rightarrow \Sigma^*$ that matches an artifact to its symbol name. More concretely, we require one function for each subset of V, i.e., μ_V, μ_F, μ_C, μ_N, μ_{Fi}, μ_I.

For clarity, we omit relations for rule definitions. Furthermore, we use recursive definitions based on several sub-rules for high expressiveness and conciseness. Note that these recursive rules comply with our implementation.

`variable` Rule

Given an architecture $G = (V,C)$, the definition of the `variable` rule is as follows:

$$\begin{aligned}\texttt{variable}(\mathit{regex}) = (N,E) \subseteq (V,C), \text{ where}\\ N = \{v | v \in V_v \wedge \exists s \in \mathscr{L}(\mathit{regex}) : (s,v) \in \mu_V\}\\ E = \emptyset\end{aligned}$$

The rule calcualtes an artifact containing separate artifacts for every variable which name matches *regex*. Resulting variable artifacts have no containment relation between them.

`function` Rule

Given an architecture $G = (V,C)$, we define $\mathscr{F}$ as a relation for recursive reuse as follows:

$$\begin{aligned}\mathscr{F}(N_F) = (N_V,E) \subseteq (V,C), \text{ where}\\ N_V = (N_F \circ C) \cap V_V\\ E = (N_F \times N_V) \wedge C\end{aligned}$$

The rule adds artifacts of containing variables to a function artifact. Now we can define the `function` rule:

$$\begin{aligned}\texttt{function}(\mathit{regex}) = (N_F \cup N,E) \subseteq (V,C), \text{ where}\\ N_F = \{f | f \in V_F \wedge \exists s \in \mathscr{L}(\mathit{regex}) : (s,f) \in \mu_F\}\\ (N,E) = \mathscr{F}(N_F)\end{aligned}$$

The `function` rule specifies an artifact that represents all functions which name matches *regex*, and all artifacts of local variables that are contained in these functions.

`class` Rule

Given an architecture $G = (V,C)$, we define $\mathscr{C}$ as a relation for recursive reuse as follows:

$$\begin{aligned}\mathscr{C}(N_C) = (N,E) = (N_M \cup N_V \cup N_F, E_C + E_F) \subseteq (V,C), \text{ where}\\ N_M = (N_C \circ C) \cap V_F\\ N_V = (N_C \circ C)) \cap V_V\\ E_C = ((N_C \times N_C) + (N_C \times N_M) + (N_C \times N_V)) \wedge C\\ (N_F,E_F) = \mathscr{F}(N_M)\end{aligned}$$

The rule adds artifacts of containing methods and member variables to a class artifact. Now we can define the the `class` rule:

$$\begin{aligned}\texttt{class}(\mathit{regex}) = (N_C \cup N,E) \subseteq (V,C), \text{ where}\\ N_C = \{c | c \in V_C \wedge \exists s \in \mathscr{L}(\mathit{regex}) \wedge \exists z \in V_C : (s,z) \in \mu_C \wedge c \in \{z\} \circ C^*\}\\ (N,E) = \mathscr{C}(N_C)\end{aligned}$$

The `class` rule specifies an artifact that represents all classes and structures which name matches *regex*, and artifacts of their nested classes. Additionally, for each (nested) class artifact the rule implicitly adds artifacts that represent its contained member variables, its methods, and their local variables.

`namespace` Rule

Given an architecture $G = (V, C)$, we define $\mathscr{N}$ as a relation for recursive reuse as follows:

$$\begin{aligned}
\mathscr{N}(N_N) = (N, E) = (N_C \cup N_F \cup N_V \cup N_1 \cup N_2, E_N + E_1 + E_2) \subseteq (V, C), \text{ where} \\
N_C = (N_N \circ C^*) \cap V_C \\
N_F = (N_N \circ C) \cap V_f \\
N_V = (N_N \circ C) \cap V_v \\
E_N = ((N_N \times N_N) + (N_N \times N_C) + (N_N \times N_F) + (N_N \times N_V)) \wedge C \\
(N_1, N_1) = \mathscr{C}(N_C) \\
(N_2, N_2) = \mathscr{F}(N_F)
\end{aligned}$$

The rule adds artifacts of containing classes, functions, and variables to a namespace artifact. Now we can define the `namespace` rule:

$$\begin{aligned}
\texttt{namespace}(regex) = (N, E) = (N_N \cup N, E) \subseteq (V, C), \text{ where} \\
N_N = \{n | n \in V_N \wedge \exists s \in \mathscr{L}(regex) \wedge \exists z \in : (s, z) \in \mu_N \wedge n \in \{z\} \circ C^*\} \\
(N, E) = \mathscr{N}(N_N)
\end{aligned}$$

The `namespace` rule specifies an artifact that represents all namespaces which name matches *regex*, and their nested namespaces. Additionally, for each (nested) namespace the rule implicitly adds artifacts that represent its contained classes, functions, and variables. For each class artifact the contained member variables, methods, and their local variables are added as separate sub-artifacts.

`inFile` Rule

Given an architecture $G = (V, C)$, we define $\mathscr{G}$ as a relation for recursive reuse as follows:

$$\begin{aligned}
\mathscr{G}(N_G) = (N_N \cup N_1 \cup N_2, E_G + E_1 + E_2) \subseteq (V, C), \text{ where} \\
N_N = (N_G \circ C^*) \cap V_N \\
E_G = (N_G \times N_N) \wedge C \\
(N_1, E_1) = \mathscr{N}(N_G) \\
(N_2, E_2) = \mathscr{N}(N_N)
\end{aligned}$$

The rule adds artifacts of containing namespaces, classes, functions, and variables to a higher-level artifact. Note that we use $\mathscr{G}$ (for *generic*) since we use the same relation below. Now we can define the `inFile` rule:

$$\begin{aligned}
\texttt{inFile}(regex) = (N_{Fi} \cup N, E) \subseteq (V, C), \text{ where} \\
N_{Fi} = \left\{ f | f \in V_f \wedge \exists s \in \mathscr{L}(regex) : (s, f) \in \mu_{Fi} \right\} \\
(N, E) = \mathscr{G}(N_{Fi})
\end{aligned}$$

The `inFile` rule specifies an artifact that represents all artifacts from a source file that matches *regex*. For each file, this rule implies its (nested) namespaces, (nested) classes, functions, and (local) variables. Additionally, for each class the contained member variables, methods, and their local variables are added.

`image` Rule

Given an architecture $G = (V, C)$, we define the `image` rule as follows:

$$\begin{aligned} \texttt{image}(\mathit{regex}) &= (N_I \cup N, E) \subseteq (V, C), \text{ where} \\ N_I &= \{i | i \in V_I \wedge \exists s \in \mathscr{L}(\mathit{regex}) : (s, i) \in \mu_I\} \\ (N, E) &= \mathscr{G}(N_I) \end{aligned}$$

The rule specifies an artifact that represents an image file (e.g., an executable or a shared library file) which name matches *regex*. For each image, the rule implies its contained (nested) namespaces, (nested) classes, functions, and (local) variables. Additionally, for each class artifact the contained member variables, methods, and their local variables are added as representative sub-artifacts.

Operator Rules

We now present our provided operators for software architecture representations. Given two artifacts A_1, and A_2, we define the *union* operator $||$ as follows:

$$A_1||A_2 = (N_1, E_1)||(N_2, E_2) = (N_1 \cup N_2, E_1 + E_2) \quad \text{(union)}$$

The shown operator specifies the union of the sub-artifacts from A_1 and A_2, and the union of their containment edges. Similarly, the *intersection* operator && is defined as follows:

$$A_1 \&\& A_2 = (N_1, E_1) \&\& (N_2, E_2) = (N_1 \cap N_2, E_1 \wedge E_2) \quad \text{(intersection)}$$

The operator produces the intersection of the sub-artifacts from A_1 and A_2, and the intersection of their containment edges. The *negation* operator ! on an artifact A is defined as follows:

$$!A = !(N, E) = (V \setminus N, C - E) \quad \text{(negation)}$$

The operator specifies all artifacts from V subtracted by those in A, and all containment edges in C that are not in E. Finally, the *abstraction* operator is defined as follows:

$$\begin{aligned} [A_1, \cdots, A_N] &= [(N_1, E_1,), \cdots, (N_N, E_N)] \quad \text{(abstraction)} \\ &= (N_A \cup N_1 \cup \cdots \cup N_N, E_A + E_1 + \cdots + E_N), \text{ where} \\ N_A &= \{T\} \\ E_A &= E_{A1} + \cdots + E_{AN} \\ E_{Ai} &= \{(T, x) | x \in N_i, \forall y \in N_i : (x, y) \in E_i^*\} \text{ for } i \in [1, N] \end{aligned}$$

The operator produces a new artifact that includes the operands as sub-artifacts. It can be used for SAR approaches that follow a bottom-up procedure.

Implicit rules

During reconstruction, Parceive automatically applies implicit architecture rules to artifact hierarchies. This ensures an effective creation of detailed architecture models without requiring precise specifications. Implicit rules establish groupings and relations between sub-artifacts for the following structures:

- *Duplicates.* Parceive's SAR module can create artifact hierarchies with multiple nodes that represent the same software elements. Rather than completely avoiding such duplicates, we add a relation to R_D to define nodes with the same elements. Our architectural visualization highlights duplicates with distinct (dashed) edges.
- *Sub-Elements.* As presented above, explicit architecture rules use recursive sub-rules to include children artifacts, such as the `function` rule that is implicitly applied by the `class` rule.
- *Inheritance.* Parceive considers class inheritance as relation between two class artifacts. Our architectural visualization shows inheritance edges, similar to UML. Let $A = (N, E, X)$ be an artifact hierarchy with nodes, containment relations, and other relations, respectively; then, the implicit inheritance rule `inh` is defined as follows:

$$\begin{aligned} \texttt{inh}(A) = \texttt{inh}(N,E,X) &= (N,E,X_I,X_A,X_C,X_T,X_D) \subseteq (V,C,R), \text{ where} \\ X_I &= (N \times N) \wedge R_I \end{aligned}$$

- *Templates.* The C++ programming language supports generic programming via templates. We group instantiated software elements from the same template into single artifacts. Let $A = (N, E, X)$ be an artifact hierarchy with nodes, containment relations, and other relations, respectively. Let $\mathscr{T}(R) \rightarrow V \cup \emptyset$ map a template relation to a unique template node representing the template. Then, the implicit template rule `templ` is defined as follows:

$$\begin{aligned} \texttt{templ}(A) = \texttt{templ}(N,E,X) &= (N_T, E_T + E_{NT}, X) \subseteq (V,C,R), \text{ where} \\ N_T &= N \cup \{\mathscr{T}(t) | t \in X_I\} \\ E_T &= \{(\mathscr{T}(t), x) | t \in X_I \wedge x \in N \wedge \exists y \in N : t = (x,y) \vee t = (y,x)\} \\ E_{NT} &= E - \{(x,y) | (x,y) \in C \wedge \exists z : (y,z) \in X_I\} \end{aligned}$$

3.2.4 Our SAR Language

We now present our specification language for declaring architecture rules. This language is based on a context-free grammar G, defined as $G = (N, T, P, S)$, where

1. N is a finite set of nonterminal characters (e.g., variables). They define a sub-language specified by G. We include the nonterminals <*specification*>, <*statement*>, <*expression*>, <*set_expression*>, <*atom*>, <*binary_op*>, and <*keyword*>.
2. T is a final set of terminals. We define *variable*, *function*, *class*, *namespace*, *inFile*, and *image*. Furthermore, we interpret <*artifact*> and <*regex*> to represent any valid artifact name and regular expression, respectively.
3. $P \subseteq N \times (N \cup T)^*$ are production rules that describe how each nonterminal is defined in terms of terminals and nonterminals. We describe our rules following the Backus-Naur-Form (BNF).
4. $S \in N$ is the start variable. We call this a <*specification*> nonterminal.

Production Rules

A specification written in our proposed language has the following form:

```
<specification>  ::= <statement> |
                     <statement> <specification>
```

Each statement defines a new top-level artifact with a given name as descriptor. The artifact is created according to an expression specified after the assignment operator `:=` or the definition operator `=` as follows:

```
<statement>  ::= <artifact> := <expression> |
                 <artifact>  = <expression>
```

The rules semantically differ only in their manifestation in the visualization. Artifacts created by the assignment operator are visualized whereas artifacts created by the definition operator are not visualized but can be used as intermediate artifacts in subsequent rules. We denote the application of the rules as *definition* and *assignment*, respectively. The `<expression>` nonterminal is specified by the following production rules:

```
<expression>  ::= <artifact> |
                  !<expression> |
                  <expression> <binary_op> <expression> |
                  (<expression>) |
                  [<set_expression>] |
                  <atom>
```

The `artifact` rule enables already defined artifacts to be used on the right-hand side of assignments and definitions. As mentioned, we support the unary negation operator `!`, and the binary intersection `&&` and union `||` operators. Their precedence follows the semantic counterparts from the set theory. However, the precedence can be changed by using parenthesis.

```
<binary_op> ::= && | ||
```

The the n-ary operator `[]` performs an abstraction of sub-artifacts into a new parent artifact. The rule is defined over a set of expressions as follows:

```
<set_expression>  ::= <expression> |
                      <expression>, <set_expression>
```

An atomic rule creates a set of artifacts representing software elements that comply with two conditions. First, the elements type must conform to the given rule's keyword. Second, the elements name must match the regular expressions within the preceding parentheses.

```
<atom>  ::= <keyword> (''<regex>'')
```

The nonterminal `<regex>` must be a valid regular expression and the `<keyword>` can be one of the following literals:

```
<keyword>  ::= variable | function | class | namespace | inFile | image
```

Note that the applied semantics of these literals correspond to the architecture rules from Section 3.2.3.

Example

Here, we demonstrate the power of the developed SAR approach using the following small program as input.

```
int global = 1;

template <typename T>
class Container {
    T value;
public:
    T get() { return value + global; }
    void set(T v) { value = v; }
};
```

Assume the `Container` class is instantiated with `int` and `double` types:

```
Container<int> c_i;
Container<double> c_d;
```

We extract a simple view on the program's architecture. It comprises the generic container logic in the `Container` class, the global variable, and the main function. For that purpose, we declare the following architecture rules:

```
classes := class(".*")
rest := inFile(".*") && !classes
```

The first rule specifies an artifact comprising one sub-artifact for each instantiated `Container` class. Note that the program contains no other classes; hence, we can use the regular expression `".*"` for matching. The second rule finds the difference between all software elements of the program (i.e., elements of all source files) and those with a mapping in the `classes` artifact.

The resulting artifacts are shown in Figure 3.3. Note that the sub-artifacts of the `Container` artifacts are derived from implicit architecture rules. The dashed arrows between the functions `get` and the `global` variable indicate their shared memory accesses. To identify these dependencies, we applied Parceive's dependency analysis (Section 3.3) on the visualized `global` artifact.

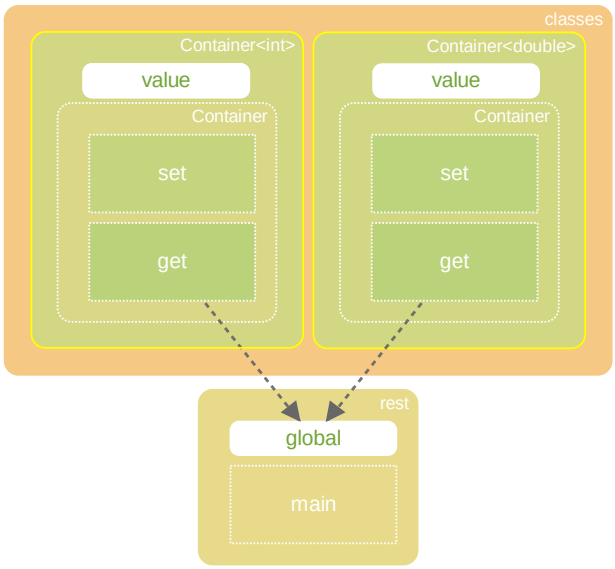


Figure 3.3: Architecture View showing artifact nodes and memory accesses of the global variable.

In another use case we may be interested in only the getters of the two instances and the global variable. In this case, we could use the following architecture rule:

```
getterAccess := function("get") || variable("global")
```

3.3 Interactive Software Parallelization

In this section we present our proposed parallelization procedure. It comprises five steps for iteratively transforming sequential applications into concurrent software systems (Figure 3.4). During the first two

steps, programmers explore the problem and program domain to identify potential scenarios for parallelization. Legacy systems are often inappropriately structured for common parallelization patterns [85]; thus, the third step is concerned with iterative software restructuring. Afterwards, relevant components must be parallelized using parallel computing libraries or language constructs. In the last step, programmers check if their changes are correct, and the parallel software produces the right results in less time than the sequential version (and/or the software design has improved). If this is not the case, another iteration of the procedure might be necessary.

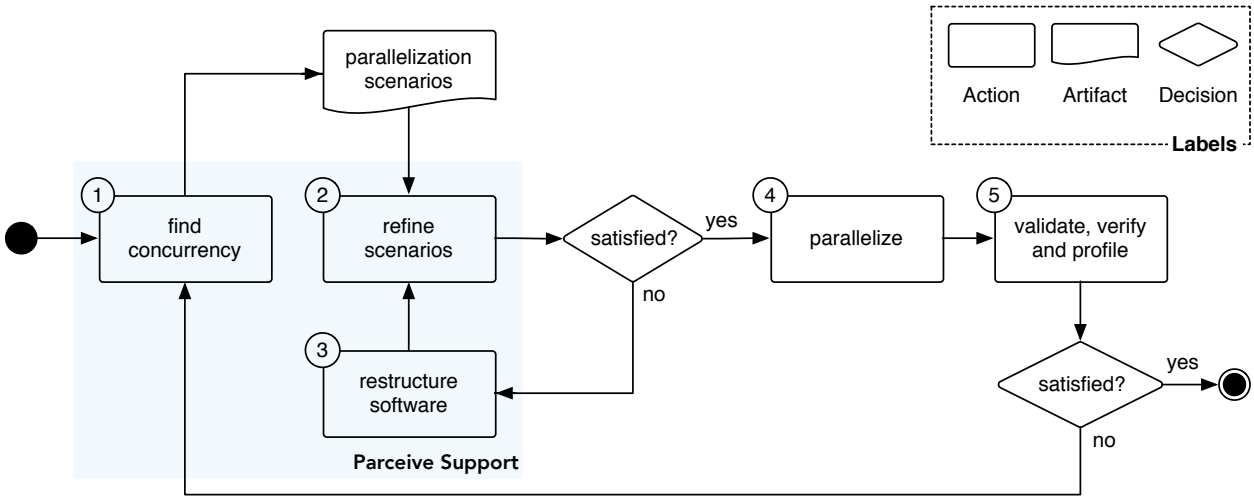


Figure 3.4: Our proposed parallelization methodology comprising five steps.

Parceive facilitates the first three steps. Several analyses and visualizations are provided for querying recorded program models to answer intricate questions relevant for parallelization.

In the remainder we explain the steps of our approach in more detail. For the first three steps, we present important features of Parceive that help programmers with typical use cases.

3.3.1 Finding Concurrency

The first step is concerned with identifying exploitable concurrency of software systems. Exploitable concurrency exists if computational problems can be decomposed into multiple sub-problems, ideally solvable at the same time to benefit from parallelism.

To find exploitable concurrency, programmers essentially require a good understanding of the problem domain, its high-level algorithmic issues, and the current software design. Focusing on the problem domain first facilitates elegant and scalable solutions. Those solutions may be missed by moving too soon into the program domain where parallelization scenarios are obscured by implementation details [87].

However, inspecting proven solutions encoded in existing software systems improves the programmer's understanding of given problems. Program executions reveal run-time characteristics that are instructive to comprehend software behavior. Reconstruction of the underlying architecture shows important components and dependencies between them. Knowing such dependencies is key to design parallel solutions.

Before parallelization, programmers must justify the required effort relative to any expected performance gains (Figure 3.5). Parallelization is only recommended if the problems contain enough exploitable concurrency and a potential program restructuring does not sacrifice other important software qualities.

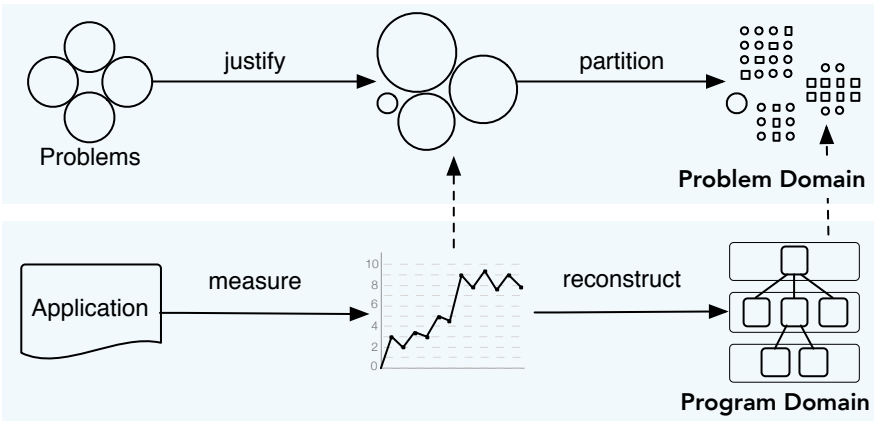


Figure 3.5: The finding concurrency step of our approach.

If parallelization is justified, programmers partition relevant problems into small computational elements that constitute so-called *parallelization scenarios*. Parallelization scenarios are the building blocks for parallel software and represent instances of parallel design patterns in the given application. Having many of those scenarios at different levels of abstraction helps to build parallel software designs with high scalability.

Justification

To justify parallelization effort, programmers evaluate the size and relevance of computational problems. Restructuring large-scale software applications and adding behavioral complexity by parallelism is tedious and error-prone; thus, weighing the potential benefits is crucial. Benefits of parallelization are improved performance (e.g., throughput or scalability), and software architectures that are more expressive and problem-oriented by considering concurrency as first-class design element.

Note that some problems are inherently serial. Their solutions are encoded by sequences of operations that subsequently rely on each other, or limit parallelism by requiring scarce resources. Examples are finite state machines that check the acceptance of input strings. Naive solutions concurrently operate on different parts of the input string. The problem is to find the right starting point for each part, which depends on the final state of the previous part. Parallelization of such problems may not be worth the effort relative to low efficiency and the added behavioral complexity (e.g., by using parallel solutions based on speculative prefix computations [94]).

Partitioning

After parallelization has been justified, programmers conceptually partition the computational problems into sub-problems. Partitioning can be conducted on data level (*data decomposition*) or task level (*task decomposition*). Here, tasks are individual sequences of instructions associated with few specific sub-problems. Data decomposition is implied by the tasks operating on the respective data chunks, or vice versa. Thus, separating task and data decomposition is often impractical; although a distinct consideration may lead to different parallelization scenarios.

A critical property of problem decompositions are dependencies between the resulting sub-problems. Those dependencies impact feasibility and performance of parallelization scenarios. Sometimes, partitioning naturally produces independent tasks or data chunks (sometimes called “embarrassingly parallel” problems [7]); in other cases, decomposition requires more effort, such as adding communication and synchronization between tasks.

Task decomposition focuses on computations and defines solutions as a collection of tasks that can be executed simultaneously. Amount and granularity (e.g., the number of operations) of tasks are significant for the efficiency of parallelization scenarios. Depending on the target architecture, many small tasks improve the exploitable concurrency but imply unfavorable overhead for concurrency management; few big tasks may result in optimal speedup with few processing units but do not scale well with larger amounts.

Three requirements are relevant to achieve good scalability relative to task decomposition. First, the number of tasks should dynamically adapt relative to the problem size and (ideally) to the target hardware architecture. Solutions which require manual configurations are non-portable and prevent quick refinement iterations. Second, the workload of multiple tasks should be similar for good load balance. Programmers rely on tools to measure those workloads and estimate the resulting performance. This information is necessary for refining task granularities to balance the workloads, accordingly. And third, operations of different tasks should be largely independent. If dependencies are induced within the problem domain or are required to satisfy user requirements, managing them should only take a fraction of the solution's execution time.

Favorable examples for task decomposition are algorithms that apply graphical filters on images. Typical parallelization scenarios perform the unmodified algorithms on multiple input images simultaneously to achieve higher throughput.

Data decomposition approaches problems differently and lead to complementary solutions than task decomposition. The decomposed data may origin from input data, output results computed by the algorithms, or their intermediate values. In cases with largely independent data chunks, the right partitioning is obvious; but often the decomposition requires substantial data refactoring if the memory is physically distributed or has complex dependencies.

The partitioning should be flexible enough to support multiple parallelization scenarios of the parallel design. Often, major data structures that constitute computational problems are involved in many scenarios or different phases of one scenario. Their partitioning should lead to scalable solutions with reduced run time and/or memory utilization. Programmers achieve this requirement by dividing application's data into chunks of equal size. Multiple chunks with different sizes imply less parallelism due to unbalanced workloads.

The next step is to partition computations performed on that data into tasks. Operations of different tasks may operate on the same data and the accesses must be managed by synchronization or communication. Different phases of the used algorithms may operate on different data structures or demand different decompositions of the same data structure; thus, single parallelization scenarios may contain multiple data decompositions at different levels of granularity.

The image filtering example from above may be solved with a parallelization scenario using data decomposition by partitioning the image data into smaller rectangles and concurrently apply slightly modified algorithms on them.

3.3.2 Finding Concurrency with Parceive

Parceive helps programmers during the justification and decomposition step. The tool uses collected analysis data to provide visualizations operating on different levels, ranging from function executions and loops to structural elements with arbitrary granularity.

Programmers must evaluate the size of their computational problems to justify parallelization. Besides domain knowledge, this requires a good understanding of possible implementations in the solution domain. Solutions encoded in existing software systems are invaluable to improve the understanding. Here, programmers analyze the program's characteristics, such as detailed execution times and their correlations to

input. This allows speedup and scalability estimations of parallelization scenarios by relying on formal execution models, such as the laws of Amdahl or Gustafson. A pragmatic approach is to focus on parts that are computationally intensive first. Understanding these parts, their key data structures, and the mechanisms that use this data is crucial to define good decompositions.

Parceive facilitates justification of parallelization scenarios by providing the Trace, CCT, and Architecture View. These visualizations highlight run-time proportions of executed functions and higher-level artifacts reflected by the coloring of their visual elements. The colors range from green to red for indicating low to high execution time, relative to the overall execution time.

Trace View

The Trace View is a hierarchical and interactive representation of a program execution at the level of called functions and loop iterations. Its primary purpose is to help programmers spotting parallelization scenarios and as navigation means for other interactive visualizations. It points to the worthwhile regions for parallel processing and depicts relevant functional execution patterns.

