Multi-Granular Conflict and Dependency Analysis in Software Engineering based on Graph Transformation

Leen Lambers
Hasso-Plattner-Institut Potsdam, Germany
leen.lambers@hpi.de

Daniel Strüber
Universität Koblenz-Landau, Germany
strueber@uni-koblenz.de

Gabriele Taentzer, Kristopher Born, Jevgenij Huebert
Universität Marburg, Germany
{taentzer,born,huebert}@mathematik.uni-marburg.de

ABSTRACT

Conflict and dependency analysis (CDA) of graph transformation has been shown to be a versatile foundation for understanding interactions in many software engineering domains, including software analysis and design, model-driven engineering, and testing. In this paper, we propose a novel static CDA technique that is multi-granular in the sense that it can detect all conflicts and dependencies on multiple granularity levels. Specifically, we provide an efficient algorithm suite for computing *binary*, *coarse-grained*, and *fine-grained* conflicts and dependencies: Binary granularity indicates the presence or absence of conflicts and dependencies, coarse granularity focuses on root causes for conflicts and dependencies, and fine granularity shows each conflict and dependency in full detail. Doing so, we can address specific performance and usability requirements that we identified in a literature survey of CDA usage scenarios. In an experimental evaluation, our algorithm suite computes conflicts and dependencies rapidly. Finally, we present a user study, in which the participants found our coarse-grained results more understandable than the fine-grained ones reported in a state-of-the-art tool. Our overall contribution is twofold: (i) we significantly speed up the computation of fine-grained and binary CDA results and, (ii) complement them with coarse-grained ones, which offer usability benefits for numerous use cases.

CCS CONCEPTS

•**Software and its engineering** → *Automated static analysis;*

1 INTRODUCTION

Conflicts and dependencies are fundamental phenomena in software engineering. For example, when a software system is developed collaboratively [61], a change operation can facilitate or prohibit other change operations. In concurrent programming, conflicts may arise from data races [28, 56] when a thread writes to a memory location accessed by another thread. From unrecognized conflicts and dependencies, severe consequences may arise, ranging from productivity obstacles to fatal safety hazards. Therefore, there is a need for techniques to detect conflicts and dependencies automatically.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '18, Gothenburg, Sweden
© 2018 ACM. 978-1-4503-5638-1/18/05...$15.00
DOI: 10.1145/3180155.3180258

Graph transformation [17, 18, 53] has been shown to be a versatile foundation for supporting conflict and dependency detection in software engineering, based on the following three principles: First, graphs are used for representing structures of interest, such as states of computation [1, 7] or versions of the system structure [8]. Second, certain changes, such as state or structure modifications, are described using graph transformation rules. Third, the provided transformation specification is fed to the static *conflict and dependency analysis* (CDA) of graph transformations [27, 48]: Given a set of transformation rules, all conflicts and dependencies arising from a given pair of rules are identified. A conflict arises, for example, if the first rule application deletes an element required by the second rule application. A key benefit of graph transformation is its mature formal foundation, which supports CDA techniques that are correct by design: all conflicts and dependencies can be detected.

Based on these principles, the CDA of graph transformations has enabled a large number of *use-cases in software engineering*, including analysis and design, model-driven engineering, and testing. For example, graph transformations can be used to model the execution behavior of Java programs in terms of preconditions and effects on the object structure; identified conflicts and dependencies are then used as oracles for test generation [1, 54]. In software product line engineering, feature interactions can be detected by specifying features as graph transformations and identifying conflicts and dependencies with CDA [33]. In model-based refactoring [46], graph transformations and CDA are used to find a suitable order of refactoring steps. One contribution of this paper is a literature survey that overviews 25 papers describing such use-cases.

Although generally helpful for the task at hand, however, several authors report that the used CDA technique showed severe limitations. In our survey, we identify three key requirements for an improved CDA technique for software engineering: it shall be (i) *domain-independent* to be applicable to a large variety of software engineering domains, (ii) *usable* in the sense that it should display a reasonable amount of information to support understandability, and (iii) *efficient* when applied to software projects of realistic size.