The Trace View supports two visualization modes: tracing and profiling. The tracing mode represents an icicle plot [70], augmented with loop information, i.e., a hierarchical view of function calls made during program execution (Figure 3.6). The calls are arranged chronologically from left to right, and the visibility of loop information can be toggled on or off. Trace visualization is useful for a detailed examination of a program's execution, especially when the invocation order is important. The profiling mode presents the user a hierarchical view of the functions called during execution. The length of each function node reflects the aggregated execution time of its calls, which is often sufficient to quickly pinpoint hot spots.

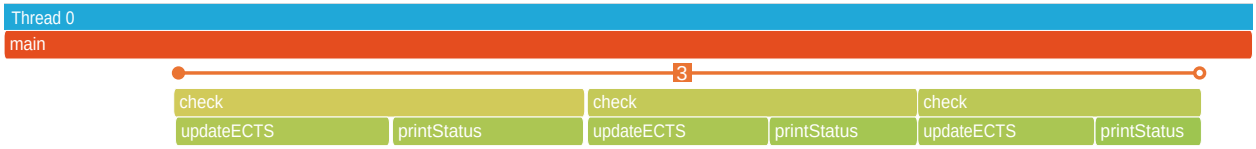
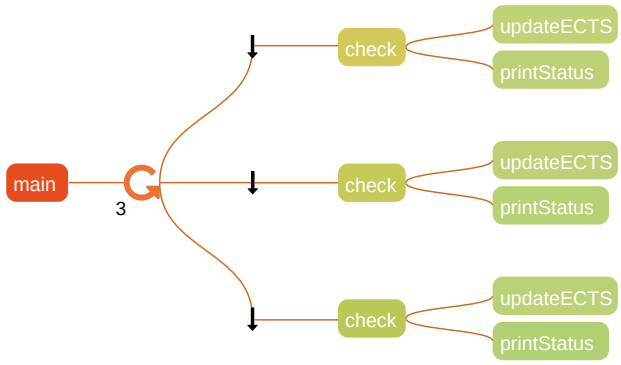


Figure 3.6: The Trace View showing call nodes and a loop of a small example in a timeline.

As program's execution time and complexity increases, so does the quantity of its performance data, making the data harder to digest at once. For this reason, the Trace View provides zooming at call level and execution time filtering. The latter sets the minimum execution time required for a call to be visualized. The minimum value depends on the percentage of the execution time of the current top-level call in the view. This only allows the rendering of calls coarse enough to be easily visible. With the call zooming feature, users can focus on a single call node, which sets it as the new top-level node, recomputes a new minimum value, and loads child calls of the focused call that weren't displayed before.

For our small example program, the Trace View indicates that most run time (~80%) was spent within a loop containing three executions of function `check` (Figure 3.6). Each of these executions subsequently calls the functions `updateECTS` and `printStatus`, with roughly the same run time share. Assuming no dependencies between the checks, a rewarding parallelization scenario would be to execute the three calls in parallel. Here, Amdahl's law provides the following (maximal) speedup:

$$Speedup \leq \frac{1}{(1-0.8)+\frac{0.8}{3}} \approx 2,14$$




```
63 int main(int argc, char** argv)
64 {
65   using namespace pcv;
66   Student studentA("tom");
67   Student studentB("jerry");
68
69   Exam exam1 { {Task(true, 2), Task(true, 3)}, &studentA, 10, 2 };
70   Exam exam2 { {Task(true, 2), Task(true, 3)}, &studentB, 10, 2 };
71   Exam exam3 { {Task(true, 2), Task(true, 3)}, &studentA, 10, 2 };
72
73   std::vector<Exam> exams { exam1, exam2, exam3 };
74
75   for (auto &exam : exams) {
76     check(exam);
77   }
78
79   return 0;
80 }
81
```

a. CCT View showing nodes for function calls and loops.

b. Source View.

Figure 3.7: The CCT View showing the example from Figure 3.6.

Calling Context Tree (CCT) View

The CCT View targets comprehension of the application's dynamic behavior. It displays a calling context tree [5] comprising nodes that represent function calls, loops and loop iterations, and memory locations (Figure 3.7 a). The visualization positions nodes according to a horizontal tree layout. Child nodes are vertically sorted by their relative start time to reflect the execution order. Memory nodes can be accessed by arbitrary tree nodes, which might lead to entangled edges in the visualization. Thus, they are not integrated in the tree layout but aligned with an unconstrained layout based on a physical force simulation (Figure 3.10).

Initially, the visualization shows only the call node for the `main` function. Users can arbitrarily expand and collapse call nodes. When a function is called multiple times during the same function execution respective call nodes are merged to *call groups*. Call groups reduce the number of nodes to be displayed, but can also be decomposed into their single call nodes. Navigating through loop executions and loop iterations is similar to calls and allows the user to see information at any desired granularity.

The visualization provides some additional features that improve its scalability and support programmer's comprehension of their programs.

- *Expand and Collapse.* Expanding and collapsing nodes to show and hide their parent or child nodes.
- *Zoom and Pan.* Seamless zooming and panning to explore arbitrary levels of detail.
- *Focus.* Centering on selected tree nodes, triggered by other visualizations (e.g., the Trace View).
- *Spot.* Expanding selected tree nodes up to their common ancestor, triggered by other visualizations.

Architecture View

The Architecture View shows artifact hierarchies reconstructed from program binaries, according to the given architecture rules (Section 3.2). The visualization uses nested boxes and edges to represent artifacts and relationships between them, respectively. It relies on a constrained force simulation to arrange the boxes and edges for optimal visibility and clarity (Figure 3.8). Every rectangular box represents a single artifact including its name and its type symbol (e.g., function, variable, class, namespace or general artifact).

The visualization helps with parallelism justification by highlighting worthwhile program components relative to the spent execution time. Furthermore, it supports with the decomposition of computational problems by showing the structure of the current software design under different perspectives.

We implemented the following features to improve scalability and support the programmer's comprehension of their programs.

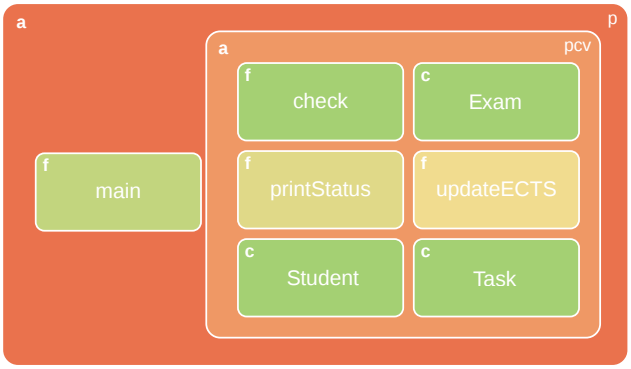


Figure 3.8: The Architecture View showing the artifacts of an extracted architecture.

- *Expand and Collapse.* Expanding and collapsing boxes to show and hide their sub-artifacts for software exploration. Dependency edges between leaf-nodes are lifted in both direction according to the presented left- and right-lifting operations (Section 3.2).
- *Zoom and Pan.* Seamless zooming and panning to explore the extracted architecture models.
- *Inheritance.* Automatically adding inheritance edges to indicate class inheritance relations, according to the implicit inheritance rule (Section 3.2).
- *Duplicates.* Automatically adding duplication edges to indicate multiple equal artifacts, according to the implicit duplication rule (Section 3.2).
- *Coloring.* The coloring indicates the aggregated run times of all executions of an artifact including the run times of its sub-artifacts.

3.3.3 Validate and Refine Scenarios

In the second step, programmers iteratively validate and refine previously identified parallelization scenarios. The objective is a parallel software design that satisfies the stakeholder's requirements and a list of work items to realize this design. During validation, concurrent computations are checked for conflicting data dependencies. During refinement, the parallelization scenarios are evaluated, ordered, and grouped relative to desired qualities, such as feasibility, efficiency, and portability.

Validation

The validation step focuses on the detection of dependencies between computational elements. Some data dependencies are accidentally caused by the sequential programming model (e.g., by re-using local variables). The programmer can easily resolve those dependencies by code refactoring. On the other hand, data dependencies that are inherent within the problem domain must be analyzed and evaluated relative to their impact on concurrency. If management of those dependencies (i.e., the cost for synchronization or communication) requires too much run time overhead, the respective parallelization scenarios must be adapted or filtered out. During validation, programmers further comprehend the limitations of the parallelization scenarios, the computational problems, and the existing software design.

Refinement

During refinement, programmers evaluate the resulting performance of individual parallelization scenarios and conceive favorable combinations of different scenarios. Each combination may involve conflicting design decisions. Programmers thus aim for balancing trade-offs during this step, which requires good comprehension of the problem and program domain. The results is a list of restructuring steps to facilitate parallelization. Furthermore, the gained knowledge leads to more precise or even new parallelization scenarios.

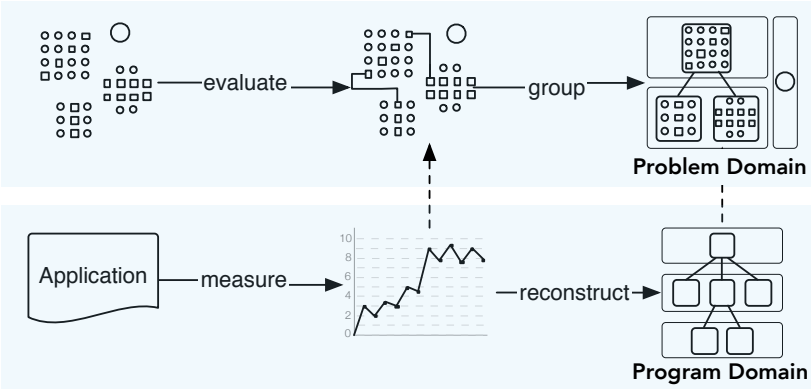


Figure 3.9: The refinement step of our approach for parallelization scenarios.

To evaluate the efficiency of parallelization scenarios, programmers rely on performance metrics or profiling techniques. Speedup estimations are then conducted by computing algorithm complexities or measuring run times of sequential applications, respectively. Parallelization approaches use these estimations to compute the scalability of parallelization scenarios by varying the input data or the number of processing units.

Special care must be taken for overhead incurred by concurrency management. For example, the run time required for context switches between threads, or necessary synchronization to manage dependencies. These overheads depend on the used programming models and hardware architectures; thus, speedup estimations must consider potential target platforms.

Another issue for efficiency evaluation is the resulting task size of parallelization scenarios. If computational tasks involve largely unequal workloads, scheduling them implies only little performance improvement. Furthermore, if the number of intended tasks do not scale well with the problem size, the overall scalability is limited. Thus, programmers aim for well-balanced solutions with high concurrency to improve the performance and flexibility of their parallel solutions.

After performance evaluation of single parallelization scenarios, programmers combine most promising scenarios to further increase the potential parallelism of the final design. If parallelization scenarios operate on different problems or orthogonal parts of the same problem, these scenarios can be used without adaptations. Otherwise, programmers must further analyze the dependencies to identify their root causes. Conflicting dependencies require communication or synchronization overhead; thus, scenarios with dependencies between the same software elements are easier to combine.

Finally, programmers must consider the abstraction level of different parallelization scenarios. In some cases, multiple scenarios interact favorable at different levels and provide more concurrency than single scenarios. In other cases, the scenarios are operating at the same levels and require different restructuring. Here, the granularity of different scenarios must be adjusted to complement each other. This is particularly important if different scenarios rely on the same supporting structures, such as task schedulers or concurrent containers.

3.3.4 Validate and Refine Scenarios with Parceive

Programmers use Parceive for explorative prototyping of parallelization scenarios. The provided visualizations enable construction, selection, and analysis of various software elements relative to existing call and data dependencies. We tailored two visualizations for this purpose. The CCT View shows dependencies at the function execution level. The Architecture View shows dependencies between structural elements.

CCT View

Data dependency analysis at function level is the primary use case of the CCT View. It supports validation and refinement of parallelization scenarios by incrementally improving the programmer's knowledge about parallelism-inhibiting dependencies. Here, the existence and the location of data dependencies between arbitrary regions in software is decisive. The visualization represents found dependencies by circular nodes that follow a dynamic force layout, independent of the call node's tree structure (Figure 3.10 a). We colorize memory nodes according to their type, i.e., blue nodes represent stack variables, yellow nodes represent static or global variables, and orange nodes represent memory locations in the heap storage.

The foundation for serious data dependency analysis is the identification and comprehension of the origins at instruction level. Programmers must understand the root causes for dependencies to treat them as accidental or inherent for parallelization scenarios. The CCT View provides dependency analyses on its nodes and shows additional information for every resulting memory node (Figure 3.10 b). This information contains data about the allocation locations for heap memory, and instructions that access memory locations (separated by read and write accesses). Parceive precisely tracks memory accesses at address level, which allows dependency analyses of different array elements or member variables. In case of stack variables and objects, the CCT View shows variable names.

a. A commonly accessed (heap) memory location. b. Detail view showing access information.

Figure 3.10: The CCT View after applying a multi-level dependency analysis on three call nodes.

The visualization provides the following features for adding memory nodes to the calling context tree:

- *Showing Accessed Memory.* The CCT View shows all memory nodes that represent memory locations accessed by single function executions or loop iterations.
- *Single-Level Data Dependency Analysis.* After the user selected multiple nodes, Parceive queries the program model for memory locations that were commonly accessed by at least two of these nodes. Parallelism is only concerned with true-, anti-, and output-dependencies; thus, we restrict the results to memory locations which values are modified during any related function execution. Note that the visualization implicitly enables users to distinguish the type of dependency (e.g., true- and anti-dependency) by the invocation order of the presented nodes.

- *Multi-Level Data Dependency Analysis.* This analysis type differs from the single-level analysis by inspecting not only single function executions or loop iterations, but whole call trees induced by all relevant executions. Parceive thus generates recursive queries to detect shared memory accesses across arbitrary deep call hierarchies. The visualization connects the nodes for shared memory locations to the selected top level nodes. This allows a clear visualization and largely reduces the amount of visual elements. However, the user can automatically expand all nodes of the call path that access memory locations of selected memory nodes.

Architecture View

The Architecture View helps programmers to validate and refine parallelization scenarios by showing relationships between artifacts. Thus, Parceive provides analyses to identify data and call dependencies between arbitrary artifact hierarchies. These dependencies are visualized by directed edges between corresponding artifact nodes (Figure 3.11). Whenever the user performs a new analysis the visualization gets rendered, i.e., the edges created by the previous analysis are removed to avoid confusing results.

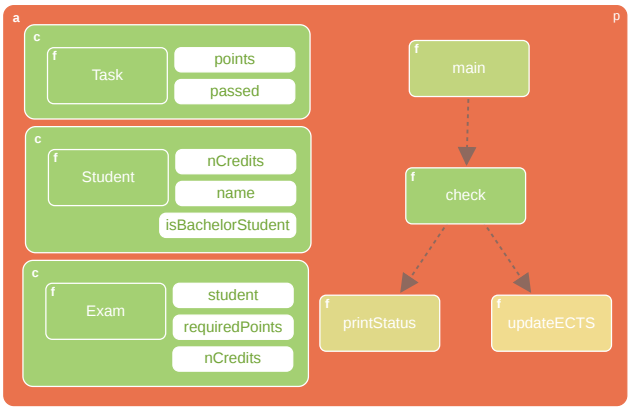


Figure 3.11: The Architecture View showing function call relationships as edges.

Note that architecture rules enable a multitude of perspectives on the given software system. This makes the visualization and analyses versatile for many use cases and matches with different viewpoints. For example, the user may be interested in dependencies between different software components. The architecture rules allow to slice these components according to several structuring techniques, such as object orientation, directories, or namespaces. This feature matches with the development view. In another typical use case users identify the dependencies between third-party libraries and their software system. The Architecture View let them easily visualize and analyze individual artifact hierarchies for every executable file. This matches with the deployment view. Finally, users can inspect the relationships between artifacts that were executed by different threads or processes. This matches with the process view.

The Architecture View uses the established mapping between artifacts and run-time information to generate sophisticated queries. These queries allow the following features for the validation and refinement of parallelization scenarios:

- *Detecting Artifact Dependencies.* The Architecture View provides two types of dependency analyses operating on an arbitrary artifact hierarchy: data dependency analysis and call dependency analysis. The former detects artifacts that access memory locations within the same hierarchy. The latter detects function call relations between functions of the same artifact hierarchy. Note that these analyses can be essentially applied on all shown artifacts of the software architecture by using the abstraction operator rule to create an abstract top artifact with all other artifacts as sub-components.

- *Detecting Selective Dependencies.* Detecting all relationships between every artifact in the given hierarchy can be computational expensive and lead to overwhelmingly many edges. We thus support dependency analyses between selected artifacts. The user selects two or more artifact nodes and applies the data or call analysis. Resulting edges are only computed between the selected artifact nodes and their respective sub-artifacts.
- *Detecting Backward Dependencies.* The user can perform a backward analysis to identify all relationships (calls and accesses) to an artifact and its sub-artifacts. As with selective dependency detection, programmers use this feature to efficiently analyze over a reduced search space and obtain precise results. If this analysis is performed on a variable node, edges from all shown artifacts that access the memory location are added. If this analysis is performed on a function node, all shown artifacts that call the function are added. If this analysis is performed on a higher-level artifact, it recursively applies the backward analysis on all contained sub-artifacts.
- *Lifting and Lowering* Dependencies between artifacts are lifted and lowered during exploration according to the relational operations presented in Section 3.2. Hence, our layout algorithm visualizes edges between artifacts of any hierarchy level. It uses a force simulation to align the boxes and edges. If the user further navigates through the artifact hierarchy, edges between appearing nodes are automatically refined. The endpoints of data (call) dependencies are variable (function) artifacts.
- *Abstraction.* Besides expanding artifacts naively, the Architecture View provides abstracted expansion of artifacts. Here, unrelated sub-artifacts for the currently lowered edges are abstracted to temporary artifacts showing their quantity as descriptor. All other (related) sub-artifacts are shown and attached with separate relation edges. This allows a clear view tailored for efficient dependency analysis, even with many software elements.
- *Heap Abstraction.* Manually allocated heap memory is not related to any program symbol (only to the allocated memory location); thus, the memory cannot be mapped to any visual artifact. To show data dependencies relative to such heap memory locations, the Architecture View adds a heap node that represents this memory location to an abstract “heap” artifact. The descriptor of this new heap node is its address in the heap.

3.3.5 Restructuring

Existing software designs can significantly impede the implementation of parallelization scenarios. Typical reasons are frequent code changes for achieving short-term goals, often without full comprehension of the intended architecture [39]. This situation leads to issues like high code complexity, inappropriate abstractions, or tangled interfaces. The result is less maintainable software, which makes parallelization challenging and time consuming since the ramifications are difficult to predict and verify.

To overcome those issues and facilitate parallelization, programmers must restructure their software. A systematic method is *refactoring* [39]; the process of changing a software system to improve its internal structure without changing external behavior (although structural changes may affect non-functional behavior, such as performance). Refactoring proposes a series of small structural modifications, supported by tests, to verify the changes. For broad applicability, practitioners came up with catalogs of proven refactoring items, such as moving features between software elements or reorganizing data structures. Combinations of consecutive refactoring steps enable software redesigns at different granularity levels, up to the software architecture level. Refactoring let programmers incrementally align software structures with the design of parallelization scenarios.

Restructuring matches favorably with our proposed parallelization approach for the following reasons:

- *Comprehension.* Small refactoring activities incrementally improve the understanding of given software designs, which is a key objective of our proposed approach. For example, several code modifications introduce more meaningful names for software elements and ensure that code reflects the system's semantics. This simplifies the understanding of cryptic code regions and lead to more profound design decisions.
- *Refinement.* The iterative manner of refactoring fits well with the analyze-restructure cycle of our proposed approach. Programmers gain decisive insights relevant for parallelization by incremental refactoring. For example, removing global variables or moving functionality to different software elements may lead to different dependencies that simplify the implementation of parallelization scenarios.
- *Analysis.* Restructuring improves software's analyzability, a decisive property for analysis tools like Parceive. This procedure supports code comprehension and makes code structures more explicit, which empowers Parceive to yield more precise results.

Note that refactoring is not always the appropriate tool for reverse engineering. Refactoring improves developmental attributes of architecture designs by a strict focus on single concerns. If legacy systems have already suffered from serious design erosion, they may bear impractical design challenges for parallelization. Threatening these challenges with refactoring would only cure symptoms, not the root problems [23, 21]. In such cases, refactoring is insufficient for achieving the required qualities and programmers should apply more expensive methods, such as re-engineering or rewriting[30]. Parceive helps programmers also with these methodologies by pinpointing to the most critical software components.

Separating refactoring and parallelization is important. Although both steps (ideally) preserve software's functionality, they pursue different objectives. Refactoring aims to improve maintainability; parallelization is an optimization, it aims to improve resource usage. Thus, refactoring and parallelization differ in their motivations and mechanics, in the kind of tests, and in the appropriate tools. Distinguishing the two processes leads to a clear separation of concerns, which improves software comprehension and fosters dependable solutions.

In the remainder we present refactorings that are - in our opinion - most useful for parallelization, i.e., code modifications that affect the interactions in software systems. For each presented refactoring, we show how Parceive benefits from the modification, or vice versa.

Extract Function

A common refactoring to improve software's readability is extracting code into new functions. Proper usages of the method encapsulate inherently complex operations and name the resulting functions after their semantics. Programmers benefit from more expressive code that enables skipping implementation details while reading. The refactoring leads to smaller functions with few responsibilities. These are notoriously easier to comprehend and improve the overall readability [84].

Besides readability improvements, extracting code to new functions eases parallelization for two reasons. First, the refactoring can be used to align granularities of functions with the scopes of task proposed in parallelization scenarios. The refactoring's mechanics ensure safe code that can then be directly used with many parallel programming models. Second, function extraction forces programmers to reason about dependencies between functions and makes them explicit; they are concerned with variable scopes, access types, and appropriate function parameters. Unnecessary dependencies must be identified and removed, which eases parallelization.

Parceive benefits from the improved analyzability implied by this refactoring. The tool's lowest level of granularity for operational software elements are function executions. Extending program's call hierarchy by extracting code to fine-grained functions enables more precise dependency analyses and visualizations.

Listing 1 contains the function `checkExam`, which has two responsibilities: it assigns ECTS credits to a given `Student` object on a passed exam, and it prints a status message in case of a bachelor student. Extracting both functionalities into separate functions leads to the changed code shown below (for clarity reasons, we do not show the target functions).

```
void checkExam(Exam &exam)
{
    // update ECTS credits
    int nPoints {0};
    for (auto &task : exam.tasks)
        if (task.passed) nPoints += task.points;
    if (nPoints >= exam.requiredPoints) exam.student->nCredits += exam.nCredits;

    // print status for bachelors
    if (exam.student->isBachelorStudent)
        cout << exam.student->name << " has " << exam.student->nCredits << " ects\n";
}
```

⇓ Extract function ⇓

```
void checkExam(Exam &exam)
{
    updateECTS(exam);
    printStatus(exam.student);
}
```

Listing 1: Example code for the *extract function* refactoring.

The refactoring allows Parceive to identify commonly accessed memory locations between the functions `updateECTS` and `printStatus`. We use the Architecture View to perform a dependency analysis on the top-level artifact. The visualization indicates accesses from the refactored functions to public member variables of the `Student` class (Figure 3.12). Note that two software elements within the object are abstracted during navigation since they are irrelevant for the shown data dependencies.

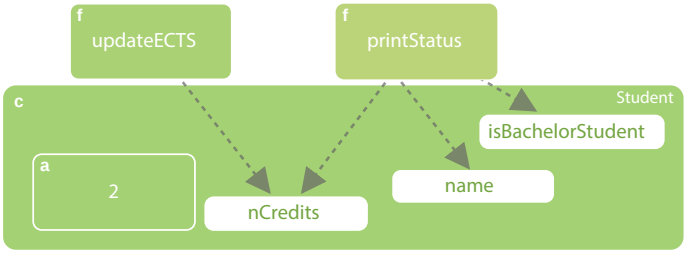


Figure 3.12: Architecture View for the example from Listing 1.

Combine Functions

Another refactoring to raise the level of abstraction is combining functions to common structures, such as classes, or namespaces. This enhances the logical cohesion of these functions and their shared data.

Combine functions to class binds data and functions that coherently belong together to a class. This step improves readability and expressiveness by proper class and method naming, relative to the new environment. Furthermore, classes encourage clean dependencies and interfaces by encapsulating data and thinking about proper method visibilities. Finally, having objects with integrated data and functionality enables to easily pass them to other software elements (e.g., task schedulers).

Combine functions to namespace groups data and functions that semantically belongs together into separate namespaces. This avoids name cluttering with other software regions, particularly with third-party libraries. Separating functions according to architectural or semantic concerns improves their understandability.

Both refactorings support our proposed approach in two ways. First, the code modifications improve software's analyzability by shaping design level abstractions. This fosters precise architecture hierarchies, which can be reconstructed and analyzed by Parceive. Second, combining functions to common environments facilitates clear dependencies relative to other code regions. Programmers benefit from the gained dependency information to further refine their parallelization scenarios.

The same example from above can be used to combine the functions `checkExam`, `updateECTS`, and `printStatus` to a separate class or namespace (Listing 2). The refactored class `ExamChecker` publishes the `apply` method to its users, which was renamed from `checkExam` to match the class context. The refactored namespace `chk` includes every function from the initial version.

```
void checkExam(Exam&);
void updateECTS(Exam&);
void printStatus(const Student&);
```

⇓ Combine functions to class ⇓

```
class ExamChecker() {
   void updateECTS(Exam&) const;
   void printStatus(Student&) const;
 public:
   void apply(Exam&) const;
};
```

⇓ Combine functions to namespace ⇓

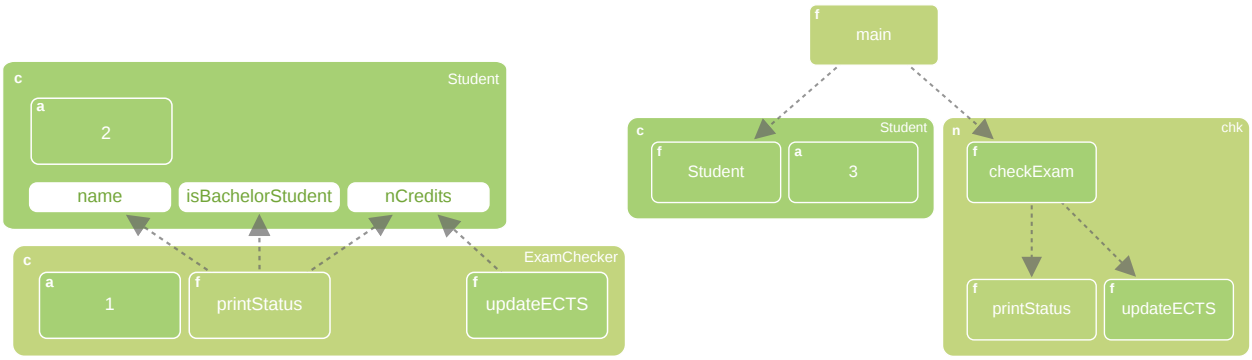
```
namespace chk
{
   void checkExam(Exam&);
   void updateECTS(Exam&);
   void printStatus(const Student&);
}
```

Listing 2: Examples for the *combine functions to class / namespace* refactorings.

We use Parceive's Architecture View to reveal the encoded artifact hierarchies after applying both refactorings. For more variety, we applied a data dependency analysis for the class case (Figure 3.13 a), and a call dependency analysis for the namespace case (Figure 3.13 b).

Move Function

Moving functions between different software elements or whole components is a typical refactoring to facilitate parallelization. Initial software designs may structure functionality (and data) relative to forces that contradict parallelization scenarios, such as Conway's Law [26]. Inappropriate function locations cause accidental dependencies that limit changeability or parallelism. To resolve these dependencies, programmers must establish proper abstraction levels and move the relevant functions closer to their accessed data or



a. Combine functions to namespace.

b. Combine functions to class.

Figure 3.13: Architecture View for the example from Listing 2.

called functionality. This step also improves software's modularity and encapsulation since the functions can then operate on local and private software elements.

Examples are programs that group their software elements into distinct abstraction layers. These layers simplify development and reduce complexity by pre-defined dependencies, mostly directed from the upper to the lower layers. However, parallelization scenarios that operate on multiple layers violate this scheme. A solution is to add orthogonal components to a new vertical layer and move the relevant functions to these components.

Deciding where to move functions requires programmers to examine current and planned code locations. They must analyze the accessed data by functions and their list of potential callers to estimate the required refactoring effort. Parceive helps to identify those structural dependencies.

```
class ExamChecker {
   void updateECTS(Exam&);
   void printStatus(const Student&);
 public:
   void apply(Exam&);
};
```

⇓ Move function ⇓

```
class ExamChecker {
 public:
   void apply(Exam&);
};

class Student {
 public:
   void updateECTS(Exam&);
   void printStatus();
};
```

Listing 3: Example for the *move function* refactoring.

Compare the resulting dependencies for the example code from Listing 3 before (Figure 3.13 a) and after moving the methods `updateECTS` and `printStatus` (Figure 3.14). The accesses between class `Student` and `ExamChecker` changed to one's inside `Student`.

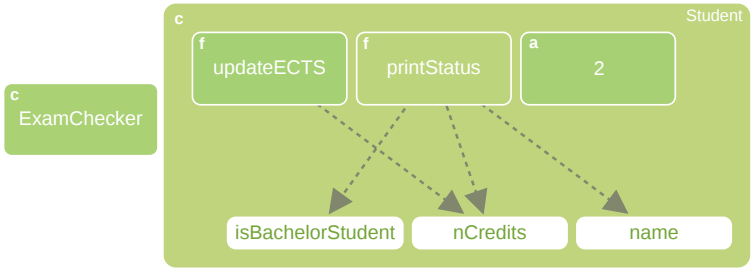


Figure 3.14: The Architecture View for the example from Listing 3.

Move Variable

Moving variables between data structures is another refactoring to ease software redesign. This method can improve a variable's suitability for a problem domain, lead to more expressive code, and requires less intricate data transformations. Typical signs for problems with data structures are frequent updates that require changes in multiple data structures. Programmers can reason about merging these data structures or moving variables between them to solve these issues.

The refactoring involves other code modifications to adjust access locations of moved data variables. Encapsulating those variables into accessor functions simplifies future refactorings by reducing the amount of necessary changes.

Note that certain data structure redesigns imply serious performance bottlenecks with specific hardware architectures. For example, iterating over non-contiguous data structures or ones that are unaligned with processor's cache sizes might increase access latencies. Another example are parallel operations accessing different data elements on common cache lines, which also has serious performance drawbacks due to the used cache coherency protocols.

```
void ExamChecker::printStatus(const Student &student) const
{
   if (student.isBachelorStudent)
      std::cout << student.name << " has "
                << student.nCredits << " credits\n";
}
```

⇓ Move variable ⇓

```
void ExamChecker::printStatus(const Student &student) const
{
   if (student.isBachelorStudent)
      std::cout << student.name << " has "
                << student.progress.nCredits << " credits\n";
}
```

Listing 4: Example for the *move function* refactoring.

In our example we move the variable `nCredits` from the class `Student` to a new structure `Progress` (Listing 4). The resulting Architecture View contains different edges representing data dependencies between the two accessing functions `printStatus` and `updateECTS`, and the variable `nCredits` (Figure 3.15).

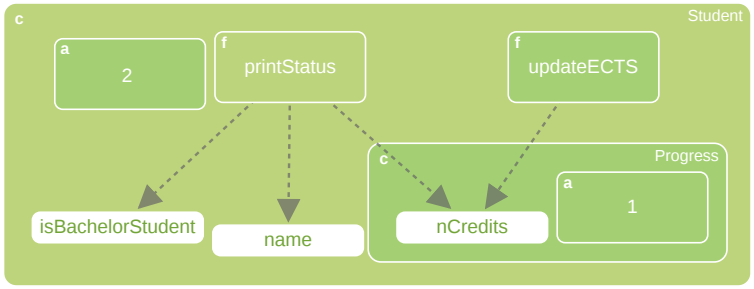


Figure 3.15: The Architecture View for the example from Listing 4.

Re-Engineering

The presented refactorings only represent a small fraction of possible code modifications to facilitate parallelization. Programmers who perform these refactorings improve their software's maintainability, establish proper abstractions, and increase thread safety.

However, more profound redesigns are necessary if parallelization scenarios propose the usage of different computation structures or algorithm strategies, such as the map reduce or pipeline pattern, respectively. Those modifications often imply several re-engineering activities operating at different levels; ranging from the low-level software design to the architecture level. Relevant parallelization scenarios may not only change the computation's granularity but move (partial) computations and functionality between different software elements.

Those design changes strongly impact peripheral software elements, such as databases or user interfaces. Hence, programmers must analyze and revise architectural dependencies and interactions between relevant elements. This step is even more important if parallelization scenarios comprise utility components, such as task pools or thread safe containers. Redesigns must consider the structural and behavioral implications to enable elegant and correct parallelization.

3.3.6 Parallelization

After the programmer elaborated on the final parallelization scenarios and restructured the code accordingly, the software can be parallelized. As with refactoring, conducting small iterative program transformations and tests reduces the complexity of this process.

At the beginning, parallel programmers choose a parallel programming model, such as task parallelism, thread parallelism, or message passing. Each of these models is supported within standards, programming languages, or libraries providing various functionalities for parallelism. These functionalities range from low-level threading primitives to full-featured parallelism frameworks with distinct task schedulers, data containers, and algorithms. Appropriate models provide higher-level abstractions to express the intended parallelization scenarios and match with the current software design. This improves readability of the implementation and understandability of the resulting parallelism. Examples for parallelism libraries are the Intel® Threading Building Blocks (TBB) [100], OpenMP [28], or MPI [86].

3.3.7 Validate, Verify, and Profile

The final step is concerned with the validation, verification, and profiling of the modified software. The objective is to determine whether the parallel program still conforms to the requirements and satisfies its intended use and user needs [18]. Verification aims to ensure that the changes are correctly implemented, in the sense that the resulting software meets the requirements. Validation aims to ensure that the software still fulfills its intended purpose. Profiling aims to investigate any changes in the run-time characteristics and performance of the software.

Validation and verification methods can be grouped into four categories: informal or formal, and orthogonal to these categories, static or dynamic. Informal methodologies rely on human reasoning and subjectivity without stringent mathematical formalism. Formal techniques are based on mathematical proofs of correctness. They can prove the absence of problems; hence, they represent the most effective methods to validate and verify software. Static techniques operate on static resources, such as software models or source code. These methods produce complete results and are efficient, even for large code bases. However, they can be unprecise and lead to *false-positives* (i.e., wrongly identified errors) if results depend on run-time decisions. Dynamic techniques inspect behavior and require execution of the software or corresponding models. These methods produce precise results, even for parallel software. However, dynamic techniques may lead to *false-negatives* (i.e., missed errors).

Validation with Parceive

Although Parceive provides no active parallelization support, such as code transformations, it helps programmers to validate their parallelized programs. The tool's backend and frontend component consider multi-threaded software. The backend records thread creation, thread joining, and various synchronization events induced by low-level threading libraries. The Trace View and CCT View of the frontend visually differentiate function executions performed by distinct threads. These visualizations enable users to comprehend their application's parallelism.

The Trace View let users randomly select a set of threads executed during a program run. Hence, the visualization's settings presents the list of all threads, sorted by their run time. The Trace View dumps the selected threads vertically aligned, analogous to the single-threaded representation (Figure 3.16). Programmers can see the temporal distribution of the threads at one glance. This helps to validate the parallelization relative to the underlying parallelization scenarios.

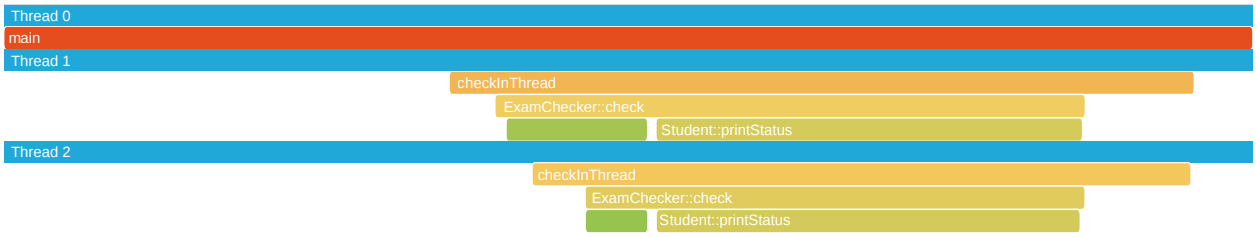


Figure 3.16: Trace View showing a multi-threaded program.

The CCT View provides a user option to enable or disable frames around call sub-trees that are executed by distinct threads. Each thread frame represents the convex hull around the tree nodes and has a different color (10 alternating colors) for better differentiation (Figure 3.17).

The visualization effectively supports parallelization validation due to the integrated data dependency analyses. Programmers can select multiple threads (represented by their respective sub-trees) and perform a deep

dependency analysis to identify commonly accessed memory locations. Occurrences that are not synchronized are race conditions and must be resolved.

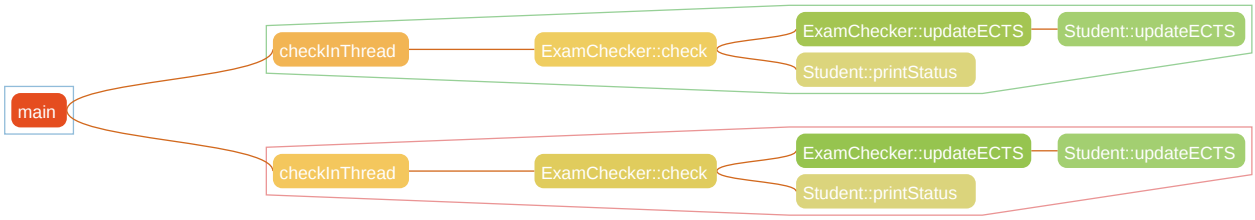


Figure 3.17: Trace View showing a multi-threaded program.

3.4 Conclusion

In this chapter we presented our tool-based approach for parallelism discovery. The approach is focused on supportive data dependency analyses at different levels of abstraction. We described the developed methodology for software architecture reconstruction that is used to recover architecture hierarchies. Then we proposed a pattern-based approach for parallelization that comprises five steps. We focused on the steps where Parceive supports programmers. The tool's provided visualizations and analyses were described and exemplified by small examples.

4

Design and Implementation of Parceive

"I'm a great believer that any tool that enhances communication has profound effects in terms of how people can learn from each other, and how they can achieve the kind of freedoms that they're interested in."

– Bill Gates

4.1 Introduction

In this chapter we highlight important design decisions of Parceive, our interactive parallelization tool. It supports Linux and Windows programs written in the C/C++ programming languages that are provided as compiled binary files. Parceive captures the dynamic behavior of programs by inspecting determined events during execution. Differing from static approaches that analyze source code, this approach can handle typical problems associated with parallelization, such as pointer aliasing and dynamic memory management. Note that the binary files must provide symbol information to enable the mapping between machine code and source code. Our tool relies on this mapping to reconstruct identifiers for various software entities, which is critical for software comprehension.

Programmers use Parceive to identify and iteratively refine parallelization scenarios (Figure 4.1). Every iteration begins with a manual configuration of the intended program analysis, i.e., defining the executable file, the analysis scope, and some filters for the data collection. Afterwards, users initiate the automatic analysis process to collect structural and behavioral information. Parceive provides a set of visualizations to depict (often orthogonal) software aspects. Programmers use the gained knowledge to refine their analyses in subsequent iterations towards the relevant parts of their application.

In Section 4.2 we present a coarse overview of Parceive's architecture showing its major components and their relations. The requirements for our tool are presented in Section 4.3. We then introduce the meta model used to capture and describe behavior information in Section 4.4. The design of our tool is separated into a backend and a frontend to achieve high extensibility and portability. In Section 4.5 we describe the backend used to read existing debug information and collect run-time information by dynamic binary instrumentation. In Section 4.6 we explain the frontend component, a standalone tool that represents Parceive's graphical user interface. We discuss the architecture and integration of the component responsible for software architecture reconstruction in Section 4.7.

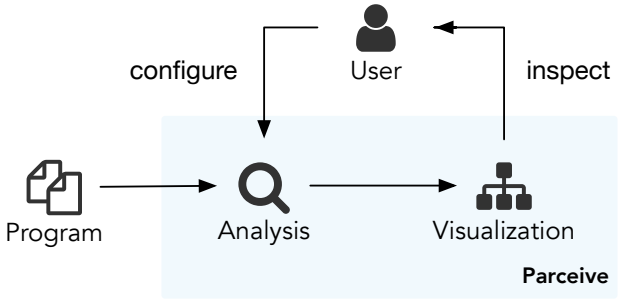


Figure 4.1: Parceive's typical workflow.

4.2 Design Overview

Parceive is built upon a modular software architecture through all design levels to ease extensibility. At highest level, the architecture comprises a backend and a frontend component (Figure 4.2). The backend contains a tracing framework for collecting data about events relevant to parallelization during execution, such as function calls and memory accesses. The frontend includes a visualization tool and a module for software architecture reconstruction. The former provides a graphical user interface, used to manage analyses, and interact with the analysis results via multiple visualizations. The latter applies a set of user-defined architecture rules to extract structural information from executables. Here, the objective is to reconstruct specific aspects of the implemented software architecture.

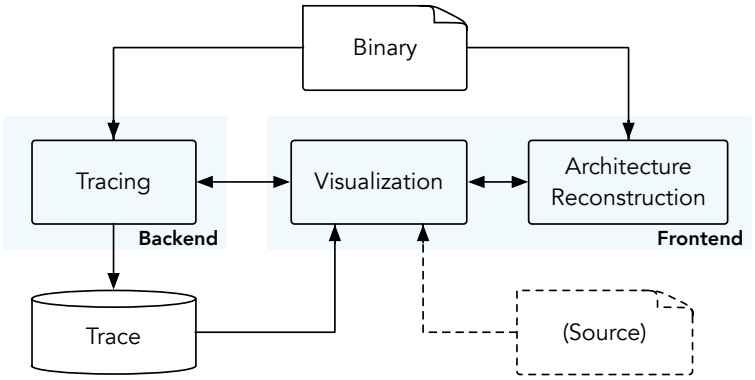


Figure 4.2: Main components of Parceive.

Users interact with Parceive by using the command-line interface or the graphical user interface. The tool accepts user applications in the form of compiled binary files. The backend executes these binary files according to a given analysis configuration, such as input arguments and filters. During the execution Parceive stores the collected run-time information into a trace database. The frontend analyzes the binary files to extract program models of user applications. That extraction is guided by a set of rules representing the architectural knowledge of the users. The resulting models are then combined with the trace data to enable useful analyses and visualizations for parallelization. Note that providing source code is unnecessary; however, source files can be used for relating visualization entities to respective code regions.

4.3 Requirements

In this section we highlight important functional and non-functional requirements of Parceive. Our tool supports parallelization of industrial software, which typically consists of numerous components with complex behavior and interactions. Thus, we tailored the requirements towards appropriate software systems with complex software architectures and non-trivial implementations.

4.3.1 Functional Requirements

The functional requirements are aligned with typical tasks during software parallelization. We have distilled five major requirements by inspecting use cases from our industry partner Siemens and interviewing the engineers. We transformed these requirements into a set of features that are - to the best of our knowledge - in their combination novel.

1. *Tracing.* Parceive's tracing capabilities record event data for the featured analyses and visualizations. Relevant events for parallelization are function calls, memory accesses, API calls of threading libraries, and loops. Function call traces are essential to recognize the temporal order of events, which is important to understand dynamic program behavior. Detailed information of memory accesses let programmers identify data dependencies. Parceive can also analyze multi-threaded programs by tracing different threading primitives. Finally, tracking of durations and trip counts of loops is required to tailor the analyses towards application hot spots.
2. *Symbol Resolving.* Software comprehension is a mental process that involves multiple actions at various levels of abstraction. The structural software components at the highest level must be mapped to the corresponding operations at source code level to understand their behavior. The expressions and statements of the source code operate on language entities, such as variables and functions. Useful visualizations that help to understand and relate these entities must provide means for showing their symbol names. Parceive has built-in features to resolve symbol names of arbitrary code entities. Additionally, our tool provides functionality to reconstruct inherent structural relations, such as class members in object-oriented code or namespace mappings.
3. *Architecture Reconstruction.* Parceive lifts trace information collected at instruction level to the software architecture level. Thus, our tool extracts existing debug symbols with application binaries to gain data about architectural software aspects. The data must be filtered, grouped, and structured according to user-defined architecture rules. We implemented this architecture reconstruction process as a separate module that can be used by all other tool components. For example, the backend relies on the resulting program models to filter the tracing towards interesting user components. Another example is the visualization that displays the architecture information in representative views.
4. *Filtering.* A filtering mechanism for the tracing capabilities is mandatory for analyzing industrial-size applications. Without filtering, the amount of collected data quickly leads to unpractical analyses and visualizations. Besides performance problems, such as scalability issues, the comprehension quality suffers from too much data. Clarity in both the information and the resulting views is crucial to focus on the important analysis results. Thus, Parceive includes static and dynamic filtering mechanisms. Static filtering enables or disables tracing for whole parts of the user software. Dynamic filtering starts and stops tracing on pre-defined events during the execution, such as specific function calls.
5. *Visualization.* Effective analysis tools provide visualizations that focus on the key elements of their individual tasks. In our case, parallelization is the driving factor for the user interface. To support the

multitude of challenges while parallelization, we provide multiple representations of the given user software to show orthogonal aspects. For example, programmers must know the order and duration of the executed system tasks, and their accessed memory locations. The resulting visualizations must interact with each other to highlight different aspects of the software elements.

4.3.2 Non-functional Requirements

To support industrial-scale applications, Parceive must fulfill the following non-functional requirements.

1. *Performance.* Providing sufficient performance is a major requirement for Parceive. Especially when applied to a long-running program execution, providing practical slowdown and good scalability is indispensable for achieving high user acceptance. Slowdown measures the delayed run time of an analyzed program when compared to an unsupervised execution. Note that instrumentation tools - like ours - cannot completely avoid slowdown. Scalability rates the relative slowdown according to the analysis input, i.e., the number of events of an execution. Good scalability results in a roughly constant factor relative to the execution duration. Parceive aims for good scalability by using algorithms and data structures tailored for trace operations, preprocessed database transactions, and filtering.
2. *Transparency.* Reliable and correct analysis results require full transparency of the analysis tools to preserve the original program behavior. Besides precise information about function calls and memory accesses, this includes the usage of external interfaces, such as I/O operations. Thus, Parceive inspects run-time events but may not modify application instructions. Additionally, any used third-party component of Parceive, such as libraries or instrumentation frameworks, must be transparent.
3. *Portability.* To support a wide variety of industrial use cases, analysis tools must rely on platform independent technologies. Those use cases differ in their targeted hardware architectures and the supported operating systems. High portability increases the tool's applicability and its usefulness. Note that Parceive's features require analyses operating on low abstraction levels, such as binary instructions and hardware registers; here, high portability is hard to achieve without a fair amount of tailored system code. We limit the scope of Parceive to Linux or Windows applications that are written in the C or C++ programming languages. However, the visualization component of our tool is platform independent by relying on web-based technologies.

4.4 Meta Model

In this section we describe our proposed meta model to store trace information. This meta model represents the interface between the backend and the frontend (Section 4.2). The backend acts as the producer and the frontend as the consumer of the trace data. Our meta model relies on a relational structure and supports optimized queries for specialized analyses relative to data dependency analysis. In the following we give an overview of the meta model, and highlight details on distinct relations for important features of Parceive.

4.4.1 Overview

The meta model includes 12 entity tables, and some pre-defined relations between these tables (Figure 4.3). Entity tables represent relevant software entities for parallelization; some entities are explicit, such as functions and memory accesses; some are conceptual, such as segments and loop executions. Here, we omit

details for better clarity, such as the table fields or extended relations (see the remainder of this section). The meta model supports static data (the dashed nodes) and dynamic data (the solid nodes) of user programs. Examples for static application data are source files, functions, or loop information. Dynamic data includes function calls, memory accesses, or loop iterations.

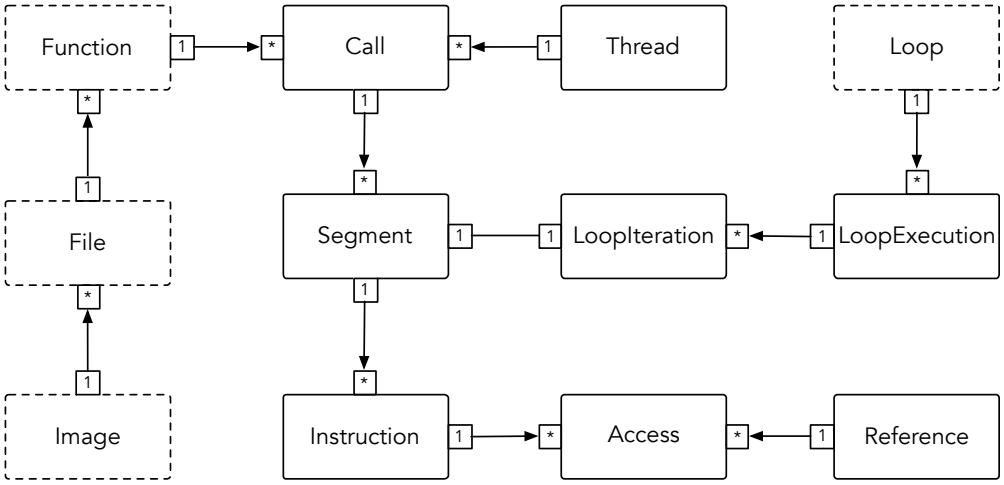


Figure 4.3: Parceive's meta model including entity tables as nodes and relations as edges.

4.4.2 Image Files

Most industrial applications are composed of multiple components from open source or inner source (i.e., organization wide code) projects. Each component is compiled to binary files forming executables or libraries. We use the term *image* for executables or libraries, according to the definition of our utilized instrumentation framework (Section 4.5). Parceive traces image information of user applications at load time (i.e., before the execution). Each run-time event is associated with a single function that maps to one file and one image.

The proposed meta model supports static image data, such as source files and functions in the respective entity tables (Figure 4.4). The *Image* table and the *File* table have single entries for each image file and source file, respectively. The table *Function* stores information for each called function during execution, such as the signature and the function type (e.g., regular function or allocation function).

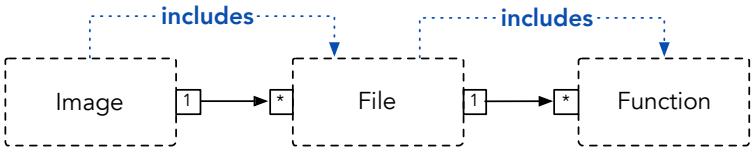


Figure 4.4: Excerpt of Parceive's meta model for capturing images.

4.4.3 Function Calls

Analysis of function calls is indispensable to comprehend software behavior. Parceive produces run-time models that include precise call graphs (i.e., calling context trees [5] including the run time of each function execution). The meta model is designed towards high efficiency relative to information density and query

performance. This quality is crucial for sophisticated analyses and visualizations, especially with recursive function calls.

The meta model supports function calls by relations between the entity tables *Function*, *Call*, *Segment*, and *Instruction* (Figure 4.5). Any called function during an execution is represented by one entry in *Function*, and one entry in *Call* for each invocation. Every call entry relates to a consecutive list of segments; a segment is a conceptual entity that represents multiple instructions which are executed all together. Examples are basic blocks (according to compiler semantics), or unconditional loop iterations. The rationale for segments is to improve query performance by minimizing required model data. Each segment relates to a consecutive list of instructions with specific instruction types, such as memory accesses or a function calls. To model relations from a caller to a callee, each call instruction (the caller) relates to a single entry in the call entity (the callee).

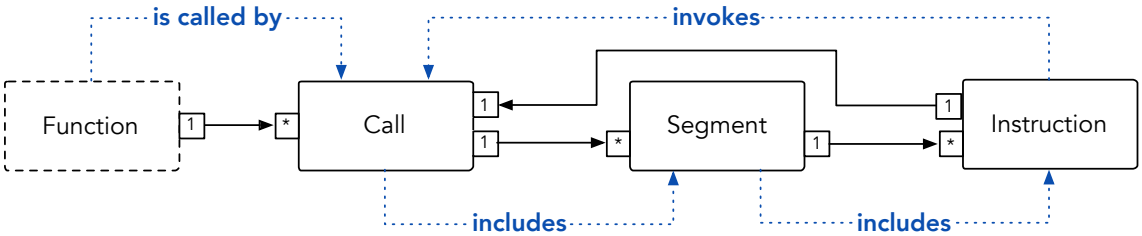


Figure 4.5: Excerpt of Parceive's meta model for capturing function calls.