To address these requirements, we present a novel static CDA technique for software engineering based on graph transformation. The technique is based on the notion of *granularity* of conflicts and dependencies introduced in [10]: Often, the user merely requires to know if a given rule pair can induce conflicts or dependencies at all, while details are irrelevant (*binary granularity*). At the next level, the user wants to pinpoint certain elements that present the root causes of conflicts or dependencies (*coarse granularity*). At the

final level, a complete description of each potential conflict and dependency is required (*fine granularity*). To enable an efficient computation, we present an algorithm suite which can compute binary, coarse-grained, and fine-grained results rapidly. The computation of fine-grained ones harnesses coarse-grained ones. Moreover, we conducted a user study, in which the participants found coarse-grained analysis results more understandable and easier to work with than fine-grained ones reported by a state-of-the-art tool.

In summary, this paper presents a **multi-granular CDA technique based on graph transformation** achieving the same level of (i) domain-independence as the state of the art, while providing major (ii) understandability and (iii) performance improvements. Specifically, we make the following contributions:

- A **literature survey** of existing CDA use-cases (Sec. 3), focusing on granularity requirements.
- A **formalization** of different granularity levels of conflict and dependencies (Sec. 4), for ensuring the well-foundedness of our technique.
- An **algorithm suite** supporting the computation of CDA results at multiple granularity levels (Sec. 5).
- An **implementation evaluation**, in which we study the performance of our algorithm suite (Sec. 6).
- A **user study** to determine the usefulness of coarse-grained conflict results in comparison to fine-grained ones (Sec. 7).

With these contributions, we aim to improve on the state-of-the-art CDA technique, critical pair analysis [42, 59] (CPA). CPA does not distinguish between granularity levels: Since its goal is to provide all conflicts and dependencies, its output—a list of *critical pairs* depicting each conflict situation in a minimal context—always exhibits fine granularity. From the lack of support for different granularity levels, two main drawbacks arise. First, computing all critical pairs can be computationally vastly expensive. Second, comprehending the critical pairs of a set of rules can be a daunting task, since the list of critical pairs generally reflects numerous options to combine the involved root causes.

Our work is the first to provide a general CDA technique for graph transformations supporting multiple granularity levels. Earlier usage scenarios either used CPA, or task-specific CDA techniques relying on the structure of the involved rules [31, 35]. Our earlier work [10, 40] serves as a formal foundation for the current one. We now amend the existing declarative definitions with constructive characterizations that support efficient computations. To the best of our knowledge, we provide the first, albeit preliminary, empirical evidence regarding the usefulness of CDA techniques.

2 RUNNING EXAMPLE

Our running example deals with requirement elicitation for a web shop, a service-oriented software system that enables a retailer to sell goods on a website. Customers can perform orders and inspect the information on goods and orders. The retailer uses the available information to manage its business processes. We focus on two business processes, one for *order management* and one for *data mining*, which we do not want to interfere with each other. This means that activities of one process should not render activities of the other process impossible; hence, conflicts between activities of different processes should not occur. To be able to analyze conflicts, we specify activity requirements with graph transformation. A transformation rule specifies pre- and post-conditions of activities.

In our example, we consider two activities of the selected business processes to be specified by rules: Rule *returnUnpaidGood* on the left of Fig. 1 returns a *Good* into the *Stock* by deleting the corresponding *OrderItem* and *BillItem*. In addition, the returned good is removed from the customer. Rule *offerGift* specifies a pattern for data mining which looks for an *Order* with at least two *OrderItems*. The *Customer* ordering and owning these items is marked for a *GiftOffer*. Note that a customer may own a good without having it ordered and paid since it may be a gift. Both rules are depicted in an integrated form where annotations specify which graph elements are deleted, preserved, and created. While the preserved and deleted elements form the left-hand side (LHS) of a rule, the preserved and created elements form its right-hand side (RHS).

Conflict and granularity considerations. To run the selected business processes for *order management* and for *data mining* concurrently, they shall not interfere with each other. This is the case if their activities do not interfere pairwise (neglecting any potential control flow on activities). We focus our investigations on activity *returnUnpaidGood* being part of the order management process and *offerGift* being some data mining activity. To investigate interferences between these two activities, we analyze two rules specifying them. If an application of the first rule renders an application of the second rule impossible or, if the second rule is still applicable, but not at the original match anymore, a conflict occurs. To reduce the amount of conflicts, we need to identify all their potential sources.