4.4.4 Memory Accesses

Data dependency analysis requires precise information of memory accesses. The meta model supports that information by providing the entity tables *Instruction*, *Access*, and *Reference* (Figure 4.6). Each instruction that accesses some memory locations is represented by a single entry in the *Instruction* table containing the access type, and a corresponding source code location. Each access instruction relates to an entry in *Access*, which includes the access type (i.e., read, write, or read-write), and the access offset of the memory location. This offset allows to detect and distinguish accessed subparts of certain data structures, such as single array elements. The *Reference* table abstracts the accessed memory locations at arbitrary levels of granularity. Examples are plain data types (i.e., numbers or strings), member variables, or custom allocated data. The meta model supports the relation between memory locations on the heap and their allocation instructions by providing a relation between *Reference* and *Instruction*.

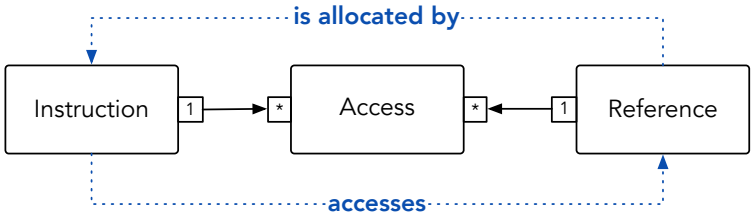


Figure 4.6: Excerpt of Parceive's meta model for capturing memory accesses.

4.4.5 Threads and Processes

The meta model associates function calls with software threads and processes. Essentially, every occurred event can be distinguished by process and thread identifiers. This enables Parceive's analyses and visualizations to consider concurrent program behavior. The entity tables *Call*, *Thread*, and *Instruction* support concurrent execution models by storing threading information (Figure 4.7). Entries in the *Thread* table identify the process and thread for each function call. The programming model can include fork or join instructions relating to single threads, or synchronization primitives (e.g., mutex locks, or conditional waits). Specific entries in the *Call* table can be used to model respective function calls of parallelization libraries.

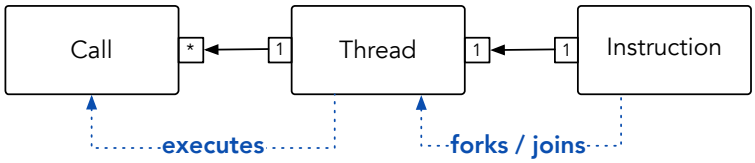


Figure 4.7: Excerpt of Parceive's meta model for capturing threads and processes.

4.4.6 Loops

Useful software parallelism often exists in long-running loops, which apply the same operations on multiple operands. The meta model supports corresponding analyses by explicit loop information. The respective entity tables are *Loop*, *LoopExecution*, and *LoopIteration*. *Loop* stores static data of source code locations for every loop in a user application. *LoopExecution* stores instances of a loop execution, together with their respective run time. Note that we do not assign run time to single iterations to reduce the resulting overhead for time tracking. Thus, *LoopIteration* only contains the trip counter for the number of iterations per loop execution. We support the explicit modeling of nested loops by a relation of the loop execution to the iteration of its parent loop.

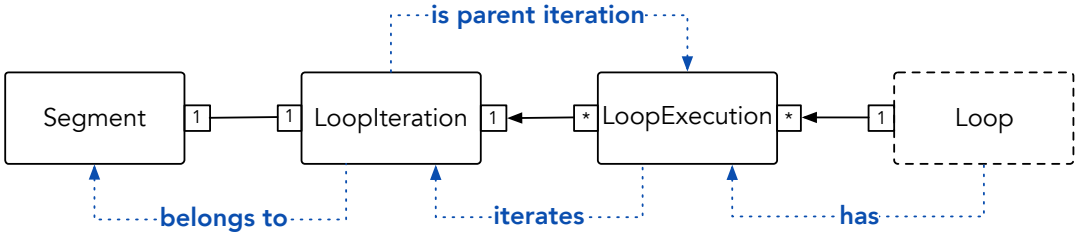


Figure 4.8: Excerpt of Parceive's meta model for capturing loops.

4.5 Backend

In this section we describe Parceive's backend component. It is based on binary analysis to obtain run-time information, which could be obtained via sampling or tracing. Sampling periodically inspects program execution and aggregates the collected data whereas tracing triggers such inspections on pre-defined events. Note that sampling incurs less overhead than tracing; however, it is not sufficiently rigorous to capture all relevant run-time events conclusively. Thus, we implemented a tracing tool that utilizes dynamic binary

instrumentation. This allows the insertion or manipulation of machine instructions during run-time. Operating on binary files is often the only practical approach to analyze large industrial applications. Other approaches, such as compiler-based instrumentation, require special compilers and adaptations of the build system, which is not necessarily feasible in many project environments.

The backend operates in four successive phases (Figure 4.9). During the initial phase (A), symbols are extracted from debug information. Parceive relies on these symbols to resolve variable names for accessed memory locations and links dynamic analysis results to elements of reconstructed software architectures. During the second phase (B), the tracing tool instruments the application binaries with callbacks to analysis functions, which will be triggered by pre-defined events. In the third phase (C), the application is executed and monitored by Parceive. The analysis functions track the program's control and data flow, and record relevant run-time information according to the proposed meta model (Section 4.4). For example, single callbacks are added before each function call. The respective analysis function gets the identifier of the called routine, the executing thread, and some register values, such as the base pointer. Parceive requires this information to maintain precise function call stacks and assign each run-time event with the appropriate function execution. The final phase (D) clears intermediate state variables that were maintained during the previous phases.

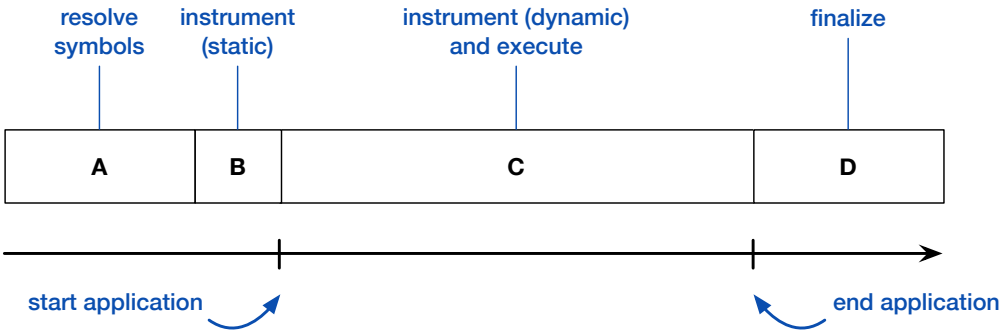


Figure 4.9: Analysis phases of Parceive's backend.

Dynamic binary instrumentation frameworks ease building analysis tools, like ours. We rely on the Pin toolbox [78], which aims to provide easy-to-use, portable, transparent, and efficient instrumentation facilities. Typical analysis tools building upon Pin are profilers, cache simulators, trace analyzers, and memory bug checkers. Pin comes with a rich API to support code manipulation at various levels, such as image or instruction levels. Furthermore, built-in analyses let tool-writers infer contextual information of user applications, such as function symbols or register values. Pin's functionality is to a large degree architecture independent; however, it provides also architecture-specific functions to access special hardware registers. For good efficiency, Pin's instrumentation technology uses a just-in-time compiler to insert and optimize code.

4.5.1 Design

The backend component has a modular architecture that is designed towards the ATOM model of dynamic instrumentation frameworks [109]. The architecture comprises a driver component, instrumentation modules, and support modules (Figure 4.10). We give more details on the instrumentation and support modules in the following sections. The driver component has three objectives: (1) it delegates the instrumentation at image load time and at execution time to instrumentation modules. These modules encapsulate logic for specific analysis concerns, such as function calls or memory accesses. (2) Additionally, the driver initializes

and forwards support modules to the instrumentation modules. Support modules handle supportive tasks for the analysis, such as trace writing or symbol resolving. (3) Finally, the driver defines the command-line interface (CLI) and uses it to control certain backend features. For example, the user can turn on or off tracing of heap memory or loops.

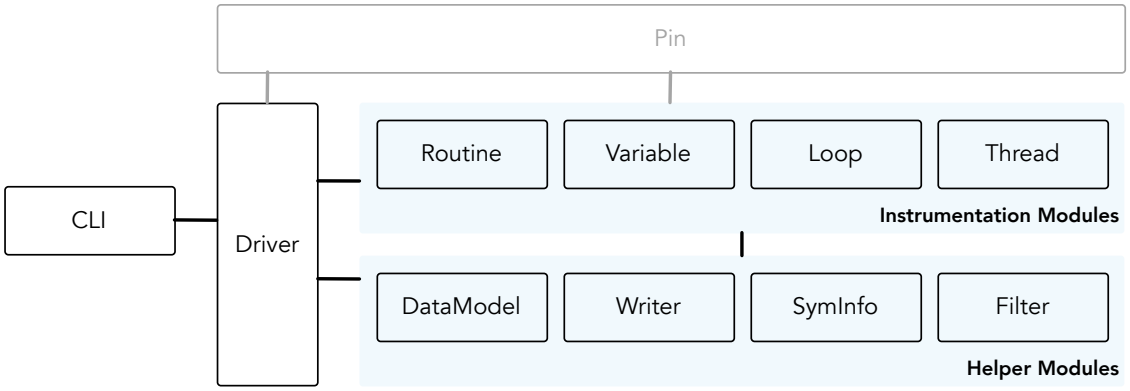


Figure 4.10: Parceive's backend architecture.

4.5.2 Instrumentation Modules

The instrumentation modules focus on distinct analysis concerns. Each module inserts distinguished callbacks to obtain run-time data at relevant events by using the Pin API. The modules share a common interface used by the driver component via an observer pattern; each instrumentation module (the observers) register themselves at the driver (the subject) to get informed about instrumentation phase changes during run-time. These phases are:

1. *Initialize (Finalize.)* Called at the start (end) of the whole analysis. The backend relies on these events to initialize (destruct) necessary state variables before (after) program execution.
2. *Before instrumentation.* Called before the driver component adds instrumentation callbacks to the user applications. Parceive's modules rely on these events for adding custom instrumentation callbacks, such as class constructor entries.
3. *Before image instrumentation.* Called before instrumentation of binary files. The events are used to add library-specific logic to the backend, such as special handling of the Posix Thread library.
4. *During image instrumentation.* Called at instrumentation time for each binary file or function of the user application. The backend uses these events to insert the static analysis function callbacks.
5. *During trace instrumentation.* Called during instrumentation of every *trace* - a similar concept to extended basic blocks in compiler theory - and every instruction of the user program. The instrumentation modules rely on these events to insert dynamic callbacks during run-time, such as callbacks for non-filtered function calls.

Routine

The Routine module is mainly concerned with maintaining a shadow call stack for function executions. The backend component uses this information to construct a precise calling context tree, including the run times of each call. For that purpose, the module instruments all (non-filtered) call and return instructions. Note that the backend component considers various techniques for function calls, such as (long)jumps and

indirect calls in case of dynamically loaded libraries. Multi-threaded user applications are supported by assigning thread identifiers of function calls. To detect object life cycles, the module considers constructor and destructor calls to keep track of owner relations and heap addresses. Additionally, it assigns identifiers of debug symbols to executed functions for two reasons. First, local variables that are accessed during the execution can be precisely resolved (Pin's integrated functionality returns only the addresses of memory accesses). Second, the symbol identifier is the key to map reconstructed software architecture information (Section 4.7) to the execution trace. Finally, the Routine module handles thrown exceptions of user applications by relying on appropriate Pin features.

Variable

The Variable module statically instruments each (non-filtered) instruction that accesses memory. For each of these instructions, the module adds the access type (i.e., read, write, or read-write), and the identifier of the accessed memory reference. If this reference is not known at instrumentation time, the Variable module resolves the reference symbol during run-time. Here, the resolver functionality distinguishes between local (stack), global, static, or heap references. Hence, the module maintains shadow variables for each memory location, separated by these memory types. Local references are kept in a stack data structure to reflect the scope of stack variables. Global and static references are stored in a hash map. Heap references are tracked by instrumenting function calls for dynamic memory allocation (e.g., `malloc` or `free`) to maintain a range-based map of heap addresses. An important feature of the module is the refinement of accessed memory references. For example, accesses to structured data (e.g., member variables, or nested structures) are refined to visualize not only variable names but their containment relations. This feature is important for improving user's software comprehension.

Loop

The Loop module analyzes loops in user programs; an optional feature that can be enabled by a provided command-line switch. The analysis relies on Aho et al.'s algorithm to statically detect loops in source code [2]. This algorithm assumes a flow graph in form of static basic blocks as input. We obtain dynamic basic blocks - called *extended basic blocks* - from Pin, which are not equivalent to static basic blocks. Thus, the module initially transforms the dynamic basic blocks to static counterparts. In the next step, all dominator nodes (i.e., basic blocks) in the resulting flow graph are identified. Afterwards, the algorithm detects all loops by searching for edges in the flow graph whose head nodes dominate their tail nodes (i.e., back edges). The module assigns an identifier to each loop and writes them together with the source code location to the trace database. During the instrumentation phase, the module instruments all entry, exit, and jump instructions of the found loops. This allows the backend to write run-time information of individual loop executions and iterations.

Thread

The Thread module supports tracing of concurrent software and provides thread-local storage to other backend components. Concurrent C/C++ code relies on threading libraries to manage threads (or higher-level abstractions) and to synchronize shared resources. Threading libraries are based on APIs that support multi-platform programming for various programming languages. Currently, the instrumentation module is defined for the Posix[1] threading API for Linux, and the Windows API. During instrumentation, the module adds callbacks to each function call that creates or joins threads. During run-time, the first and last function called by a thread is mapped to the respective thread create and join function, respectively. This results in a sub-tree of the calling context tree for each executing thread. For synchronization, locks are instrumented and associated with single lock references.

[1] http://standards.ieee.org

4.5.3 Helper Modules

We encapsulate functionality orthogonal to instrumentation and analysis features into four helper modules (Figure 4.10). These modules are initialized by the driver component during the startup phase. The driver component provides an interface to the instrumentation modules to access these helper modules.

DataModel and Writer

Instrumentation modules do not explicitly write trace entries, but rely on the DataModel and Writer modules.

The DataModel module represents an abstraction layer to hide details of the meta model for easier trace writing. For example, the module writes complementary entries to the tables *Segment*, *Instruction*, *Access*, and *Reference* when memory accesses are processed. This lowers the complexity of instrumentation modules and eases refactoring caused by potential meta model changes.

The Writer module is used by DataModel to perform actual write operations to the trace database. It provides a common interface to write each entity of the meta model (Section 4.4), which offers two benefits: it fosters the usage of other trace formats or databases (e.g., the OTF trace format, or in-memory databases), and it is the key for global optimizations of trace writing. Currently, we implemented transactional writing and prepared query statements to reduce module latencies. Additionally, the database schema uses no table indices or constraints during the run-time, but establishes them post-mortem. Parallel write operations are synchronized by the Writer module because these are not supported by our current database.

SymInfo

The SymInfo module parses debug information attached to program binaries for extracting symbol data, such as symbol names or source code locations. This symbol data can be related to any software element to improve user's comprehension. Every function can be assigned to symbol data by statically comparing function signatures. Every variable can be assigned to symbol data by comparing their memory addresses during run-time. Note that memory addresses are not generally known at instrumentation time; thus, we infer the symbol information for stack variables and heap variables during run-time by considering stack registers and variable offsets. The module considers a specific language syntax to structure user applications, such as namespaces, or object-oriented code (e.g., classes, member variables, or inheritance). The Routine module uses SymInfo to fill symbol information of function call instrumentation. The Variable module uses SymInfo to resolve the accessed memory references during run-time. Additionally, SymInfo is used by the software architecture reconstruction component (Section 4.7).

Filter

Filtering is essential for tracing tools like Parceive to ignore parts of the user programs that are not relevant for specific analyses. The benefits are less run-time overhead, improved user's comprehension by reduced clutter, and lower latencies by the user interface (e.g., analysis queries, or visualizations). The Filter module provides a user interface to specify such filters at various levels of granularity. Thus, we extended the architecture rule language based on regular expressions to define entities and components of the software architecture (Section 4.7) that shall be filtered. Examples for such components are executables or libraries, functions, or source files and directories. The Filter module supports static, dynamic and partial filters.

Static filters are valid during the whole run time. The Routine module hence instruments direct function calls only if the corresponding function is not filtered. In case of indirect function calls (e.g., using function pointers, or functions of shared libraries), the filter is checked during run-time.

Dynamic filters allow to start and stop filtering at the execution of pre-defined software elements. A typical use case is the analysis of certain phases of user software. Technically, whenever a function of a dynamically filtered software component is executed or left, the filter will be enabled or stopped, respectively.

Partial filters are similar to dynamic filters, but do not filter memory accesses. This results in a partial calling context tree, in which the accessed memory references by filtered function calls are assigned to the last non-filtered call nodes. A use case is data dependency analysis of software that relies on third-party libraries. Here, potentially deep call levels of these libraries are often unimportant for the programmer's comprehension and can be ignored.

4.6 Frontend

In this section we present key aspects of Parceive's frontend component. The frontend is a standalone tool to manage analyses of user programs and to interact with the resulting visualizations. The frontend's architecture comprises a user interface and a visualization framework (Figure 4.11). The user interface includes a dashboard for organizing and navigating through single analyses. Views are interactive visualizations of the collected trace data. A visualization framework passes this trace data to the Views by providing them a common interface to abstract details of the meta model and provide objects, events, and optimizations. Some visualizations require structural information of user programs, which exists in debug symbols. The symbol loader extracts such information from binaries and provides it to the Views.

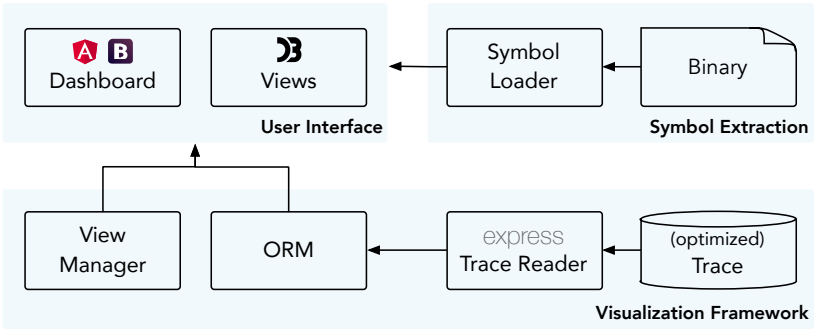


Figure 4.11: Software architecture of Parceive's frontend component.

The frontend is based on the Electron toolkit [47] for cross-platform applications built upon mature web technologies, such as JavaScript, HTML, and CSS. Electron relies on the chromium browser kernel [50] and NodeJS [38] to enable stand-alone applications with native performance. Parceive benefits from the toolkit by the following features. First, applications gets a native appearance (e.g., OS-dependent dialogue windows or menus) and provide installers for various platforms. Second, Electron gives us access to native OS functionality, such as file handling, or launching executables. This lets the frontend interoperate with Parceive's backend to initiate the execution of user programs similarly at any platform. Last, the Node.js runtime in Electron enables execution of user-defined modules - written in system languages - to achieve high performance. We provide a Node.js module for invoking the Symbol Loader to extract structural information of user software from the attached debug symbols (Section 4.7).

4.6.1 Dashboard

Parceive's user interface includes a dashboard to navigate through analyses, organized into projects and runs. Each project represents a set of runs for a given user program; here, a run is a traced program execution with

individual input arguments and analysis parameters. After successful execution, the provided visualizations show different aspects of the analysis results. The dashboard is based on Angular.js [49], a framework to build web-based applications according to the model-view-controller (MVC) pattern. Angular.js alleviates data binding between controllers and Views (for clarity, we call them dialogues), and allows state handling across Parceive's four dashboard dialogues (Figure 4.12). The Main dialogue is the user's starting point and includes links to all conducted projects and runs. Furthermore, the dashboard contains a navigation bar providing the same links for quick navigation. We describe the other dialogues in the following paragraphs.

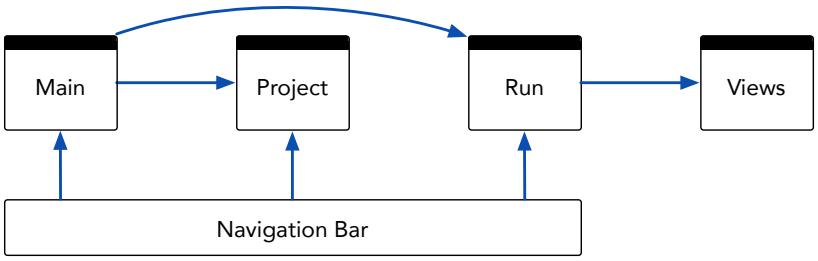


Figure 4.12: Dialogues (rectangles) and interactions (edges) of Parceive's dashboard.

Project

Users create new analysis projects with the Project dialogue. They must specify the project's name, the application's executable, and some program arguments used as default for prospective runs. Once created, the project cannot be changed to avoid inconsistency across different project runs (e.g., by analyzing different versions of user applications). Executable files are checked for existence, debug symbols, and potentially problematic compiler optimizations. If the checks were successful, the dialogue's controller initiates the project state in a persistent registry, and creates a project directory to keep all analysis data.

Run

Runs represent single analyses, i.e., traced executions of user applications. Users use the Run dialogue to create, modify, and execute project runs. The dialogue includes various controls to set run settings, such as input fields for the run name and command-line arguments. Four checkboxes configure analysis options to constrain the tracing step: (1) tracing memory accesses; (2) tracing memory accesses to (local) stack variables; (3) tracing offsets of memory accesses (e.g., to distinguish accesses of different array elements); and (4) tracing loop executions and iterations. Users specify trace filters with the editor control by our rule-based architecture language (Section 4.7). Afterwards, the user creates a run, which initiates some filter checks, the creation of a run directory, and initiating mandatory state variables. The run settings can be changed until the user executes the run. This execution initiates the backend component to trace the user application accordingly during execution. After successful execution, a post-processing step optimizes the trace database (Section 4.5), and enables navigation links to the analysis visualizations (called Views).

Views

Whenever a run has been executed successfully, the frontend enables a navigation link to its analysis visualizations. These visualizations are shown in the Views dialogue, which supports a flexible grid layout to arrange an arbitrary number of visualizations both vertically and horizontally (Figure 4.13). A dropdown control at the top of each grid window enables to display any available View in it, such as the Profiling View or the CCT View.

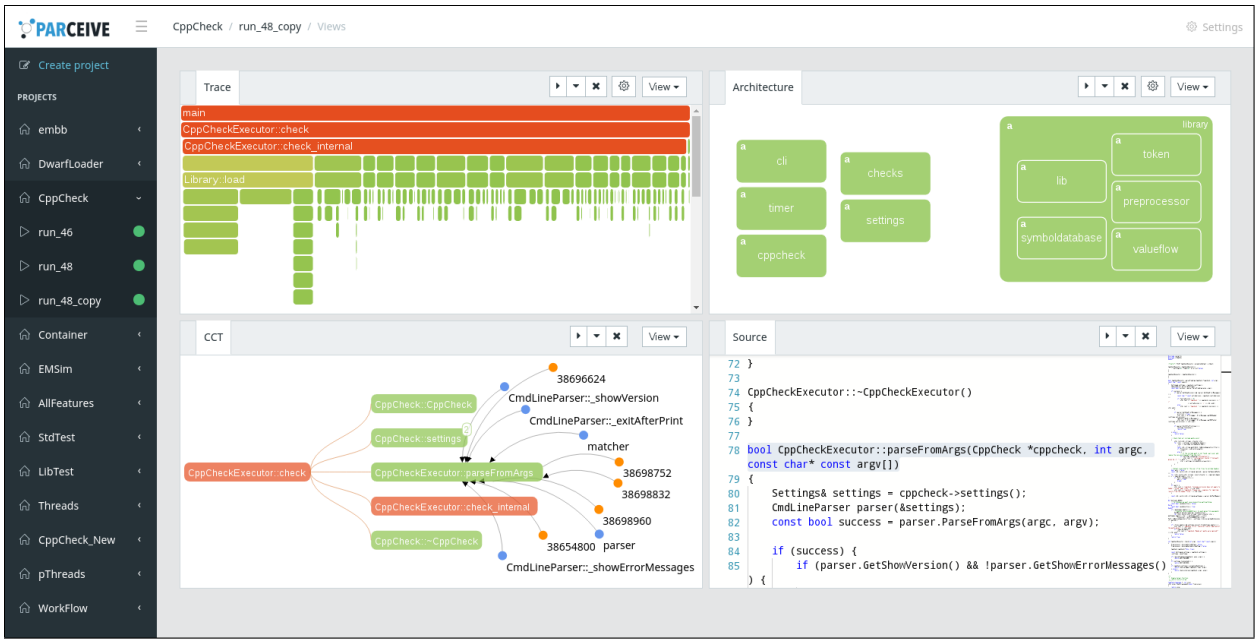


Figure 4.13: Parceive's dashboard with four active visualizations: (top left) Trace View; (top right) Architecture View; (bottom left) CCT View; and (bottom right) Source Code View.

4.6.2 Visualization Infrastructure

In this section we present certain aspects of our web-based visualization infrastructure. It facilitates the integration of arbitrary Views into Parceive by providing a common interface to access abstracted run-time traces. The biggest challenge when dealing with traces is the potentially overwhelming amount of data. This often leads to unmanageable visualizations with unacceptable delays. Our infrastructure addresses this problem by building upon a responsive client-server architecture that provides four key services: (1) trace optimization, (2) on-demand loading, (3) caching and pipelining, and (4) state management and communication. The infrastructure includes a trace reader to query run-time data, an object relational mapper (ORM) as abstract data interface, and a view manager for view interactions.

Trace Reader

This service provides on-demand reading of collected trace data, which improves responsiveness of the visualizations. Often, loading entire traces is not feasible due to memory restrictions. To solve this problem, we developed a server based on the Node.js express framework. The server provides a REST API [34] to perform data retrieval in separate processes. The implementation uses multiple parallel reads to the database for reducing latencies and increasing throughput when large amounts of data is requested. This lets users seamlessly explore and navigate through traces containing an otherwise unmanageable amount of data.

The Trace Reader includes generic and specialized queries to retrieve analysis data. Generic queries return one or multiple entries of individual table entities by specifying an entity identifier or arbitrary attribute values, respectively. For sophisticated analyses, which often involve many table entities, generic queries induce too much latency; thus, special queries define tailored functionality for particular Views. For example, the CCT View relies on special queries to identify commonly accessed memory references across deep function call paths, but only if there is at least one write operation (i.e., a parallelism-inhibiting data dependency).

ORM

We provide an object relational mapper (ORM) module to simplify View development and improve throughput. This module enables accesses to pre-defined model entities and manages the relationships between them. The API abstracts relational trace data as entity objects with custom accessor methods to return related objects. Additionally, global methods give access to arbitrary entity objects based on identifiers or types. The API supports asynchronous and parallel accesses. The greatest benefit for Views using the ORM are optimized data loading facilities, such as caching and pipelining. Caching avoids repeated loading of data accessed in former analyses. Pipelining combines multiple queries to the same program model into single ones. When requesting numerous entities, pipelining heavily improves the throughput with only a small latency overhead.