If the developer is just interested in knowing whether a given pair of rules can induce a conflict, this information can be easily given in a table such as Table 1 where a '+' marks that a conflict for the given rule pair arises while '-' marks that there is no conflict.

Rule 1 / Rule 2	returnUnpaidGood	offerGift
returnUnpaidGood	+	+
offerGift	-	-

Table 1: Binary information about conflicts

To get a *coarse understanding* of conflicts, the developer may be interested in knowing which rule elements can cause conflicts. Since rules *returnUnpaidGood* and *offerGift* specify activities of two processes that shall not interfere, we are especially interested in understanding all conflicts related to this rule pair. Two reasons for conflicts are shown in the middle of Figure 1. Conflict-causing elements are included in minimal graphs that describe the needed overlap of participating rules (depicted on the left and on the right) to cause an actual conflict on their applications. The upper graph specifies the deletion of an order item needed to offer a gift. The lower one shows the deletion of an *owns*-edge needed to offer a gift. Each of these graphs with their embeddings into rules are called *minimal conflict reasons*. Overlapping the rules along such a conflict reason yields two conflicting rule applications called *critical pair*.

For a *fine-grained representation* of all conflicts, we also have to consider all possible combinations of root causes. This means that all possible compositions of minimal conflict reasons describe further conflict situations which we just call *conflict reasons*. One example of such a conflict reason is shown in Figure 2, where the order item as well as the targeting *owns*-edge for one and the same

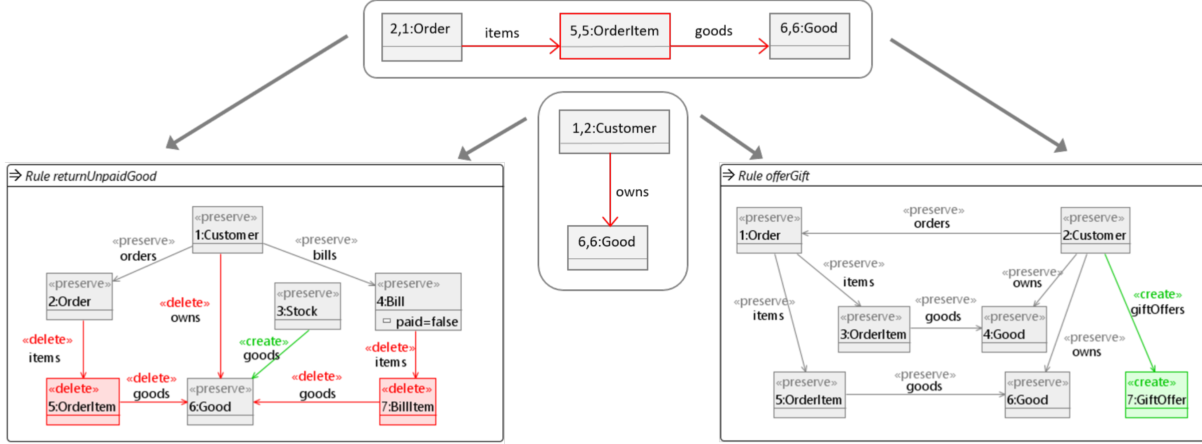

Figure 1: Rules *returnUnpaidGood* and *offerGift* and two minimal conflict reasons – Coarse-grained information

good are deleted, both needed to offer a gift. This graph induces one of altogether 6 conflict reasons caused by rule *returnUnpaidGood* on rule *offerGift*. The conflict reasons not shown are analogous to the three ones depicted in Figures 1 and 2. The remaining three ones overlap *5:OrderItem* and *6:Good* of rule *returnUnpaidGood* with *3:OrderItem* and *4:Good* of rule *offerGift*.

Figure 2: One of 6 conflict reasons of rule *returnUnpaidGood* on rule *offerGift* – Excerpt of fine-grained information

The state-of-the-art CDA which computes *essential critical pairs* [42], however, yields more results for this rule pair, namely 10 pairs of conflicting transformations. While 6 of them directly correspond to the ones discussed above, there are 4 further results which do not show basically new conflict reasons but just nuances of the considered ones. Hence, the existing CDA provides an even more fine-grained information about conflicts. Table 2 compares the numbers of coarse and fine-grained results for our example, distinguishing our intended fine-grained analysis (indicated by 'New Fine') from the existing CDA (indicated by 'Ex. Fine') . While the entry for rule pair (*returnUnpaidGood*, *offerGift*) shows a moderate increase of numbers, this is already more striking for rule pair (*returnUnpaidGood*, *returnUnpaidGood*) demonstrating a bigger difference between numbers of coarse and existing fine-grained conflict information (3 versus 19). In general, a lot of different critical pairs can exist and it may be tedious to go through all of them. In Section 6, we found an example rule pair with 1588 ess. critical pairs vs. 24 min. conflict reasons.

Rule 1 / Rule 2	returnUnpaidGood	offerGift
returnUnpaidGood	Coarse: 3 New Fine: 7 Ex. Fine: 19	Coarse: 4 New Fine: 6 Ex. Fine: 10

Table 2: Number of conflict reasons and ess. critical pairs, resp., in coarse and fine-grained representations

Conclusion. Instead of overwhelming the user with a large number of conflicts as the state-of-the-art fine-grained CDA does, a multi-granular analysis supports a continuously deeper understanding of conflicts where needed. In this example, rule pair (*returnUnpaidGood*, *offerGift*) is of special interest since it may be conflicting and specifies activities of two processes that should run independently of each other. From the analysis, we can deduce that data mining on unpaid goods may lead to inconsistencies. To avoid such conflicts, the activities may be adapted by, e.g., offering a gift only for goods that have been already paid.

3 LITERATURE SURVEY

We conducted a literature survey to explore the variety of software engineering domains in which critical pair analysis (CPA) has been applied, the designated state-of-the-art technique for conflict and dependency analysis (CDA) based on graph transformation. First, we present collected statements to performance and usability of the CPA. Then we elicit requirements regarding the actual granularity level needed when performing CDA in specific use-cases.

To identify use-cases in the literature, we applied the search clause `"critical pair*" AND ("graph transformation" OR "model transformation")` to the five major CS online libraries of ACM, IEE, Elsevier, Wiley, and Springer, restricting the search to mentions in title and abstract. This search yielded an initial body of 37 papers, to which we added 11 based on our knowledge about CPA uses. We discarded those that focused on theoretical results, and those represented by other papers in the same line of work on the same overall use-case. We grouped the remaining 25 papers in four main software engineering application domains (see Table 3).

Performance and usability statements. In several papers such as [13, 23, 44, 62], the authors recognized severe *performance problems* when using the state-of-the-art implementation of the CPA in AGG [59]. They conclude that CPA does not scale for industrial use. Often a too large number of critical pairs is computed which makes the *manual inspection* of the CPA results *nearly impossible*. As one solution to increase the performance and to drop the number of results, authors tried to constrain the allowed graphs. Another way out of this dilemma was to replace the static CDA by a runtime check. However, these actions either change the kind of graphs considered or switch from static to dynamic analysis. From this review of *performance and usability* statements, we conclude that

Granularity / SE domain	Analysis and design of software systems	MDE techniques	Testing	Optimization of rule-based computations
Binary				Graph parsing [12], Activity diagram validation [20], Edit operation recognition [35], Nondeterminism detection [23, 29]
Coarse-grained	Consistency validation of use-cases [26], service-based systems [38, 41], context-aware and adaptive systems [13, 15, 44], activity diagrams [20]; Feature interaction detection [2, 33, 45, 62]	Model versioning [39, 60], Refactoring recommendation [46, 50], Detecting & resolving model inconsistencies [47]	Test case generation [31] & validation [54]	
Fine-grained		Verification of model transformations [6, 30]		

Table 3: Granularity requirements and software engineering domains of the CPA in literature survey

a highly performant static analysis is needed which produces a concise set of results that is easy to inspect manually. The essential CPA [42], also available in AGG, was introduced as a first solution to these requirements yielding a considerably smaller set of results in a smaller amount of time. We observed, however, that even the essential CPA often returns *too fine-grained results* representing an obstacle for performance as well as for usability when doing CDA. Therefore, we analyze now the requirements w.r.t. the actual level of granularity needed when performing CDA.

SE domains and granularity requirements. We describe our findings w.r.t. granularity requirements of the CPA use cases along their application domains in SE as illustrated in Table 3.