View Manager

The visualization framework provides global state management and communication facilities for Views. The former is based on a centralized and persistent state storage to keep the state of views across user interactions. Currently, the visualization layout and the marked nodes are automatically stored as part of the state. In addition, each view can save tailored information at any time and retrieve it during rendering. Local storage is used to keep all the state information, making it persistent. This service reduces computational effort for Views by reusing results across UI events.

The visualization framework enables arbitrary Views to interact by triggering pre-defined events. These events let users explore their applications with synchronized representations from complementary viewpoints. One example is simultaneous highlighting of entities such as functions in different views. Another example is spotting of distinct entities for further inspection in separate views. This way, the number of nodes to be displayed in a View is reduced, which increases scalability. Currently, the following types of communication are provided:

- *Focus.* Focusing enhances user's attention by centering the representations in all Views.
- *Mark.* Marking allows to share selections of entities between Views.
- *Spot.* Spotting replaces the shown entities in Views by a new collection of entities.
- *Hover.* Hovering highlights entities in multiple Views by reducing the opacity of all other entities.

4.6.3 Views

In this section we present the design decisions of Parceive's Views built upon the presented visualization infrastructure. View interactions foster a scalable top-down approach to identify interesting program regions, develop parallelization strategies, and validate these strategies relative to data dependencies.

Trace View

The Trace View represents function calls as rectangular nodes by an icicle plot layout [70]: a space-filling variant of node-link tree diagrams. Instead of drawing explicit links between nodes, their placement relative to other nodes reveals the position in the hierarchy. We rely on D3.js [17] to construct the required layout.

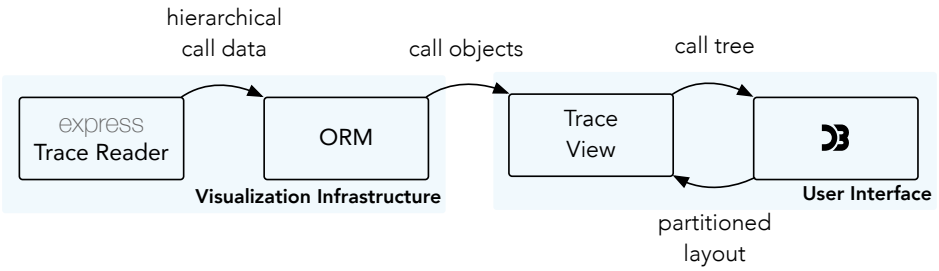


Figure 4.14: Data flow for Parceive's Trace View.

The visualization is based upon the following process:

1. *Initialization.* The Trace Reader initiates the data flow by providing hierarchical call information applying a specialized query (Figure 4.14). This data includes one entry for every function call that a) was invoked by a given caller, and b) which run time exceeds a given duration for function execution.
2. *Transformation.* The ORM module transforms the raw trace data into a list of call objects with accessor methods to parse through their hierarchy relations. The Trace View transforms the call objects into a call tree according to a hierarchical format expected by the partition layout algorithm of D3.js. The data for each call node of the tree gets extended by the algorithm with layout positions.
3. *Rendering.* Finally, the rendering is done by the Trace View to draw resulting rectangles. In case of multiple threads, the described process is applied for every called thread-start function.

CCT View

The CCT View operates on call, loop, and reference information, queried by the Trace Reader (Figure 4.15). This data is abstracted by the ORM to return their representative objects. Rather than using them, the CCT View utilizes a call graph service to maintain required state for visualization. Using such a service helps to make the View code easier and allows reuse of the call graph functionality for other Views. Examples for call graph services are children calls that can be expanded, or accessed memory references of call nodes.

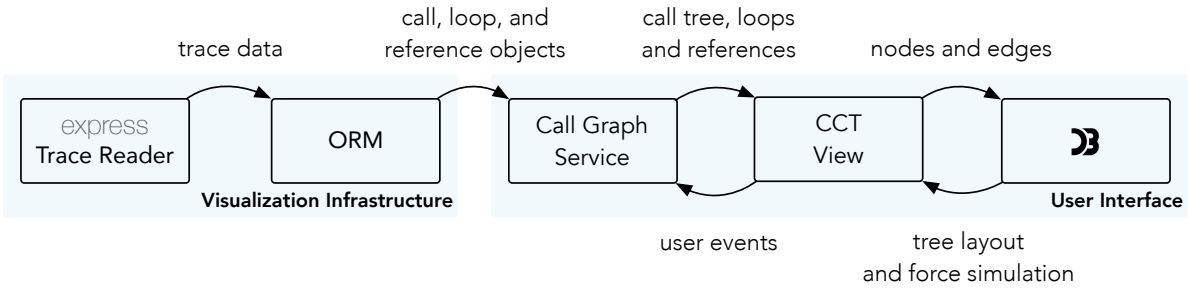


Figure 4.15: Data flow for Parceive's CCT View.

The visualization uses two different layouts for the call tree and the memory references, respectively. The call nodes are presented by a tree layout, and the memory references are shown in a network layout using force simulation to avoid cluttering of the visualization. For that purpose, the CCT View forwards the nodes and edges to the D3.js toolkit to compute these layouts. Afterwards, Nodes and edges are rendered by the CCT View. Whenever users interact with the graph, the call graph service is notified to change the internal graph state.

Architecture View

The Architecture View visualizes structural software elements and combines them with run-time information. Here, the software architecture is represented as a hierarchy of software artifacts, provided by the Architecture Builder (Figure 4.16). We rely on the WebCola.js framework [91] to generate a layout with hierarchical grouping, defined over rectangular nodes for artifacts. The layout algorithm expects artifacts and some optional links between them as input. It generates constraints for keeping the bounding boxes of disjoint artifacts from overlapping. Additionally, it preserves nested artifacts fully contained within their parent artifacts.

The Architecture View uses run-time information for two objectives. First, the artifact nodes are colored relative to their accumulated inclusive run time. The required execution times of the function calls are provided by the visualization infrastructure. Second, the View shows users specific relations between artifacts, such as shared memory accesses or function calls. We defined special queries in the Trace Reader. For example, we defined a query to obtain arbitrary deep call stacks between two specific function sets.

The visualization's design requires some tweaks to obtain the desired functionality with the layout algorithm of WebCola.js. For example, we require the visualization to show links between abstracted artifacts (i.e., group nodes) and entity artifacts (i.e., leaf nodes). However, the framework does not support edges to or from group nodes. Thus, we add such edges manually by computing the intersection of lines between the center of leaf nodes and the bounding boxes of group nodes.

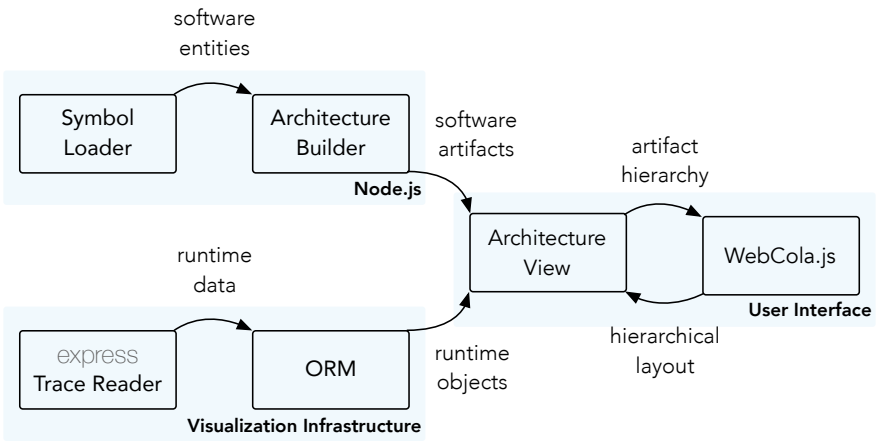


Figure 4.16: Data flow for Parceive's Architecture View.

Source View

The Source View lists source code of the analyzed program, if available. The usefulness of this visualization becomes apparent when communicating with Parceive's other Views. The simplest interaction is *focusing*, where the Source View displays functions, loops, and memory accesses. Focusing eases to follow program execution through source code. *Hovering* provides additional information; for calls it indicates where the call originated, and for memory references where they were allocated and referenced. The Source View uses location information from ORM's file objects to retrieve the source code to display (Figure 4.17).

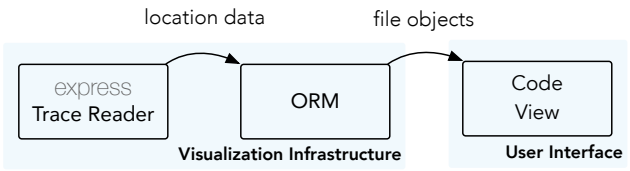


Figure 4.17: Data flow for Parceive's Source View.

4.7 Software Architecture Reconstruction

In this section we highlight design decisions for Parceive's software architecture reconstruction module. We first describe the overall architecture including its data flow. Afterwards, we present the designs of sub-components in more detail.

4.7.1 Overview

Parceive's SAR component aims for recovering and presenting a representation of a program's software architecture based on user-defined architecture rules. Two components are relevant for SAR: the frontend for user interaction, and an own-written Node.js module for the actual reconstruction step. The frontend accepts a set of abstraction rules, written in our DSL (Section 3.2.4). The included Parser component transforms these rules into a syntax tree, a required input for the reconstruction step. The other input is information about software entities contained in user applications. This data is extracted from debug symbols within application binaries by the Symbol Loader. The Architecture Builder produces the resulting software architecture as a hierarchy of software artifacts according to the given rules. Finally, these artifacts can be visualized by the frontend's Architecture View, which combines them with corresponding trace objects provided by the ORM.

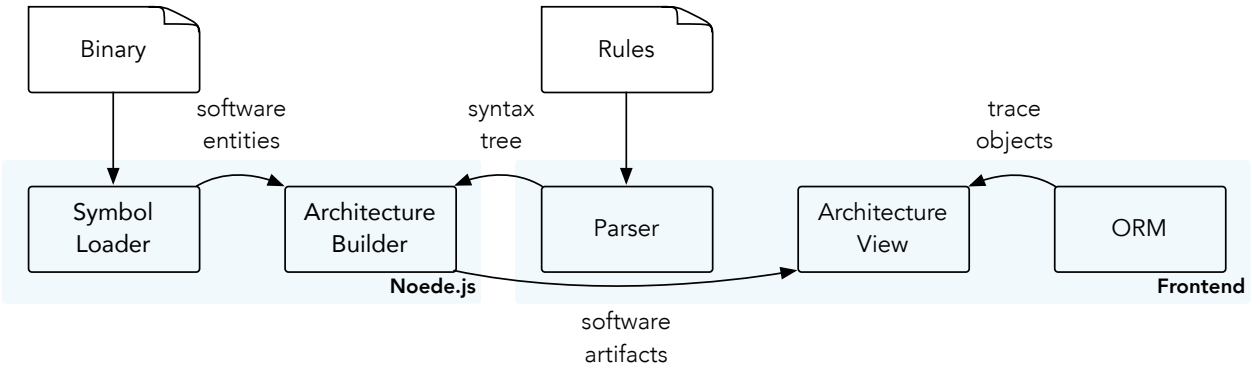


Figure 4.18: Data flow for Parceive's SAR process.

We designed the SAR component as a native Node.js module for three reasons. First, we rely on specialized frameworks to read debug symbols, which are only available as native libraries. Such tools facilitate parsing of symbols by providing a common API for the extraction of attribute values, and for navigation. Second, we reuse the backend's Symbol Loader component (Section 4.5), which improves maintainability. This component relies on frameworks for reading debug symbols to create a list of software entities. Third, the performance of native addons is superior to modules written in interpreted languages. Node.js uses the

V8 JavaScript engine [77] to provide fast mechanisms for interacting between web-based technologies and native functionality, such as creating objects, or calling functions.

4.7.2 Parser

We now describe the design of the Parser component by highlighting some key factors of its underlying grammar. The parser takes a set of architecture rules, written in our DSL (Section 3.2.4), and produces a corresponding abstract syntax tree. The Architecture Builder parses that syntax tree to reconstruct software architectures and to create detailed filters of user applications. The output has a structured JSON (JavaScript Object Notation) format, i.e., hierarchical objects for the representation of the syntax tree. The parser provides lexical and syntactical analysis to report possible input errors. The resulting error messages are very important for users to locate non-conforming rules.

The parser is generated by PEG.js [81], a parser generator for the JavaScript programming language. The generator is based on parsing expression grammar (PEG) formalism. PEG differs from context-free grammars (CFGs) by its interpretation: rather than CFG, PEG is unambiguous in parsing strings, i.e., it has exactly one valid parse tree. This yields more powerful grammars; for example, a regular expression cannot find an arbitrary number of matched pairs of parentheses, but PEG can. This strength comes with a cost: PEG requires a linear amount of memory proportional to the input whereas CFG require constant memory. However, the higher memory consumption is no practical limitation for our intended use cases.

Design

One major feature of PEG.js is its expressive grammar syntax to generate parsers. On the top level, it consists of a set of grammar rules to specify the intended language (Section 3.2.4). Each rule has the following form:

```
<name> = ( <expr>  <code>  )*
```

The rules consist of three elements: a name to uniquely identify the rule, one or more parsing expressions defining the patterns to match against the input text, and optionally some parser action that will be executed if the corresponding pattern matches. Parsing expressions are matching characters (or classes of characters) which may contain references to other rules. Parsing of rules is done from top to down, beginning with a so-called *start rule*. This order is essential to produce definite parsing trees. In our case, the order defines the precedence of the operators.

We present the grammar design of our DSL based on a subset of the used rules. Our start rule is the following Program rule, which indicates that a program, written in our DSL, consists of several statements, each defined in a separate line. The parsing expression propagates through the grammar by using a reference to the `Statement` rule, which is labeled as `s`. On a matching input text, the code returns the result of the statements. Here, the result will be some JSON objects that are propagated from the leaf rules.

```
Program
  = s:Statement* "\n"* { return s; }
```

Next, the `Statement` rule needs to be defined. Here, we differentiate between three types of statements: definitions, assignments, and filters. The first two rules diverge only in the matching characters for the used operator (`"="` or `":="`). Both require an artifact and an expression to build the respective output. `Artifact` rules represent identifiers of software artifacts. `Expressions` are hierarchies of architecture rules (see below). Note that the code section may contain function calls to external JavaScript functions. The

```
Statement
  = l:Artifact  "=" r:Expression { return buildDefinition(l, r); } /
    l:Artifact ":=" r:Expression { return buildAssignment(l, r); } /
    l:Filter        r:Expression { return buildFilter(l, r); }
```

`Filter` rule matches filters and expressions, where filters are represented as certain filter keywords, such as `include` or `exclude`.

The specification of the following `Expression` grammar rule implicitly states the precedence of operators. The `AndExpression` rule is evaluated before the `OrExpression` rule (which complies with the algebraic notation) by referencing the `AndExpression`. To support arbitrary input rules that are commutative and associative, we divide the pattern of the `OrExpression` into two sub-expressions: `head` and `tail`. Note that the pattern for the `tail` expression supports an arbitrary number of inputs that match the `AndExpression` rule.

```
Expression
  = expr:OrExpression { return expr; }

OrExpression
  = head:AndExpression tail:("||" AndExpression)* {
      return buildBinaryExpression("Or", head, tail);
    }
```

Input rules that match the pattern of the `OrExpression` will call the `buildBinaryExpression` function. This function reduces the results of the tail expressions to a nested JSON object. We rely on JavaScript's `Array.reduce` method, which applies the given function against an accumulator (initiated with the head expression) from left to right. The rest of the grammar rules follow the shown schema to handle other operators, such as negation or abstraction.

```
function buildBinaryExpression(name, head, tail) {
  return tail.reduce(function(result, element) {
    return {
      type: name,
      left: result,
      right: element[2]
    };
  }, head);
}
```

Example

We use the architecture rule `setters := function("set.*") && !image("extLib")` to depict a parser output. The rule is intended to extract all setter functions (i.e., functions that start with "set") of an user application, except those included in the extLib image. The syntax tree for the input rule includes four nesting levels (Figure 4.19). Note that some expressions are binary (`Assignment`, `AndExpression`), unary (`NotExpression`), and atomic (`RegExp`). The parser produces the right type for each matching pattern.

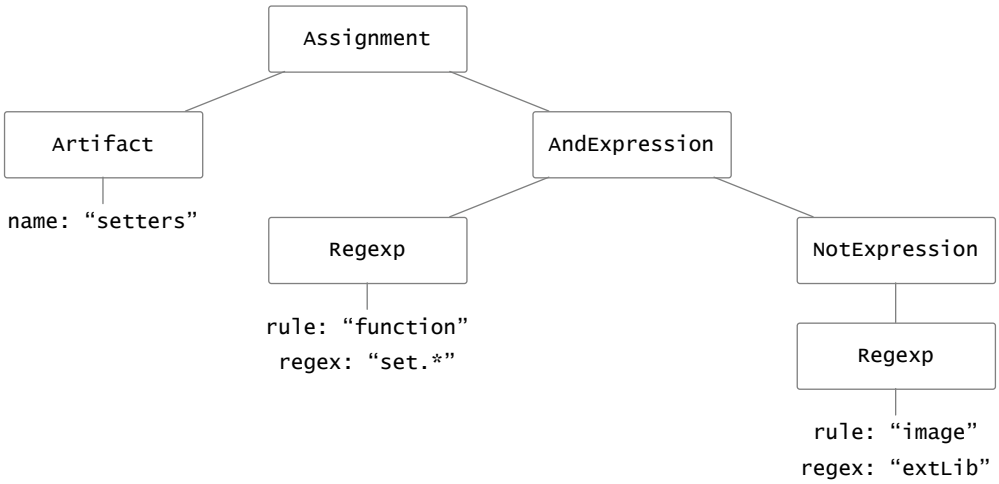


Figure 4.19: Syntax tree for the rule `setters := function("set.*") && !image("extLib")`.

4.7.3 Architecture Builder

The Architecture Builder uses given architecture rules and a set of software entities, provided by the Symbol Loader, to construct artifact hierarchies. At algorithmic level, the component is based on the visitor design pattern to traverse through syntax trees and apply respective architecture rules on an evolving set of artifacts.

We divided the Architecture Builder into three sub-components: a set of syntax nodes, the visitor component, and a set of architecture rules (Figure 4.20). Syntax nodes are generated data structures by the parser generator, which are inferred by the abstract syntax tree. Every type of syntax node has individual attributes, such as a left and a right expression in the case of the `AndOperator` type. The types inherit from the base class `Node`, which provides an interface for accepting visitors. The architecture visitor traverses through the syntax tree and uses the Architecture Builder to perform matching architecture rules on each syntax node. These rules contain the logic on how to build the architecture model, according to the given user rules. Every architecture rule inherits from the base class `Rule`, which provides a common interface to apply specific rules to a given software artifact.

Visitor

The visitor design pattern enables the definition of operations on elements of an object structure without changing the element's classes [42]. In our case, the nodes of the syntax tree (the object structure) accept a concrete visitor and send a request to this visitor to apply the architecture building (the operations). The approach represents a double-dispatch mechanism to call appropriate operations during run-time, based on two types: the visitors and the nodes.

Our tool benefits from a clear separation between the nodes of the syntax tree and the respective methods for architecture building. Mixing these concerns would lead to a system that is hard to understand, maintain, and change. The logic for traversing through the syntax tree and applying the architecture builder is defined in a single visitor class. Another benefit is the ability to define and accumulate state in concrete visitors without passing it as arguments. For example, architecture building keeps the result of expressions and applies them to the containing statements afterwards. Finally, the pattern eases the integration of further operations on the syntax tree by simply adding new concrete visitors. A possible extension for Parceive is conformance checking, i.e., to check if the defined architecture of a software system conforms with the intended architecture.

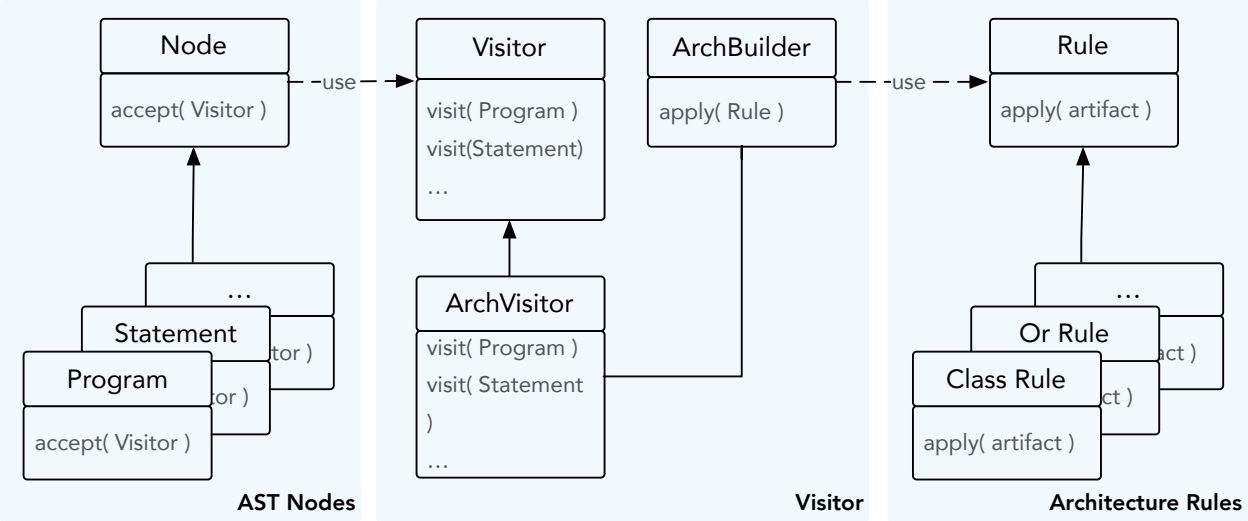


Figure 4.20: Visitor pattern used for Parceive's SAR approach.

Architecture Builder

The Architecture Builder (Builder) controls the construction of a software architecture model and dumps it to an output file. The Builder applies the visitor's requested architecture and filter rules on an evolving set of software artifacts. On each rule invocation, a list of software entities and a current working-artifact hierarchy is passed by the Builder. The rules then evolve the artifact hierarchy, according to their specific logic. Initially, the Builder includes two empty artifacts; one is filled by architecture rules and the other is filled by filter rules.

Sub-expressions are evaluated before the containing statements or expressions. Thus, the Builder stacks expressions according to their appropriate evaluation order. After all rules have been applied to the artifacts, the Builder writes the resulting artifact hierarchy to a file in a structured JSON format. This file is used by the frontend and the backend to visualize the reconstructed software architecture and for filtering, respectively.

Architecture Rules

The architecture rules are classes that encode the construction of software architecture models. They inherit from the abstract class `ArchRule`, which specifies methods for applying the rules. Every rule takes a software artifact and an optional set of software entities as input to modify the artifact hierarchy. Logically, we divide architecture rules into entity rules and operator rules. Entity and operator rules are used for entity keywords and operators, respectively.

The following list describes the entity rules that create new (sub-)artifacts for entities with an expected entity name relative to a given regular expression. Entity rules can be either applied on all software entities of a user application or on a constrained set of entities. The latter case enables utilization of entity rules by other rules (e.g., to combine entity rules with operator rules). Note that we presented the formal definition of the rules in Section 3.2.

- `VariableRule.` This rule creates a new artifact for every variable with an expected variable name.
- `FunctionRule.` This rule creates a new artifact for every function with an expected function name. Additionally, it utilizes the `VariableRule` to append the function's local variables as sub-artifacts.

- `ClassRule`. This rule constructs a hierarchy of classes. It creates a new artifact for every class with an expected class name, and links it relative to inheritance or nesting. Implicitly, the rule applies the `VariableRule` and `FunctionRule` to create sub-artifacts for member variables and methods, respectively. Additionally, instantiations of a template class are grouped to the same template artifact.
- `NamespaceRule`. This rule identifies (nested) namespaces with an expected name and creates an artifact for each of them. Additionally, the rule uses the `ClassRule`, `FunctionRule`, and `VariableRule` to append contained sub-artifacts to the namespace artifacts.
- `ImageRule`. This rule creates a new artifact for each image with an expected file name. It utilizes the `NamespaceRule`, `ClassRule`, `FunctionRule`, and `VariableRule` to append contained sub-artifacts to the image artifacts.

Now, we describe the design of the operator rules that encode the application of operators. The interested reader can find the formal definitions of these operators in Section 3.2.

- `AndOperatorRule`. The rule accepts two artifacts and returns a new artifact containing their common sub-artifacts. Identifying common abstraction artifacts (i.e., artifacts that do not map to existing software entities) is not trivial since these artifacts have no common identifier for comparison. Thus, we encoded the rule by three consecutive steps. During the first step, the rule removes artifacts relative to variable and function entities that are not common to both input artifacts. During the second step, artifacts that are not associated with variables or functions, and have no sub-artifacts, are removed since they are not contained in the operand's intersection. In the final step the top-level artifact is aligned with both operand artifacts, i.e., the resulting top-level artifact is the lowest top-level artifact that exists in both operand artifacts.
- `OrOperatorRule`. The rule returns a new artifact by merging two input artifacts. The rule collects all entities from both operand artifacts and applies the `NamespaceRule`, `ClassRule`, `FunctionRule`, and `VariableRule` on them.
- `NotOperatorRule`. The rule returns artifacts related to the software entities of the user application that do not exist in a provided artifact hierarchy. Thus, it initially identifies all entities related to the operand and ones in the rest of the user application. Then, the rule creates new artifacts relative to the rest of the entities by utilizing the `NamespaceRule`, `ClassRule`, `FunctionRule`, and `VariableRule`.
- `SetOperatorRule`. The rule copies the artifact hierarchies of a given set of artifacts to a new set artifact. If the operands are named artifacts (i.e., non-temporal artifacts), the corresponding sub-artifacts get the same name. Otherwise, the new sub-artifacts are unnamed.

4.8 Conclusion

In this chapter we presented the design of the Parceive tool. Particular attention has been paid to a high modularity at all levels of abstraction to ease reusability and reduce duplication. For example, the backend uses the Symbol Loader component for resolving symbol names, and the frontend uses it for reconstructing software architectures from binaries. Another important aspect is the interaction of (sub-)components relative to the outlined requirements, such as performance or portability. Here, the trace database acts as main interface between the backend and the frontend to exchange run-time information. We explained how Parceive collects such run-time information and how the mapping to software architecture data is achieved. The design has proven to be flexible enough for a variety of input applications on multiple platforms.

5

Case Studies

> *"There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks."*
>
> – Leslie Lamport

We demonstrate Parceive by applying our proposed approach on two case studies. The first case study is *WMSim*, a program to simulate the results of the 2018's soccer championship. The second case study is *CppCheck*, an open source tool for static analysis of C and C++ programs.

Our objective is to illustrate the described principles with realistic, but not overly complex use cases. To be realistic, we assume the programmer has only a vague understanding of the problem and program domain. Necessary comprehension will be established while conducting the proposed steps. We attempt to identify the potential parallelism of the applications, justify the parallelization effort and find appropriate scenarios. For that purpose, our tool uses multiple executions with varying input data to extract the required information from the application binaries. The resulting visualizations aim to support our design decisions for parallelization scenarios. In the case of WMSim, we parallelize the software and validate the resulting implementation. In the case of CppCheck, we identify a parallelization scenario similar to the one implemented by the original authors.

5.1 WMSim

Our example program *WMSim* simulates the 2018's soccer World Cup by relying on historic match data. This simulator has been developed by our group as a student assignment for the graduate course *Parallel Programming*. The students were introduced to the tournament's match schedule and its realization in sequential software. Their task was to understand, redesign, and parallelize WMSim using different parallel programming models, such as Posix Threads or OpenMP.

Although WMSim is clearly no industrial-size application, it favorably substantiates the explanation of our proposed approach for two reasons. First, the simulations contain sufficient exploitable concurrency to benefit from parallel execution. As with most industrial applications, the concurrency is scattered across multiple sub-problems, and the dependencies that limit parallelism are irregular. Second, the problem's inherent complexity and the software's accidental complexity are appropriate for a textbook example. The problem and program domain is easy enough for comprehension and complex enough for challenging design decisions.

Soccer World Cup

Good comprehension of the World Cup's match schedule is essential to understand the structure, behavior, and particularly the exploitable concurrency of WMSim. During the tournament, 32 qualified national soccer teams participate to compete for the world champion title. The World Cup is separated into two subsequent stages, an initial group stage and a final knockout stage.

For the group stage, all teams are distributed into eight groups with four teams per group. Every team plays a series of three matches against the other teams in the same group to qualify for the subsequent knockout stage (Figure 5.1). The team that scores more goals during a match wins and gets three points, the other team gets no points. On a draw (i.e., no team wins) both teams get one point. In real tournaments, different matches of one team cannot be played at the same time.

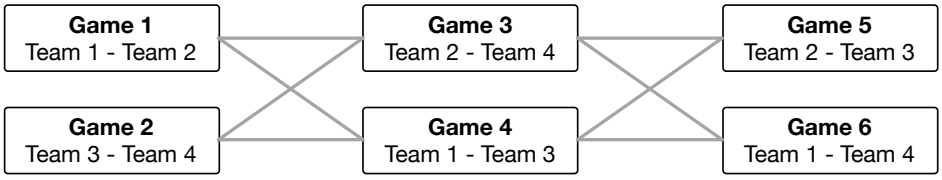


Figure 5.1: Matches and dependencies of the soccer World Cup's group stage.

After all group matches have been played, the first and second winner of each group is qualified for the knockout stage. The following rules determine the ranking in the presented order. If the rules do not lead to a definite order, there are five other rules which we omit for clarity reasons (e.g., considering fair play points, and drawing of lots).

1. Greatest number of points obtained in all group matches.
2. Goal difference in all group matches (scored versus received goals).
3. Greatest number of goals scored in all group matches.

Unlike the group stage, all matches during the knockout stage result with clear winners. Draws are excluded by providing overtime periods and penalty sessions. The matches are structured in a binary tree starting with the leaf matches, which represent the round of the best 16 teams of the group stage (Figure 5.2). Only winning teams of each match progress to the next knockout level until the final match that determines the world champion. The individual pairings of the matches are given by a static scheduling. Note that a single team participates at most in one match of a particular knockout level, i.e., there are no dependencies between matches at the same level.

WMSim Simulator

WMSim is a standalone program to simulate the FIFA® soccer World Cup. It estimates the result of every match by performing simple heuristics on historic tournament data from the last 70 years.

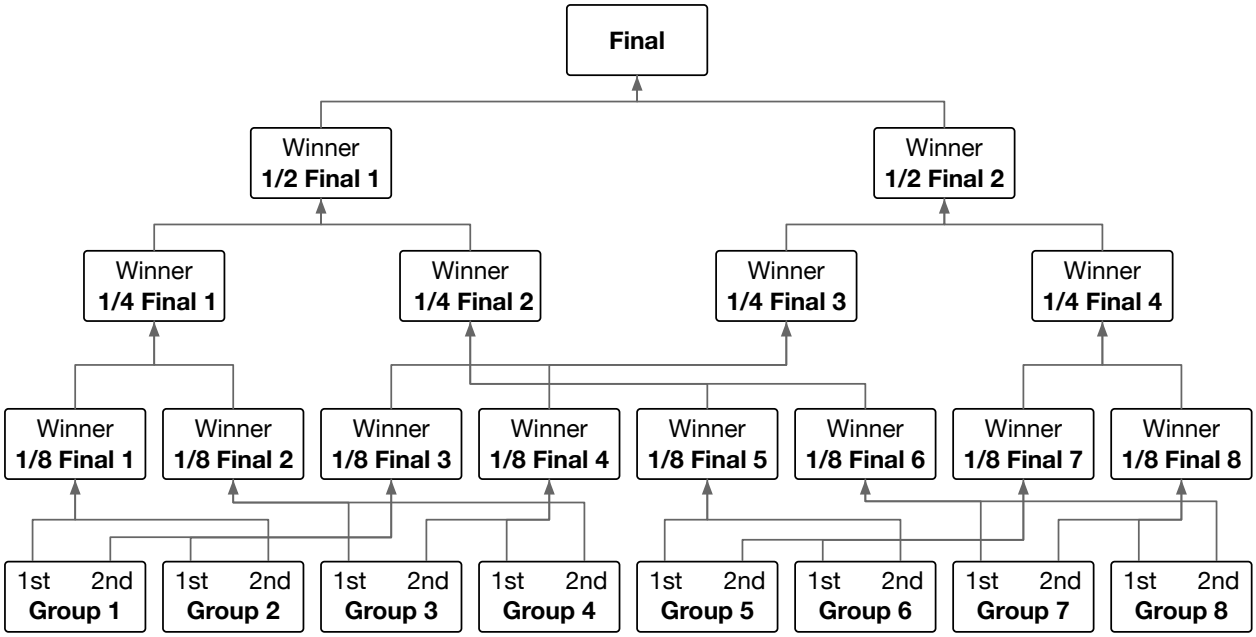


Figure 5.2: The knockout stage of the 2018's soccer World Cup.

The program's software architecture comprises four main components: a data loader, a driver, a simulator, and a visualization component (Figure 5.3). The historic match data is given as a SQLite database which contains information about every World Cup match played since 1954, including detailed statistics about the involved players. The data loader is responsible for the database connection and provides a simple interface to retrieve data. Initially, the driver component obtains information for every qualified team of the soccer World Cup. Afterwards, this component controls the group and knockout stages, and forwards the matches to be played to the simulator component. During the simulation process, the visualization component dumps all simulated match results to the command-line.

WMSim simulates every soccer match between a team A and a team B by conducting the same two steps. First, the simulator computes the average result of all played matches between A and B. The average number of goals scored are aggregated and multiplied with a time factor to rate current results higher than older ones. Second, the algorithm sums up the number of total goals scored by players that played in any match between A and B. Although these heuristics hardly correlate with real winning factors of soccer matches, their computation requires significant run time effort sufficient for parallelization.

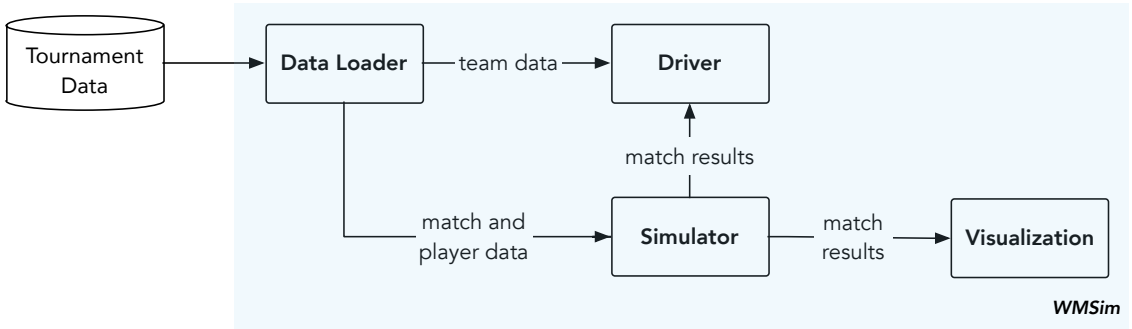


Figure 5.3: The software architecture of WMSim.

5.1.1 Justifying Parallelization Potential

We evaluate if the soccer simulation contains enough exploitable concurrency to justify any parallelization effort. We thus initially explore its application environment by executing the simulator program. Its most relevant parts are the database with historic soccer results as input, the execution process, and the match results as output. On our target system, WMSim simulates the 63 matches for the group and knockout stages of the tournament in 4,6 seconds.

Next, we use Parceive's Trace View to further inspect the run time distribution across all call paths. The visualization indicates that the vast majority (99%) of the run time is used for the actual simulation, with roughly two third of the run time (60%) spent in the group stage and the rest (40%) spent in the knockout stage (Figure 5.4). During the other 1% of the run time, WMSim invokes the `getTeam` function multiple times (not shown), probably to initialize data for each team.

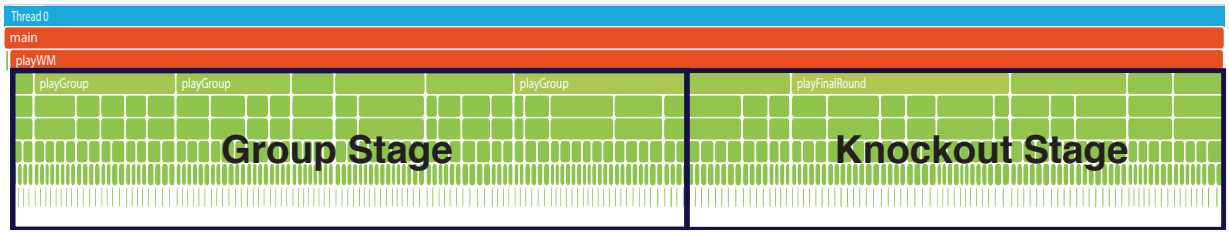


Figure 5.4: Parceive's Trace View for a traced execution of WMSim.

For justification, we assume a sequential data initialization phase which run time is constant for different input sizes. Furthermore, we assume to have unlimited processing units, and the simulation scales perfectly with any number of those processing units. By relying on Amdahl's law, we can estimate a maximal speedup of 100 (see below). Although the assumptions are too optimistic (e.g., dependencies and parallelism overheads are not considered), the potential speedup justifies a restructuring to introduce parallelism.

$$Speedup \leq \frac{1}{(1-0.99)+\frac{0.99}{\infty}} = 100$$

5.1.2 Partitioning

To identify parallelization scenarios for WMSim, we require a good understanding of the simulation process and its algorithmic realization. The starting point is our intuition about the World Cup simulation, which applies the same set of operations on each of the 63 matches. Currently, we are not aware of the operation semantics, at which granularity level they are operating, and what data they are using. However, we know certain conceptual dependencies between different phases of the tournament, such as the relation between matches of the group stage and the round of best 16. This knowledge leads to scenarios that simultaneously calculate results for different groups, and for different matches in the same knockout levels.

We use Parceive to support this intuition. A closer look at the resulting CCT View reveals details about the inspected function call hierarchy (Figure 5.5). Shown nodes represent single function calls made during execution, or aggregated calls of the same function with appended call counts. At highest level, the simulation (the single call of function `PlayWM`) encapsulates the group and knockout stage computations (the calls to the function `playGroup` and `playFinalRound`) into eight and four sub-problems, respectively. Note that the run times for simulating the group stage and the knockout stage varies.

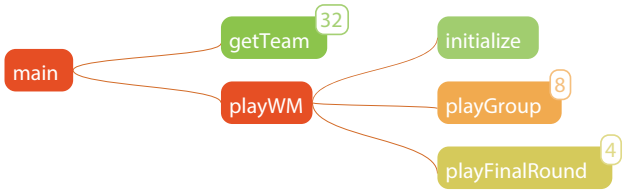


Figure 5.5: Parceive's CCT View showing the top of WMSim's function call hierarchy.

```
void playWM(team_t* teams)
{
  team_t* winner_teams[NUM_GROUPS * 2];

  initialize(teams);

  for (int g = 0; g < NUM_GROUPS; g++)
    playGroup(teams + g * NUM_TEAMS_PER_GROUP,
              winner_teams + g * 2,
              winner_teams + NUM_GROUPS * 2 - g * 2 - 1);
  ...
}
```

Listing 5: The calls of function `playGroup` within a loop.

By navigating to the call site of the `playGroup` function in the Source View, we can see the provided arguments. The used pointer arithmetic indicates that WMSim stores team data in a single array (Listing 5); one array element per team and all elements are ordered by their corresponding group. This leads to the first parallelization scenario.

S1: Data Decomposition of Groups. This parallelization scenario proposes a decomposition of the group stage to simulate each group simultaneously. The array with team data for individual groups will be partitioned into separate data chunks. A functional decomposition that operates on these data chunks follows naturally due to the `playGroup` function, which already exists in the code. We are not aware of any dependencies between different groups caused by the problem domain (we will check this assumption later with Parceive). If we assume equal run times for the group simulations, the maximal speedup of this scenario is:

$$Speedup(S1) \leq \frac{1}{(1-0.6)+\frac{0.6}{8}} = 2,105$$

To improve our comprehension of the simulation process, we further inspect the operations performed during the group and knockout stage. Our objective is to identify common computations that may lead to new scenarios. The CCT View features expansion of arbitrary call paths, which we use for analyzing the call paths for both stages (Figure 5.6). Note that we expanded only one group and final stage for clarity.

Every soccer match played during the group stage is effectively performed by a distinct call of function `playGroupMatchGen`. For the knockout stage, we expanded the call paths for different knockout levels (the final match's call path is shown). Every match during the same knockout level is simulated by a single call of the `playFinalMatch` function. Each call executes the same generic simulation function as

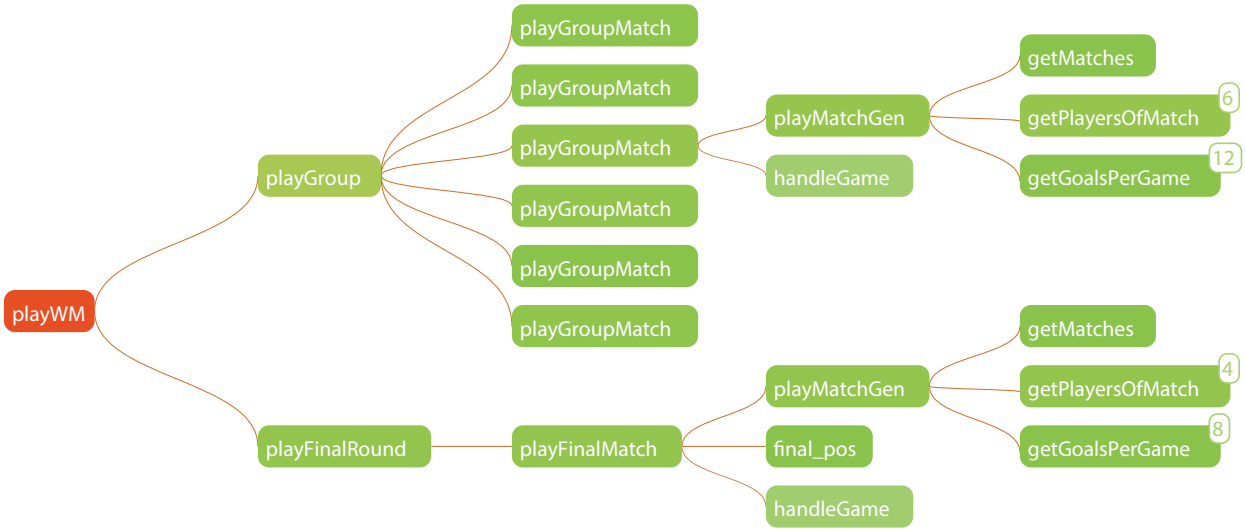


Figure 5.6: Parceive's CCT View showing more details of WMSim's function call hierarchy.

for the matches during group stage. Parallelization scenarios should exploit the concurrency of this regular operation structure.

S2: Functional Decomposition of Final Matches. This scenario proposes parallel execution of all matches within one final stage. Simulation of matches between different final levels have a causal dependency and cannot be performed simultaneously (Figure 5.10). The participating teams of one knockout level are determined by match results from preceding knockout levels. However, matches of the same knockout level are independent. Note that the exploitable concurrency varies between different knockout levels, ranging from eight (round of best 16) to one (final). According to the program's recorded run times, scenario S2 results in the following maximal speedup:

$$Speedup(S2) = \frac{1}{(1-0,4) + \frac{0,213}{8} + \frac{0,107}{4} + \frac{0,053}{2} + \frac{0,027}{1}} = 1,415$$

Performance of parallelization scenarios highly depend on their component's granularity. Optimal solutions provide enough work to efficiently utilize all processing units and do not require unpractical concurrency overhead. Scenario S1 and S2 propose parallelism with coarse granularity based on single match simulations. Scenarios with finer granularity exploit more concurrency and might imply higher scalability.

We further analyze the `playGroupMatchGen` function to understand the used algorithm for match simulations. Within this function, the simulator fetches data for every historic match between two teams (performed in function `getMatches`), and about all players participating in these matches (performed in function `getPlayersOfMatch`). Then, our encoded metric estimates match results by aggregating and comparing the scored goals of both teams and their players.

Parceive's CCT View shows that most run time ($\approx$ 98%) is used for obtaining player information (Figure 5.4). Scenarios that simultaneously obtain this information for different matches may exploit high concurrency. We assume that querying the database for different teams or players is independent, based on our domain knowledge. However, the sequential programming model causes causal dependencies between these tasks. The function call hierarchy demands that single executions of `getPlayersOfMatch` must

finish before other matches can be simulated. Hence, parallelization scenarios must perform those queries asynchronously to exploit most concurrency.

Another issue is highlighted by the function's call counts in the CCT View. They indicate that the number of obtained matches between different teams varies, relative to existing database entries. This leads to unbalanced workloads, which limits resulting parallelism. Asynchronous execution naturally alleviates this issue by relying on processing frameworks that provide dynamic scheduling mechanisms.

S3: Task Graph. Parallelization scenario S3 proposes a different software design based on the task graph pattern. Here, the simulation is considered as a collection of individual tasks with pre-defined input and output dependencies, representing a directed acyclic graph. Single tasks can have multiple input dependencies and may only start to work if all their predecessor tasks have completed. This structure matches favorably with non-trivial task relationships, such as the result propagation within WMSim.

The proposed task graph starts with the simulation of group matches (Figure 5.7). All groups are simulated by individual tasks that can be performed simultaneously. To improve scalability, each of these tasks may simulate the contained group matches in parallel (e.g., by using a fork join pattern).

Matches within the knockout stage represent single tasks within the task graph. Each task depends on the respective match results from the preceding knockout or group level. Note that these dependencies enable even parallel simulations of matches across different knockout levels. The execution of the whole task graph stops when all tasks have been completed.

More parallelism is achievable by using dynamic tasks to obtain player data for match simulations. These tasks may be executed simultaneously with ones from other matches due to the asynchronous execution policy of the task graph.

Note that the task graph pattern is not the only one which allows asynchronous execution of static and dynamic tasks. Another example is the actor model, which relies on computational entities - so-called *actors* - that concurrently perform defined actions and send messages to other actors. In this thesis we use the task graph pattern because of its easier descriptiveness.

The speedup of scenario S3 can be computed as

$$Speedup(S3) \leq \frac{1}{\frac{0,4}{48} + \frac{0,213}{8} + \frac{0,107}{4} + \frac{0,053}{2} + \frac{0,027}{1}} = 8,680$$

5.1.3 Validation and Redefinition

Now we validate and refine the identified parallelization scenarios to develop a parallel software design for WMSim. The final design may comprise a composition of scenarios that complement each other. These scenarios can operate on distinct problems or different levels of abstraction. Validation aims to find dependencies between computational elements of scenarios. Unconsidered dependencies may require a refinement of intended design decisions. Different qualities must be considered for the refinement, such as performance, portability, or elegance. This requires a proper understanding of relevant structural and behavioral aspects within the program domain, such as data flow between software components, or the implemented database management. We conduct this step for every parallelization scenario individually before we determine the final design. Parceive supports us with special analyses and visualizations showing decisive information.

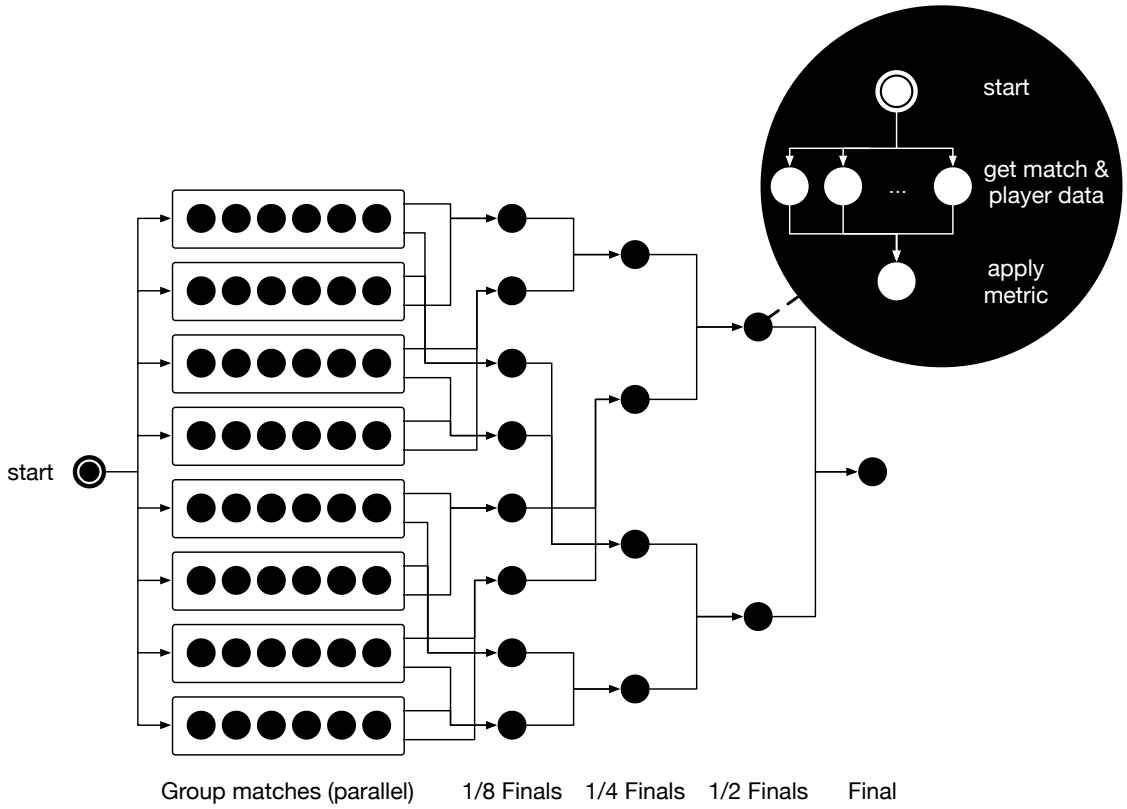


Figure 5.7: The task graph for WMSim's match simulation, as proposed by scenario S3.

Scenario *S1*

We begin with the validation of scenario S1, which proposes the parallel simulation of individual groups. The relevant source code simulates every group by one iteration of a loop (Listing 5). Each iteration operates on four subsequent elements of an array with team information, which are ordered by group number. Parallelization with the parallel loop pattern seems to be an obvious choice here. This pattern enables a parallel execution of individual iterations by separate processing units. However, this parallelization is invalid if any conflicting data dependencies exist between iterations, which leads to problematic race conditions. Identifying such data dependencies is challenging in the given context. We must precisely compare the accessed memory locations during different group simulations. Essentially, this requires precise data flow analysis across all executed (assembly) instructions.

We use Parceive to identify shared memory locations between simulations of different groups. Our tool collects data regarding accessed memory locations during run-time, which then can be analyzed using tailored visualizations. For WMSim, the used analyses must handle arbitrary deep call stacks, external libraries, and language specific peculiarities, such as pointer aliasing, indirect function calls, or generic programming.

After the simulator has been executed by Parceive, we use the CCT View to analyze the group stage simulation. We navigate to the respective loop and expand the eight iteration nodes. Each iteration simulates one tournament group by calling the function `playGroupMatch`. We then mark each iteration and apply the multi-level data dependency analysis on them. This analysis searches for memory locations that are accessed within multiple group simulations. To be relevant for parallelism, at least one access of each memory location must be a write access.

The analysis identifies more than 15.000 heap objects that are commonly accessed by the group simulations. Fortunately, Parceive groups these objects by their size, type, and allocation site. This abstraction lets us identify the main reason for the overwhelming amount of data dependencies. Almost all accesses origin from the used sqlite3 library that effectively performs the database queries. The operations of this library are currently not thread safe, i.e., performing multiple operations in parallel may lead to concurrency issues. According to the library's documentation, a synchronized operating mode must be used (by enabling a runtime switch) for parallel execution. Additionally, each thread must use a separate database connection to be thread safe. The latter prerequisite effectively requires a redesign of the database management component. This component must create, organize, and assign single database connections to each accessing thread.

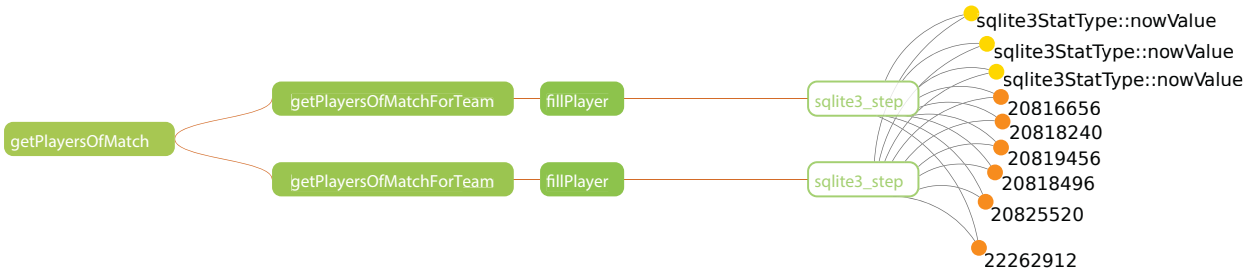


Figure 5.8: Parceive's CCT View showing data dependencies between sqlite3 library calls.