In *software system design and analysis*, the conformance of behavior models such as activity models and live sequence charts, with the rule-based specification of activities (methods, operations, or services) is investigated. The CPA is mostly used to find and understand conflicts and dependencies in the data flow and to reason about their plausibility w.r.t. the considered system. In this context, a *coarse-grained CDA* seems to be sufficient to start with.

Model-driven engineering (MDE) techniques are often specified on the basis of model transformations. The CPA can be used to detect and to reason about the plausibility of conflicts and dependencies between specified transformations. In model version management, the CPA is moreover used to resolve conflicts between model changes. The *coarse-grained analysis* seems to be sufficient to find conflicts and dependencies between transformation specifications here. Confluence proofs, however, have to be performed based on a *fine-grained analysis* since these proofs are based on completeness of the CPA results which is only given in the fine-grained case.

In *testing*, the CPA is used for reasoning about and generating interesting test cases. A *coarse-grained analysis* of rule interdependencies seems adequate to understand the specified activities.

In *optimization of rule-based computations*, the CPA has been used to find out which rule pairs are conflicting or dependent at all and to exploit this information for improving the computation. The non-existence of conflicts or dependencies may allow computations without backtracking. Hence, *binary information* about the existence of conflicts or dependencies is usually sufficient to either avoid or deliberately postpone backtracking. For further optimization, the elimination of existing conflicts or dependencies may be a choice. Further information, i.e., a *coarse-grained analysis*, is then needed to understand their causes and to modify rules accordingly.

Threats to Validity Regarding *construct validity*, we omit investigating expressiveness of the analyzed graph transformations w.r.t. advanced transformation features such as negative application conditions, which is orthogonal to the required granularity level. The extension of our technique to these features is ongoing work. While not being trivial, we are confident that this is possible, since the underlying theory is given in a category-theoretical setting.

Conclusion. We deduce the following granularity requirements for CDA: (1) *Binary granularity* refers to the situation where the relevant information is whether a conflict/dependency between two graph transformations rules exists or not. Such information is sufficient, e.g., to trim a solution space of possible alternatives. (2) *Coarse granularity* refers to the situation where users need to inspect individual conflicts and dependencies, but do not need to know the precise details of each possible conflict or dependency situation. (3) *Fine granularity* is needed to inspect each conflict and dependency situation in-depth. This is necessary, for example, to reason about confluence of a state transition relation. Our observations (in Table 3) demonstrate that for most considered applications of conflict and dependency analyses, a binary or coarse-grained analysis would have been sufficient or would have represented a good starting point for analysis that can – only if necessary – still be refined to a more fine-grained one. These granularity requirements support the *need for a multi-granular approach* to CDA.

4 CONFLICT AND DEPENDENCY CONCEPTS

We revisit conflict and dependency concepts for graph transformation with varying granularity level in Sec. 4.1. As a new contribution, we instantiate the binary, coarse and fine granularity level (as identified in Sec. 3) of our multi-granular CDA technique with adequate formal conflict and dependency concepts in Sec. 4.2.

4.1 Background

We recall main concepts from graph transformation with the conflict notion underlying our work [17, 42]. Then, conflict and dependency concepts with varying granularity level are recalled [10, 40].

Graph transformation. Representing complex structures as graphs, *graph transformation* is one of the main paradigms to describe their rule-based modification. A *rule* mainly consists of two graphs: L is the left-hand side (LHS) of the rule representing a pattern that has to be found to apply the rule. After the rule application, a pattern equal to R, the right-hand side (RHS), has been created. The intersection $K = L \cap R$ is the part that is not changed, the part that is to be deleted is defined by $L \setminus K$, while $R \setminus K$ defines the part to be created. To make the deletion part of a rule a graph, we add all

boundary nodes $B \subseteq K$, hence obtain *deletion graph* $C = L \setminus (K \setminus B)$. If C is empty, the rule is called *non-deleting*.

A *graph transformation* $G \overset{r,m}{\Longrightarrow} H$ between two graphs G and H is defined by first finding a *match* m, that is a mapping of the LHS L of rule r into G such that m is injective and fulfills the *dangling condition* [17]: all adjacent graph edges of a graph node to be deleted must be deleted as well. Second, we construct H in two passes: (1) build $D := G \setminus m(L \setminus K))$, i.e., erase all graph elements that are to be deleted; (2) construct $H := D \cup m'(R \setminus K)$ such that a new copy of all graph elements that are to be created is added.