We now focus on data dependencies that are not related with the sqlite3 library. The CCT View shows only one other memory location that is accessed during multiple group simulations. Its origin is the static variable `num_total_goals` in WMSim's visualization component (Figure 5.9). The numeric variable gets incremented by a function located two call levels below `playGroupMatch` in the call tree. Source code usages indicate that the variable is used to aggregate the number of goals scored in all tournament matches. We can avoid the resulting race condition in scenario S1 by synchronization of the aggregation operations, e.g., by using an atomic variable. The performance overhead incurred by this synchronization is negligible for one aggregation per match simulation.

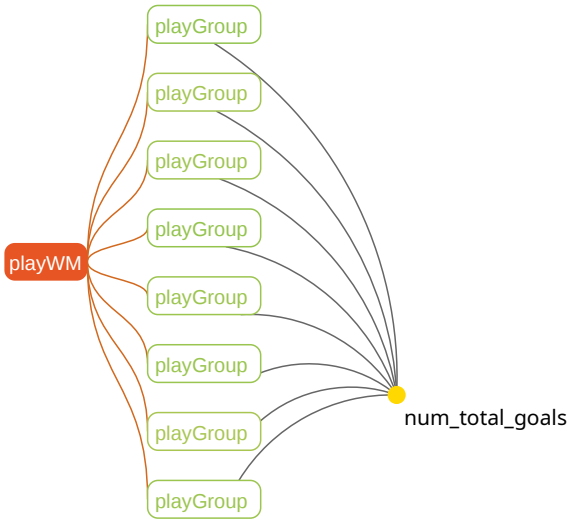


Figure 5.9: Parceive's CCT View shows common data accesses between calls of `playGroup`.

Scenario *S2*

Scenario S2 proposes the parallel simulation of matches within single levels of the final stage. We validate this scenario analogously to the previous procedure, except for the concerned level of abstraction. Rather than applying dependency analysis on groups of matches, we focus on single matches. Each final stage level is simulated by one call of the generic `playFinalRound` function. This function performs every match simulation by calling the function `playFinalMatch` in a loop (Listing 6).

```
void playFinalRound(int numGames, team_t** teams, team_t** successors)
{
  int i, goals1 = 0, goals2 = 0;

  for (i = 0; i < numGames; ++i)
    playFinalMatch(teams[i*2], teams[i*2+1], &goals1, &goals2);
  ...
}
```

Listing 6: The function calls of `playFinalMatch` for every final stage.

We mark and apply Parceive's multi-level data dependency analysis on the call nodes to identify shared memory locations between match simulations at the final level. The analysis returns three commonly accessed memory locations; the static variable `num_total_goals`, and two stack variables `goals1` and `goals2` (Figure 5.10). The former data dependency is known from scenario S1 and may be resolved by synchronization (Figure 5.9). The latter two dependencies are caused by WMSim's sequential software design; every execution of function `playFinalMatch` writes simulated match results to the same two variables on the caller's stack via references. Safe parallelization must consider these dependencies, e.g., by variable privatization for every concurrent match simulation.

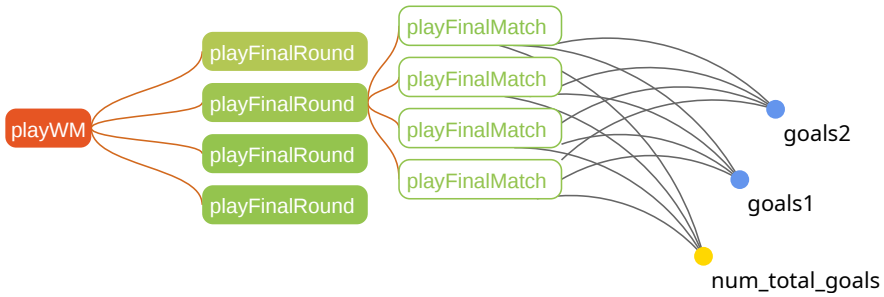


Figure 5.10: Parceive's CCT View shows common data accesses between function calls of `playFinalMatch`.

A combination of scenario S1 and S2 is plausible since both operate on temporal distinct simulation phases; the group and the final stage, respectively. The maximal achievable speedup is:

$$Speedup(S1 + S2) \leq \frac{1}{\frac{0,4}{8} + \frac{0,213}{8} + \frac{0,107}{4} + \frac{0,053}{2} + \frac{0,027}{1}} = 6,375$$

Scenario *S3*

Now we evaluate our final scenario from the previous step, which proposes a parallel design based on the architectural task graph pattern.

Our speedup estimation indicates that scenario S3 performs better than the described combination of scenarios S1 and S2 (S1+S2: 6,375 - S3: 8,680). Essentially, this speedup advantage stems from the more fine-grained parallelism during the group stage. Scenario S3 performs each simulation of individual group matches by single tasks. Additionally, the task graph facilitates overlapping computations between the group and knockout stage, and different levels of the knockout stage. The simulator can start any match simulation as soon as the required input results from the precedent matches are computed. For example, semi final 1 and quarter final 3 can be simulated in parallel after completion of quarter final 1 and 2. Additional performance improvements may be gained by fetching player data asynchronously within every match simulation. Before we can create a parallel design to realize scenario S3, we must validate its computational elements.

Initially, we focus on validation of the parallel simulation of group matches. During the validation of scenario S1 we already identified dependencies between simulations of whole groups, i.e., data dependencies between iterations of the respective loop (Figure 5.9). Note that scenario S3 may contain different data dependencies due to its lower level of abstraction. Appropriate software entities for this validation are executions of the `playGroupMatch` function (Listing 7). The function calls `playMatchGen` to simulate the matches, which writes the simulated results to the stack variables `goals1` and `goals2`. Afterwards, the function `visualizeGame` is called to transfer the match data to the visualization component. We apply Parceive's multi-level dependency analysis on respective call nodes in the CCT View. The results indicate no additional data dependencies between simulations of different groups, i.e., the only found dependencies are caused by accesses of `num_total_goals`.

```
void playGroupMatch(int groupNo, team_t* team1, team_t* team2)
{
  int goals1, goals2;
  ...
  playMatchGen(team1, team2, &goals1, &goals2);
  team1->goals += goals1;
  team1->goals -= goals2;
  team2->goals += goals2;
  team2->goals -= goals1;
  ...
  visualizeGame(gameNum[groupNo]++ + NUMGAMESPERGROUP * groupNo,
                team1->name, team2->name, goals1, goals2);
}
```

Listing 7: Excerpt of the `playGroupMatch` function that simulates matches during the group stage.

When analyzing simulations within the same groups, the dependency analysis detects two new types of data dependencies (Figure 5.11). The first type is caused by accesses of the global array `gameNum`, which stores the number of simulated matches for each group. This number is incremented after each match simulation. Parceive correctly rates array accesses between different groups as independent because of the distinct array indices (representing the group numbers). The second type of dependencies are identified between group matches having the same teams as one opponent. A closer inspection reveals that the simulator aggregates the total number of scored goals for every team within the corresponding team objects. Note that all related orange (heap) data nodes are equally labeled; this indicates that the calls access different elements of the same array. These dependencies reflect the match plan of the tournament's group stage. Parallel execution of matches with the same teams may lead to race conditions and distort the simulation results; hence, the task graph must be specified accordingly.

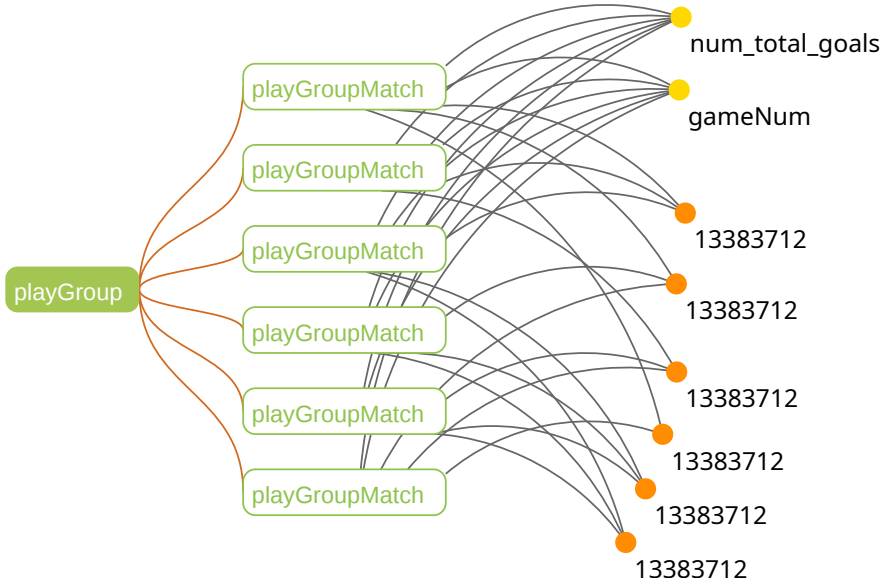


Figure 5.11: CCT View showing common data accesses between calls of function `playGroupMatch`.

After the proposed tasks within the group stages have been analyzed, we validate other match combinations that may be simultaneously simulated by the task graph. These are matches from different stages or matches from different levels of the knockout stage. For that purpose, we apply Parceive's multi-level dependency analysis on the call nodes for the functions `playGroupMatch` and `playFinalMatch`. Besides the already known dependencies caused by accesses of variables `num_total_goals` and `goals`, the analysis identifies one new group of data dependencies. There are data dependencies between simulations of different knockout levels. The causes are accesses to the stack array `successors`, which is reused as argument for every level simulation. These accesses explicitly expresses causal dependencies from the problem domain at implementation level.

5.1.4 Restructuring

After evaluating our identified parallelization scenarios, we develop a parallel software design based on scenario S3 that considers the found data dependencies. Naive designs use synchronization to avoid simultaneous accesses of shared variables. However, synchronization may slow down execution, and increases code complexity by obfuscating the inherent dependencies of the problem domain. For example, synchronizing accesses of team objects obfuscates concrete dependencies between match simulations of the same teams; resulting code misleadingly indicates dependencies between all match simulations. Fortunately, the intended task graph provides syntactical means to explicitly define task dependencies. This enables a parallel solution that is inherently synchronized by the causal order of its task graph.

However, scenario S3 requires some profound changes in WMSim's software design. Most importantly, implementation of task graphs requires special parallelization frameworks. These frameworks provide syntactic means to specify tasks and their dependencies, such as library calls or external configuration files. The frameworks include a runtime for static and dynamic task management, which usually rely on task pools to reduce task creation overhead. We benefit from such abstract parallelism by improved performance portability.

Unfortunately, the current implementation complicates the realization of a task graph. The data structures used for communication between match simulations are designed for sequential accesses with incremental

array indices. Decomposing these accesses into multiple tasks and dependencies would lead to code that is hard to understand and maintain; thus, we propose redesigning the abstractions for the simulation process and its necessary communication structures. This redesign affects two levels of simulations: for single matches, and for whole groups. The former expects both playing teams as input and returns the simulated result. The latter encodes the static group scheduling as nested task graph with single match simulations. The resulting task graph improves the understandability of the simulation process by making its dependencies explicit.

To prepare the intended changes, we performed the *extract function* refactoring on the `playWM` function to separate the different phases and levels of the tournament simulation (Listing 8). Note that the simulation results are explicitly stored in variables. This methodology is required to conform with the intended task graph API (see below).

```
void playWM(team_t* teams)
{
   // play groups
   for (g = 0; g < NUMGROUPS; ++g)
      playGroup(g, teams + (g * cTeamsPerGroup), cTeamsPerGroup,
                successors + g * 2, successors + (numSuccessors - (g * 2) - 1));

    // play final rounds
    while (numSuccessors > 1) {
       playFinalRound(numSuccessors / 2, successors, successors);
       numSuccessors /= 2;
    }
}
```

⇓ Extract function ⇓

```
void WM::playWM()
{
   auto teams_best16  = playGroupPhase();
   auto teams_quarter = playRoundOfBest16(teams_best16);
   auto teams_semi    = playQuarterFinals(teams_quarter);
   auto teams_final   = playSemiFinals(teams_semi);
   playFinal(teams_final);
}
```

Listing 8: The *extract function* refactoring for WMSim.

Scenario S3 proposes to asynchronously obtain player information for each match simulation. This potentially improves performance by providing more fine-grained tasks to balance workloads. To efficiently utilize available processing units, the same task management must be used as for the task graph. Note that the amount of query operations vary for each match depending on the historic match data; thus, obtaining player information requires tasks with dynamic dependencies rather than the design time dependencies from the described task graph. To identify dependencies between the intended tasks, we mark all call nodes of the function `getPlayersOfMatch` and apply Parceive's multi-level dependency analysis. The analysis returns no common accesses during the analyzed execution. Realization with a fork join pattern is the easiest way to introduce parallelism.

Scenario S3 implies various concerns for WMSim's overall design. Most importantly, implementation of task graphs requires special parallelization frameworks. These provide mechanisms to specify tasks and their dependencies, such as library calls or external configuration files. These frameworks include a runtime for static and dynamic task management, which usually rely on task pools to reduce task creation overhead. Users benefit from abstract parallelism without the need to adjust their solutions for different hardware architectures. Additionally, the proposed task graph improves the understandability of the simulation process by making its dependencies explicit.

5.1.5 Parallelization

Now we describe a possible parallelization of WMSim based on scenario S3. The conducted process illustrates typical decisions that must be made during such tasks. Additionally, the resulting software design endorses our claim that restructuring sequential software at various levels is important to achieve expressive parallelism.

We use the Intel® *Threading Building Blocks* (TBB) library for the parallelization [100]. TBB's main intention is to abstract from the threading model and provide higher-level parallelism tailored towards patterns. The library comprises task schedulers, data containers, and computational algorithms at different levels of abstraction. It strives for portability by dynamically managing threads. Its API relies on generic programming for broad applicability across multiple data types.

Parallelization scenario S3 proposes the arrangement of match simulations within a task graph. TBB fosters this structure by providing explicit graph parallelism. Its core is a *flow graph* with nodes and edges that represent computations and communication channels between them. Messages flow through the graph across the edges that connect the nodes. When a node receives a message, one task is spawned that operates on this message. After the task computed its result, it gets forwarded to all connected nodes. At the end, the graph algorithm waits for all nodes to complete their computations.

Initially, every flow graph requires a graph object to invoke global operations on it. For example, these operations allow to wait for all tasks related to the graph to complete, reset the state of the operations, or cancel all operations. The code below creates such a graph object (all shown software elements for the flow graph are located in the namespace `tbb::flow`).

```
using namespace tbb::flow;
graph g;
```

The first node of the graph is the starting point for all group simulations of WMSim (Figure 5.7). The node's single responsibility is to broadcast an empty message (a `continue_msg`) to all group simulations. Broadcasting nodes are a special type of graph node that invokes all of its successor nodes by forwarding incoming messages.

```
broadcast_node<continue_msg> init(g);
```

Every tournament group is simulated by an execution of the function `playGroup`. TBB's approach to encapsualte functionality into graph nodes is by declaring function nodes. These are templated functions with defined input and output types, which are assigned to the graph. Their function body (the lambda function below) gets the message from the successor node as argument and must return the computational result. To connect the initial broadcast node with all nodes for the group simulations, we create an edge between them with `make_edge`. We demonstrate the described functionality with group A of the tournament below, the other groups are handled analogously.

```
function_node<continue_msg, Result> playGroupA(g, 1, [&](continue_msg m) {
   return playGroup("Group A", &groupA);
});
make_edge(init, playGroupA);
```

After the tournament groups were simulated, their results (i.e., the first and second winner teams) must be forwarded to the simulations of the first knockout stage. Note that the tournament uses a specific scheduling for these matches, e.g., in the first match the winner of group C plays against the second of group D. Essentially, this requires combining the results of two group nodes and forward it to a single node that represent the simulation of the first knockout stage. TBB supports this combination by providing join nodes; generic functions with two input ports, one for each incoming edge. These join nodes can be connected to one output node that can consume both messages.

```
join_node<tuple<Result, Result>> joinGroupC_GroupD(g);

function_node<tuple<Result, Result>, Result>
playBest16_1(g, 1, [&](tuple<Result, Result> &result) {
   return _simulator->playMatch(std::get<0>(result).winner, std::get<1>(result).second);
});

make_edge(playGroupC, input_port<0>(joinGroupC_GroupD));
make_edge(playGroupD, input_port<1>(joinGroupC_GroupD));
make_edge(joinGroupC_GroupD, playBest16_1);
```

Connecting the other match simulations of the knockout stage follows the same pattern. The library's syntax requires creating functional nodes for each match and separate join nodes to connect their results. The code below shows the last nodes for the semi finals and the final match.

```
make_edge(playSemiFinal_1, input_port<0>(join1));
make_edge(playSemiFinal_2, input_port<1>(join1));
make_edge(join1, playFinal);
```

So far, we only specified the flow graph but did not start its computations or obtained its result. This is conducted by the code below; `try_put` puts the first (empty) message into the starting node and `wait_for_all` blocks until all nodes completed their computations. The final result (i.e., the winner of the tournament) is finally written to the variable `result` by invoking `try_get` on the last graph node.

```
init.try_put(continue_msg());
g.wait_for_all();

Result result;
playFinal.try_get(result);
```

5.1.6 Discussion

We presented some details of a parallelization of WMSim. Certainly, the parallelization scenario could also be realized with other tasking models, such as those in OpenMP or C++. However, the flow graph framework favorably encodes the computational structure of the simulator. Rather than implicit and obscured data dependencies, the non-intuitive tournament scheduling is explicitly expressed within the source code. This improves understandability and alleviates debugging, which is particularly tedious in the case of parallelism. Additionally, the used framework lets us abstract parallelism to achieve a portable software design. This also

improves the performance of the simulator; TBB's internal dependency management facilitates overlapped simulations of different matches.

In conclusion, our example indicates the benefits of clear software design and the right tools for parallelization. Parceive helped us to identify parallelization scenarios and found intricate obstacles for valid restructuring. Most importantly, during the process we gradually improved our understanding of the simulator. This essentially led to the final parallelization scenario based on the domain-specific match scheduling. The implementation of this scenario required a moderate restructuring of the software design. Although such efforts are often tedious and error-prone, the resulting software gets more maintainable and expressive. Hence, approaches like our proposed one definitely pay off for relevant applications.

5.2 CPPCheck

In this section we demonstrate our proposed approach by inspecting CppCheck [83], a widely used static analysis tool. The goal of this case study is to understand CppCheck's underlying structure and run-time behavior to identify its potential parallelization scenarios. Parceive assisted us in gradually refining our understanding of the relevant components and their dependencies. The author of CppCheck confirms these findings and provides an illustration of the program's software architecture. We include this information into the specified architecture rules for the software architecture reconstruction step.

CppCheck accepts multiple source files as input and applies a set of checks to each file. Users typically use CppCheck continuously during development to obtain quick feedback about potential issues. Good overall performance is crucial for this iterative process relative to the acceptance of CppCheck. Here, parallelism is an obvious choice to reach performance goals; thus, we consider CppCheck to be a valid case study.

We executed CppCheck with Parceive to analyze all source code files in the supplied sample directory. During this execution, 18 files containing nine different issues (one "good" and one "bad" file per issue) were analyzed. We configured Parceive to consider all function calls and memory accesses, including static and shared libraries, such as the C++ standard library. With an optimized version of CppCheck (GCC 4.9, optimization flag `-Og`), the execution yielded 416 megabytes of trace data and took 32 seconds. On our system, this is a relative slowdown of 102.

After describing CppCheck, we demonstrate the essential steps that helped us identify parallelization scenarios for CppCheck, and highlight the key findings. For each step, we begin with a brief discussion of our motivation and then focus on the case study. Note that the steps are not meant to be general guidelines for parallelization or the use of Parceive. Rather, they aim to support our claim that tackling the additional complexity introduced by parallelism requires good comprehension of an application's software architecture.

5.2.1 Overview

CppCheck is an open-source static analysis tool for the C and C++ programming languages. It is written in C++ and comprises roughly 200,000 lines of code. The tool provides a variety of checks that are performed statically at the source code level, such as coding standard conformance, identification of undefined behavior, and security issue checks. The project is actively maintained and used to analyze numerous popular applications. A famous example is the Linux Kernel, where CppCheck has found several bugs.

At a coarse level, CppCheck's software architecture comprises a command-line interface (CLI) and a check library. The CLI reads source code files to be checked and initializes the check library, which performs

the static analyses on those files. By relying on the author's given architecture description and Parceive's visualizations, we identified the following major components of the check library: the library class, the preprocessor, the individual check classes, the symbol database, the settings manager, the value flow handling, and the token creation.

For each source code file, CppCheck uses the same analysis workflow consisting of six distinct phases grouped into *preparation* and an *analysis* (Figure 5.12). During the first phase, each input file is preprocessed and forwarded to the tokenizer component to construct an abstract syntax tree (AST). Then, CppCheck performs a value-flow analysis on the AST and applies all enabled checks on it. During this phase, the error reporting component collects detected errors and warnings, which are reported to the user. In the following, we describe each of these steps in more detail.

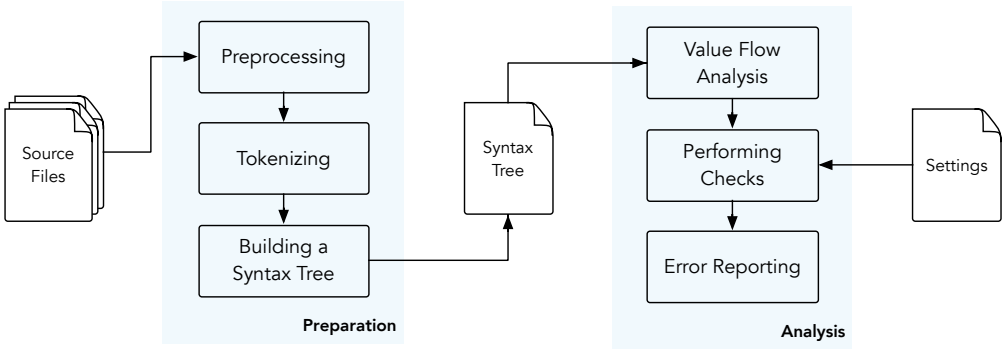


Figure 5.12: CppCheck's workflow for static analysis.

1. *Preprocessing.* During preprocessing, CppCheck removes unnecessary comments from source files, and expands macros to improve analyzability. The latter includes special features for extracting `ifdef` configurations and removing unused macros. The functionality is defined in the `Preprocessor` class.
2. *Tokenizing.* This phase is responsible for tokenizing the code by lexical analysis defined in the class `Tokenizer`. Single tokens are represented by the class `Token`, which are stored in the container `TokenList` for each source file.
3. *Building a Syntax Tree.* During this phase, CppCheck generates an AST with extracted tokens as nodes. Every statement in the source code is part of this syntax tree.
4. *Value Flow Analysis.* Before executing any checkers, CppCheck performs a context-sensitive value flow analysis. This analysis tracks the possible variable values in the source code and stores them in the corresponding tokens. After this process, every token has a list of possible values.
5. *Performing Checks.* CppCheck supports a wide variety of static checks that are defined in single classes inheriting from the `Check` class. The user selects a subset of all available checks by declaring them in a settings file. Each selected check is then performed on the current source file.
6. *Error Reporting.* Whenever a checker detects a problematic situation in the user code, CppCheck dumps an error message to the command-line. This functionality is defined in the `Check` sub-classes and the CLI.

5.2.2 Partitioning

First, we attempted to identify good candidates for task and data decomposition by inspecting the distribution of execution times. The Performance View revealed that the program execution can be separated into two

parts (Figure 5.13), i.e., the initial loading of the CppCheck library (~4%) and checking of the input files (~95%). The execution time for checking the input files varied for each of the 18 input files. However, the call sequences in this part are similar relative to execution time distribution. Generally, checking a file comprised some preprocessing (10% rel.), an initial check phase for normal tokens (~50% rel.), and a final check phase for simplified tokens (~40% rel.). This recurring pattern indicates that decomposition at the file level is a good candidate for parallelization in this case.

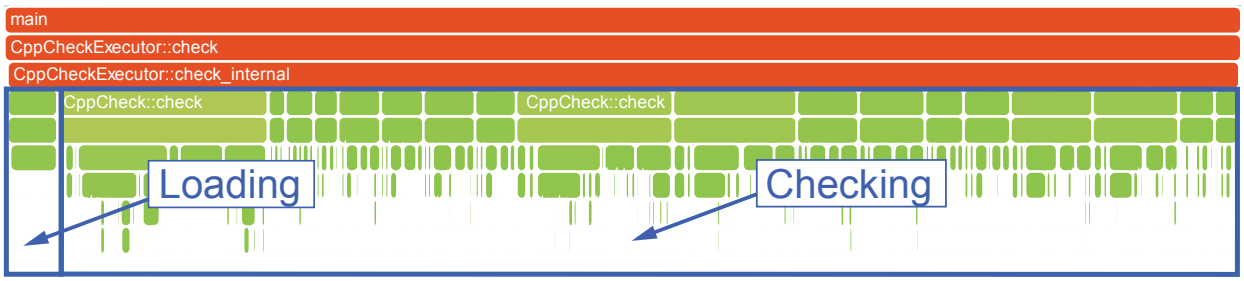


Figure 5.13: Performance View showing hierarchical nodes to represent function executions.

5.2.3 Validation

In the second step, we refined the task parallelism scenario from the first step. Here, we identified locations to spawn the tasks. The CCT View revealed that the starting point for checking the input files is the `CppCheck::check` method. This method is called for each input file within a specific loop; thus, parallelizing that loop was identified as an option. A straightforward approach for loop parallelization is to execute each iteration by a separate task. However, a subsequent dependency analysis exposed memory locations that would be accessed concurrently (Figure 5.14), which could cause severe issues during run-time (e.g., segmentation faults).

We found two origins for the dependencies by inspecting the memory-allocation instructions and the call stacks of the accesses (using the built-in interaction between the CCT View and the Source Code View). The first origin, which was caused by writes to the standard output, is the string implementation of the C++ standard library. The second origin, which was caused by accesses to a shared list in the `CppCheckExecutor` class, is the list implementation of the same library.

5.2.4 Redefinition

After identifying the data dependencies between the tasks, we had to determine the parallelization strategy. From an operating system perspective, parallelism can be achieved using threads (i.e., shared memory parallelism) or processes. For our scenario, multi-threading means the execution of single checks in separate threads. Note that this requires synchronization of all shared accesses to avoid race conditions during execution. Dividing the work into multiple processes is to execute the file checks in separate address spaces. This strategy avoids race conditions by design; however, it requires inter-process communication for CppCheck's logging facility (e.g., by using pipes). After inspecting all data dependencies in the previous step, we selected the multi-process scenario.

We evaluated the multi-process scenario using Parceive's Architecture View. A new issue caused by the duplicated memory regions for each child process was identified. In the present implementation, a single

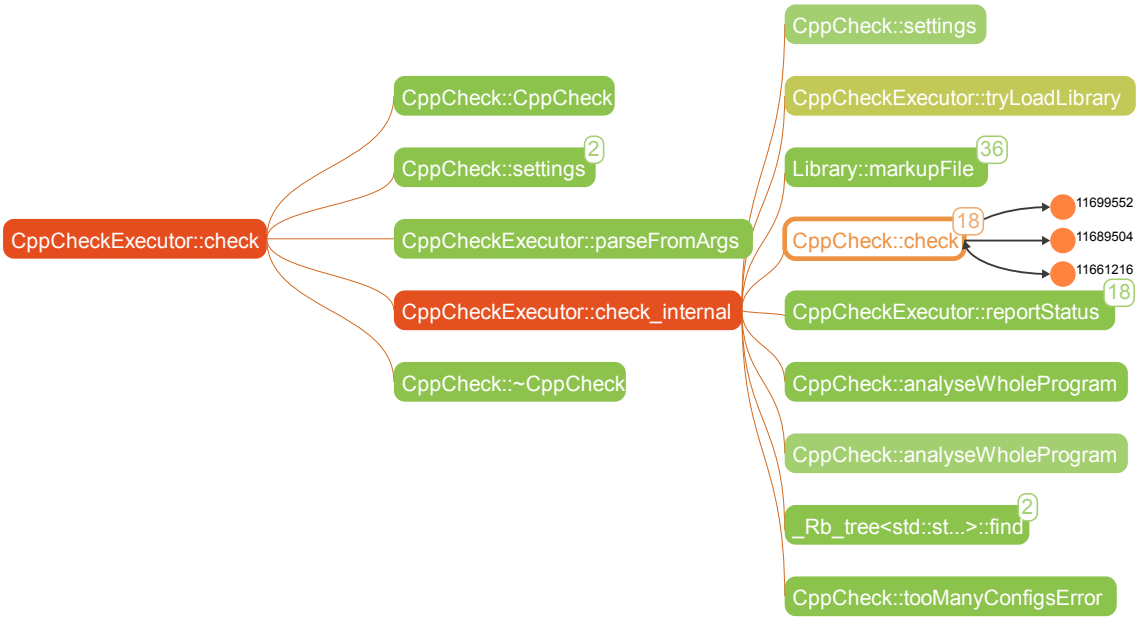


Figure 5.14: CCT View showing a calling context tree and accesses to shared memory locations.

object of the `CppCheck` class is responsible for initiating all analyses by calling the virtual `check` methods of the derived analysis classes. The scenario would duplicate this object and, as a result, remove any existing dependency on the parent process. This would inevitably change the semantics of the application. We thus had to check the `CppCheck` class for data dependencies on parts of the application that would be executed by different processes. The Architecture View supports this task by displaying the memory accesses between arbitrary artifacts of the software. Thus, the tool can highlight data dependencies between the class and any other structural component, including those executed in the parent process.

Recall that rules to extract meaningful artifacts were defined in the former steps. The author of CppCheck confirmed most of the rules and helped us refine them. CppCheck's overall software architecture is divided into a command-line interface and a library. The command-line interface initializes the library and file handling. The library checks the input files and includes the following main components: the library class, the preprocessor, the single check classes, the symbol database, the settings manager, value flow handling, and token creation.

```
checks         = class("Check.*")
token          = class("Token.*")
preprocessor   = infile(".*preprocessor.*")
symbol         = infile(".*symboldatabase.*")
settings       = infile(".*settings.*")
valueflow      = namespace("ValueFlow")

cli           := infile(".*/cli/.*")
library       := [class("CppCheck"),
                  class("Library"), checks,
                  token, symbol, settings
                  preprocessor, valueflow]
```

After extracting the corresponding artifacts from the debug information, we investigated the memory accesses of the `CppCheck` class. We identified four (out of 26) methods that contained data dependencies to

the artifact containing all check classes (Figure 5.15). Here, all accesses occured during file checking; thus, they are not an issue for this parallelization scenario. However, we also identified that all other artifacts in the library artifact are instantiated by the `CppCheck` object. This required analysis of these artifacts to determine possible dependencies on the command-line interface. We found two dependencies, i.e., the same dependencies as we found during validation. This confirmed the parallelization scenario.

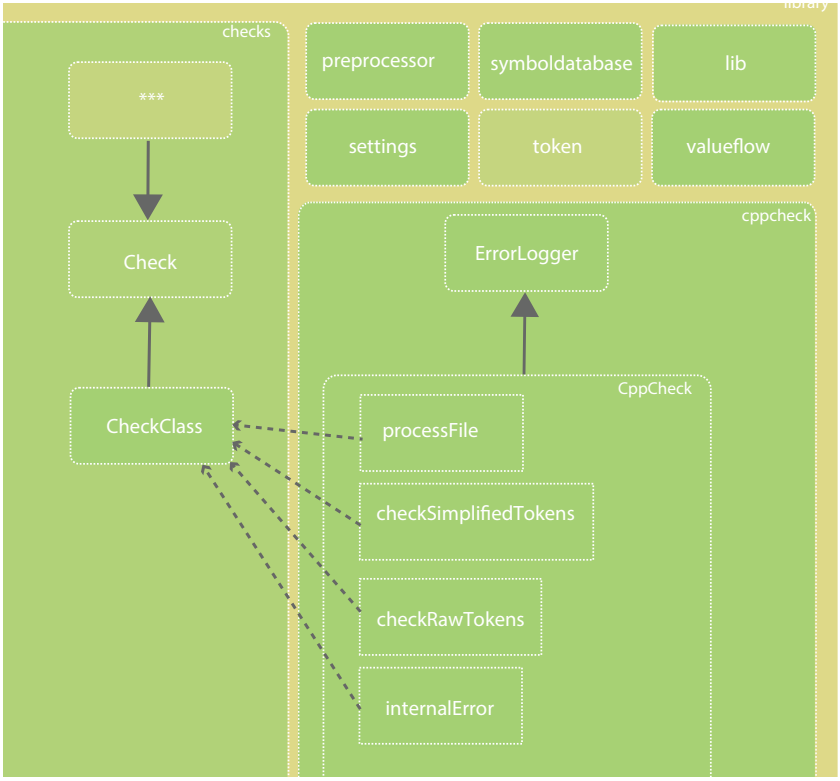


Figure 5.15: Parceive's Architecture View (excerpt) that shows the library artifact, inheritances relations (solid arrows), and memory accesses (dashed arrows) for an execution of `CppCheck`.

5.2.5 Discussion

The case study demonstrates how Parceive helps identify and evaluate parallelization scenarios for existing applications. The identified parallelization scenario is largely identical to what the author of CppCheck implemented in the Linux version of the tool. Since the resulting parallelization is relatively coarse-grained, it scales well and can utilize multiple processing units efficiently.

However, the truly interesting findings are not related to the parallelization results. They were gained while conducting the described steps. In this case, an important benefit of Parceive was a progressive refinement of our understanding of the software. This gradually helped us understand both the behavior and the structure of CppCheck independent of the relevance of the information for parallelism. One example is a flexible usage of dependency analysis at the architectural level and on call level. This revealed several relations, such as static or global variables, we would have missed by simply reading the source code. Another example is the coloring of artifacts in the Architecture View based on the recorded execution times. This feature identified parts of the application that were not executed during the analysis. One such component is the `ThreadExecutor` class that implements the multi-process scenario, which we identified.

Analyses that operate at the machine code level led to unexpected benefits. The displayed views gave analysis results originating from the executed binary, which may or may be not optimized. Using multiple runs with different optimization strategies and compiler flags, we could compare different compilation results. Some optimizations would influence the parallelization scenarios by removing entire call paths or memory accesses. For example, major differences were observed when the compiler used return value optimization or inlining. Another benefit of the binary analysis is the possibility of inspecting single template instantiations in C++ code. Source code analysis would not identify different variations of a template because instantiation occurs at compile time.

In addition to its use as an analysis tool, one additional aspect of Parceive that should not be underestimated is how it facilitates communication. During discussions with our colleagues, we noticed that the visualizations are very helpful to explain various aspects of the software. As a result, we can describe the important aspects abstractly separate from unimportant aspects of the software architecture to pinpoint bottlenecks in specific regions during run-time. In addition, the views and analyses allowed us to identify relationships between classes that are difficult to describe by simply looking at the source code.

The most notable limitation of our approach is the incompleteness of its dynamic analysis. The knowledge gained about the application's behavior must always be related to a specific execution. If some parts of the software were not executed, the analyses cannot identify any events in those unexecuted regions. Another restriction relative to the analysis of long-running executions is the slowdown and memory consumption incurred by Parceive. In some situations, this can make the analysis impossible because the input data is too extensive for visualization. Finally, comprehending the visualizations sometimes requires significant knowledge about the features of the target programming language and compiler. The visualizations show all nodes that exist in the machine code; thus, they may show implicit software entities that are not explicit to the target application's developer, which could cause misunderstanding of analysis results.

6

Conclusion and Outlook

"Instead of having people that only understand a small sliver of the vertical stack, we need to think about how to build teams that put together people who understand applications, languages, domain specific languages and related compiler technologies together with people who understand (hardware) architectures."

– John Hennessy

6.1 Conclusion

We presented a novel program analysis tool to discover parallelism in sequential programs. The tool differentiates from other related work by its focus on program comprehension and data dependency analysis at different granularity levels. The tool combines results from static and dynamic analyses with architectural information extracted from existing software systems.

Parallelization of industrial applications remains a demanding problem. Such applications typically comprise numerous components at several architecture levels. Proper comprehension of software, which is necessary for any serious re-engineering, is impossible without effective tool support. Parallelism discovery tools aim for high levels of automation to reduce the programmer's effort, e.g., the effort of manual code annotations. Their basic assumption is that substantial parallelism of computational problems is derivable from their sequential solutions. The effectiveness of those approaches highly depends on the quantity and regularity of exploitable concurrency within low-level code structures. Fostering user's comprehension of the computational problems is missing, e.g., by providing insights at higher abstraction levels. On the other hand, tools for software architecture reconstruction aim at program comprehension. However, their objectives are rarely targeting run-time qualities, such as parallelization. Although dependency analyses are essential techniques within SAR tools, we are - to the best of our knowledge - not aware of tools that consider data dependencies.

We argued in this thesis for less automation and put more focus on knowledge of programmers and architects. The objective of our tool-based approach is to support parallelism discovery at different levels,

particularly within the problem domain. We introduced a set of interactive visualizations to depict different viewpoints of user programs. Each view presents specific software elements that represent behavioral or structural aspects relevant for parallelization. Users of Parceive utilize these views to explore and evaluate arbitrary parallelization scenarios independent of parallel programming models. The views rely on the presented visualization framework that is key for user-oriented dependency analysis and information abstraction. The framework generates custom data queries to efficiently retrieve visualization objects and analysis results on demand. Novel interactions between structural and behavioral elements complement the exploration process.

To match the great diversity of architectural styles in real-world software and the even greater diversity on related user queries, we introduced a language to specify architecture rules. The language is based on binary relational algebra and regular expressions to steer an automated architecture recovery process. This process uses the explicit rules and some implicit algorithms to recover architectural representations from program binaries. Results are visualized by hierarchical box-and-arrow diagrams that enables users to effectively explore and analyze their programs. The user benefits from analyses and views that can be flexibly tailored towards various parallelization scenarios. The resulting perspectives provide key insights for restructuring and parallelization.

Two case studies demonstrated major steps to identify parallelization scenarios. Parceive enabled us to identify and decompose potential tasks, apply dependency analysis on the respective software regions, and evaluate the parallelization scenarios. The analyses and visualizations gradually refined our understanding of the applications, and helped us communicate about the software effectively with our colleagues.

6.2 Future Directions

Note that we regularly apply Parceive to open-source legacy applications and to proprietary software of our industry partner, which has led to various continuation paths for the work presented in this dissertation. In this section we outline four research directions that we would like to pursue in the future.

Performing more case studies. Although the performed case studies demonstrate great potential for parallelism discovery at varying design levels, industrial usage of Parceive requires further discoveries. Three important directions are:

- *Portability.* Parceive's backend component heavily relies on existing techniques for dynamic binary analysis. During evaluation, we noticed a moderate diversity in binary formats produced by different operating system, language, and compiler versions. The backend must handle this diversity by supporting the most relevant formats and platforms. We will perform more case studies targeting inhomogeneous application environments to improve the tool's portability.
- *Scalability.* Our tool changes performance characteristics of user programs in a non-linear and heterogeneous fashion. Based on the amount and types of events, the scalability of program runs differ largely. For example, programs with regular and data-intensive workloads may incur more slowdown than irregular programs with lots of control logic. To optimize Parceive, we require more case studies that exceed the quantities of performed analyses by few orders of magnitude.
- *Variability.* Real-world applications differ widely in their architectural and development styles, which largely affects program's analyzability. Parceive aims at supporting this variability by the presented explicit and implicit architecture rules and the considered types of dependencies. However, our evaluation has been solely based on selected case studies. Currently, the built-in

analyses support only monolithic software systems without distributed communication facilities. To extend Parceive's applicability and impact, we will perform analyses on programs with distributed software architectures, e.g., service oriented designs.

Providing more automation. In this thesis, we propose a flexible parallelization approach that requires substantial user interaction. Nevertheless, we recognize large potential towards the integration of more automated techniques. Two promising directions are:

- *Detecting design patterns.* The different levels of abstraction concerned by Parceive match closely to existing approaches for parallel pattern detection [90, 61]. Integrating such approaches into our tool's analyses phases would simplify the manual identification of parallelization scenarios. Furthermore, parallel patterns could be explicitly expressed by extended or new visualizations. For example, a representation of single stages of a pipeline pattern would be beneficial for the descriptiveness of the CCT View and could be combined with special dependency analyses.
- *Estimating speedup.* Our proposed approach considers performance estimation of parallelization scenarios as a manual task. The parallelism discovery process could be improved by relieving the user from this task. Existing approaches build performance models based on instruction counts [75] or profiling results [27, 43], and hardware architecture specifications. We intend similar techniques for Parceive with interesting extensions for higher-level patterns, such as parallel task graphs.

Optimizing for less slowdown. We will attempt to minimize the slowdown incurred by dynamic analysis to make the tool more practical when analyzing applications with long run times. Most savings in memory and computation time can be achieved by optimizing the data dependency analysis. Currently, Parceive uses shadow memory and region-based comparisons on accurate hash maps to detect accesses to common memory locations in one pass.

One approach to optimize both for speedup and memory consumption is to reduce the number of inspected data accesses. For example, the Parallel Advisor uses two consecutive passes for parallelism discovery [27]; the first pass constructs the structure and performance model, and the second pass adds dependency information. Applying this technique to Parceive could enable user-selected dependency analyses on certain software elements to reduce their slowdown and memory consumption.

An orthogonal approach is to optimize the data structures and algorithms used for dependency analysis. For example, existing tools compress strided data accesses within loops [67], parallelize the dependency analyzer [75], or track data accesses with signature-based data structures [103].

Supporting user queries. Parceive uses an accurate meta model to store program models. These models are currently accessed, refined, and abstracted by generated but statically determined queries. Useful extensions are user defined queries to obtain behavioral and architectural aspects of software systems. We will pursue elegant and efficient query languages. These languages use relations and operations with precise mathematical definitions to do various kinds of analyses. Typical notations are relational binary algebra [58] or n-ary relations similar to Datalog [14]. Potential use cases for Parceive include software architecture conformance checking (comparable to the reflexion model [93]), or the detection of parallelization issues, such as thread-safety violations.

Bibliography

[1] Our Pattern Language: A Design Pattern Language for Engineering (Parallel) Software. `https://patterns.eecs.berkeley.edu`. Accessed: 2018-09-30.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2006.

[3] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[4] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. ACM, 1967.

[5] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *ACM Sigplan Notices*, 1997.

[6] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D Patterson, W. Plishker, J. Shalf, S. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[7] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, University of California, Berkeley, 2006.

[8] E. Ayguadé, R. M. Badia, D. Jiménez, J. R. Herrero, J. Labarta, V. Subotic, and G. Utrera. Tareador: A Tool to Unveil Parallelization Strategies at Undergraduate Level. ACM, 2015.

[9] H. C. Baker and C. Hewitt. The Incremental Garbage Collection of Processes. ACM, 1977.

[10] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 1993.

[11] V. Basili and H. Mills. Understanding and Documenting Programs. *IEEE Transactions on Software Engineering*, 1982.

[12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2003.

[13] A. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, 1966.

[14] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, 2005.

[15] R. Blumofe and C. Leiserson. Space-Efficient Scheduling of Multithreaded Computations. *SIAM Journal on Computing*, 1998.

[16] F. Bodin and M. O'Boyle. A Compiler Strategy for Shared Virtual Memories. Springer, 1996.

[17] Mike Bostock. D3.js. https://d3js.org, 2018.

[18] P Bourque, R. Fairley, et al. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[19] R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International journal of man-machine studies*, 1983.

[20] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic Generation of Nested, fork-join Parallelism. *The Journal of Supercomputing*, 1989.

[21] F. Buschmann. Gardening Your Architecture, Part 2: Reengineering and Rewriting. *IEEE software*, 2011.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons, 1996.

[23] Frank Buschmann. Gardening your Architecture, Part 1: Refactoring. *IEEE software*, 2011.

[24] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An Integrated Framework for MPSoC Application Parallelization. ACM, 2008.

[25] A. Christl, R. Koschke, and M. Storey. Equipping the Reflexion Method with Automated Clustering. IEEE, 2005.

[26] M Conway. How do Committees Invent. *Datamation*, 1968.

[27] Intel Corporation. Parallel Studio XE. https://software.intel.com/en-us/parallel-studio-xe, 2018.

[28] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-memory Programming. *IEEE computational science and engineering*, 1998.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[30] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Elsevier, 2002.

[31] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[32] S. Ducasse and S. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 2009.

[33] P. Feautrier. Automatic Parallelization in the Polytope Model. Springer, 1996.

[34] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine Doctoral Dissertation, 2000.

[35] F. Fittkau, A. Krause, and W. Hasselbring. Exploring Software Cities in Virtual Reality. IEEE, 2015.

[36] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. 2013.

[37] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley Publishing Company Reading, 1995.

[38] Node.js Foundation. Node.js. https://nodejs.org/en/, 2018.

[39] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[40] H. C. Gall. *Archview-Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology Vienna, 2005.

[41] K. Gallagher, A. Hatch, and M. Munro. Software Architecture Visualization: An Evaluation Framework and its Application. *IEEE Transactions on Software Engineering*, 2008.

[42] E Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[43] S. Garcia, D. Jeon, C. Louie, and M. Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. ACM, 2011.

[44] D. Garlan and M. Shaw. An Introduction to Software Architecture. World Scientific, 1993.

[45] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the Evolution of Class Hierarchies. IEEE, 2005.

[46] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming*, 2006.

[47] GitHub. Electron. https://electronjs.org, 2018.

[48] Silexica GmbH. SLX. https://www.silexica.com, 2018.

[49] Google. Angular.js. https://angularjs.org, 2018.

[50] Google. Chromium. https://www.chromium.org/Home, 2018.

[51] J. Grundy and J. Hosking. High-Level Static and Dynamic Visualisation of Software Architectures. IEEE, 2000.

[52] J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 1988.

[53] A. Hatch. *Software Architecture Visualisation*. PhD thesis, Durham University, 2004.

[54] Matthew Hause. The SysML Modelling Language. 2006.

[55] J. Hennessy, D. Patterson, P. Prinz, and T. Crawford. Computer Architecture: A Quantitative Approach. 2017.

[56] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.

[57] R. Hilliard. IEEE-std-1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE, http://standards. ieee. org*, 2000.

[58] R. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. IEEE, 1998.

[59] R. Holt. Software Architecture as a Shared Mental Model. *Proceedings of the ASERC Workhop on Software Architecture, University of Alberta*, 2002.

[60] R. Holt and J. Pak. GASE: Visualizing Software Evolution-in-the-Large. Citeseer, 1996.

[61] Z. Huda, R. Atre, A. Jannesari, and F. Wolf. Automatic Parallel Pattern Detection in the Algorithm Structure Design Space. IEEE, 2016.

[62] A. Jannesari. Detection of High-Level Synchronization Anomalies In Parallel Programs. *International Journal of Parallel Programming*, 2015.

[63] A. Jannesari and F. Wolf. Automatic Generation of Unit Tests for Correlated Variables in Parallel Programs. *International Journal of Parallel Programming*, 2016.

[64] Ali Jannesari. Advances in Engineering Software for Multicore Systems. 2018.

[65] R. Kazman and J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 1999.

[66] K. Keutzer and T. Mattson. A Design Pattern Language for Engineering (Parallel) Software. *Intel Technology Journal*, 2010.

[67] M. Kim, H. Kim, and C.-K. Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. IEEE, 2010.

[68] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. Static Evaluation of Software Architectures. IEEE, 2006.

[69] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[70] J. Kruskal and J. Landwehr. Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 1983.

[71] M. Lanza and S. Ducasse. Polymetric Views-a Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 2003.

[72] M. Lanza and S. Ducasse. Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 2003.

[73] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. IEEE Computer Society, 2004.

[74] S. Letovsky. Cognitive Processes in Program Comprehension. *Journal of Systems and software*, 1987.

[75] Z. Li, R. Atre, Z. Huda, A. Jannesari, and F. Wolf. Unveiling Parallelization Opportunities in Sequential Programs. *Journal of Systems and Software*, 2016.

[76] S.-W. Liao, A. Diwan, Bosch R., A. Ghuloum, and M. Lam. *SUIF Explorer: An Interactive and Interprocedural Parallelizer*. ACM, 1999.

[77] Google LLC. Chrome V8. https://github.com/v8/v8, 2008.

[78] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 2005.

[79] M. Lungu and M. Lanza. Exploring Inter-Module Relationships in Evolving Software Systems. IEEE, 2007.

[80] M. Lungu, M. Lanza, and O. Nierstrasz. Evolutionary and Collaborative Software Architecture Recovery with Softwarenaut. *Science of Computer Programming*, 2014.

[81] David Majda. PEG.js. https://pegjs.org, 2018.

[82] J. Maletic, A. Marcus, and M. Collard. A Task Oriented View of Software Visualization. IEEE, 2002.

[83] Daniel Marjamäki. CppCheck. http://cppcheck.sourceforge.net, 2018.

[84] R. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[85] B. Massingill, T. Mattson, and B. Sanders. Reengineering for Parallelism: An Entry Point into PLPP for Legacy Applications. *Concurrency and Computation: Practice and Experience*, 2007.

[86] ANL Mathematics and Computer Science. MPI. https://www.mcs.anl.gov/research/projects/mpi/.

[87] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.

[88] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[89] Leonel Merino and Oscar Nierstrasz. *The Medium of Visualization for Software Comprehension*. PhD thesis, Universität Bern, 2018.

[90] K. Molitorisz. Pattern-Based Refactoring Process of Sequential Source Code. IEEE, 2013.

[91] Australia Monash University, Melbourne. cola.js. https://ialab.it.monash.edu/webcola/, 2018.

[92] H. Müller, S. Tilley, and K. Wong. Understanding Software Systems using Reverse Engineering Technology Perspectives from the Rigi Project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*. IBM Press, 1993.

[93] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. *ACM SIGSOFT Software Engineering Notes*, 1995.

[94] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-Parallel Finite-State Machines. In *ACM SIGARCH Computer Architecture News*. ACM, 2014.

[95] R. Netzer and B. Miller. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1992.

[96] A. Nicolaescu and H. Lichter. Behavior-based Architecture Reconstruction and Conformance Checking. IEEE, 2016.

[97] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A Lexical Pattern Matcher for Architecture Recovery. IEEE, 2002.

[98] Terry Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley Professional, 2000.

[99] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 1994.

[100] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[101] T. Richner and S. Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. IEEE, 1999.

[102] C. Riva. *View-Based Software Architecture Reconstruction*. Citeseer, 2004.

[103] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. IEEE, 2007.

[104] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? 2012.

[105] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013.

[106] Z. Sharafi. A Systematic Analysis of Software Architecture Visualization Techniques. IEEE, 2011.

[107] M.. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice-Hall*, 1996.

[108] B. Shneiderman and R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences*, 1979.

[109] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *ACM SIGPLAN Notices*, 2004.

[110] M Storey, C. Best, and J. Michand. Shrimp Views: An Interactive Environment for Exploring Java Programs. IEEE, 2001.

[111] M. Storey, D. Fracchia, and H. Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Systems and Software*, 1999.

[112] K. Sullivan, W. Griswold, Y. Cai, and B. Hallen. The Structure and Value of Modularity in Software Design. ACM, 2001.

[113] Alfred Tarski. On the Calculus of Relations. *The Journal of Symbolic Logic*, 1941.

[114] S. Tichelaar, S. Ducasse, and S. Demeyer. Famix and Xmi. IEEE, 2000.

[115] S. Tilley, S. Paul, and D. Smith. Towards a Framework for Program Understanding. IEEE, 1996.

[116] T. Tilley, R. Cole, P. Becker, and P. Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. Springer, 2005.

[117] Q. Tu and M. Godfrey. The Build-Time Software Architecture View. IEEE, 2001.

[118] OMG UML. Unified Modeling Language. *Object Management Group*, 2001.

[119] A. Van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. ACM, 2012.

[120] A. Vermeulen, G. Beged-Dov, and P. Thompson. The Pipeline Design Pattern. Citeseer, 1995.

[121] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, d. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-Level Models. ACM, 1998.

[122] A. Wilhelm, F. Čakarić, M. Gerndt, and T. Schüele. Tool-Based Interactive Software Parallelization: A Case Study. ACM, 2018.

[123] A. Wilhelm, V. Savu, E. Amadasun, M. Gerndt, and T. Schüele. A Visualization Framework for Parallelization. IEEE, 2016.

[124] A. Wilhelm, B. Sharma, R. Malakar, T. Schüle, and M. Gerndt. Parceive: Interactive Parallelization based on Dynamic Analysis. IEEE, 2015.

[125] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 2009.

[126] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM Sigplan Notices*, 1994.

[127] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: A System for Discovering Architectures from Running Systems. IEEE, 2004.

[128] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. IEEE, 2009.

[129] B. Zhao, Z. Li, A. Jannesari, F. Wolf, and W. Wu. Dependence-based Code Transformation for Coarse-Grained Parallelism. ACM, 2015.