Example [Graph rule and transformation] Figure 3 shows a graph to which rules *returnUnpaidGood* and *offerGift* in Figure 1 are applicable. Considering, e.g., rule *returnUnpaidGood* the red and the grey parts form the LHS and the RHS consists of the green and the grey parts. Hence, K is represented by the grey part. Boundary nodes are *1:Customer*, *2:Order*, *4:Bill*, and *6:Good*. Deletion graph C consists of the red rule part together with all boundary nodes.

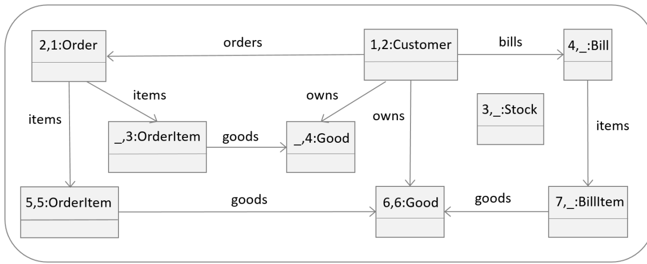

Figure 3: Example graph to which rules *returnUnpaidGood* and *offerGift* are applicable, with indicated matches

The rule's matches are indicated by numbers. For example, *1:Customer* of rule *returnUnpaidGood* and *2:Customer* of rule *offerGift* are both mapped to *1,2:Customer*. An '_' indicates that this node is not in the corresponding match. Note that rule *returnUnpaidGood* can be applied in two different ways to this graph. Since rule *offerGift* is non-deleting, the dangling condition is always fulfilled. Rule *returnUnpaidGood*, however, deletes two nodes: *5:OrderItem* and *7:BillItem*. Their images in the graph are not allowed to have dangling edges, i.e., edges without origins in the LHS. This is the case here, hence both mappings fulfill the dangling condition.

The effect of applying rule *returnUnpaidGood* at the given match is the deletion of edge *owns* from *1,2:Customer* to *6,6:Good* as well as of nodes *5,5:OrderItem* and *7,_:BillItem* with adjacent edges. In addition, a new edge from *3,_:Stock* to *6,6:Good* is added.

Conflict. Given a graph G there are, in general, several rules applicable at different matches. A pair of transformations ($G \overset{r_1,m_1}{\Longrightarrow} H_1$, $G \overset{r_2,m_2}{\Longrightarrow} H_2$) is *in conflict* if the first rule application deletes graph elements used by the second one, i.e., if $m_1(C_1 \setminus B_1) \cap m_2(L_2)$ is not empty. Since matches are injective, we can build a *conflict part* $m_1^{-1}(m_1(C_1 \setminus B_1) \cap m_2(L_2)) \subseteq C_1$ that can be completed by adding incident boundary nodes to a *conflict graph* S_1, being a subgraph of the deletion graph C_1 of rule r_1. Moreover, we have a mapping $e_2 = m_2^{-1} \circ m_1$ of S_1 into L_2. Together with its embedding into C_1, a *span* $s_1 = (C_1 \overset{\supseteq}{\leftarrow} S_1 \overset{e_2}{\rightarrow} L_2)$ for rule pair (r_1, r_2) can be defined which distills the cause of a conflict and therefore, is called *conflict reason* for $G \overset{r_1,m_1}{\Longrightarrow} H_1$ and $G \overset{r_2,m_2}{\Longrightarrow} H_2$. Given a span $s_1 = (C_1 \overset{\supseteq}{\leftarrow} S_1 \overset{e_2}{\rightarrow} L_2)$ for rule pair (r_1, r_2) and mappings $m_1 : L_1 \rightarrow G$ and $m_2 : L_2 \rightarrow G$ we say that these mappings *overlap* in s_1 if $m_1(S_1) \cap m_2(e_2(S_1)) \subseteq m_1(L_1) \cap m_2(L_2)$.

Example [Conflict] In the graph in Figure 3 with the given matches of rules *returnUnpaidGood* and *offerGift*, the resulting pair of transformations is in conflict since $m_1(C_1 \setminus B_1) \cap m_2(L_2)$ contains node *5,5:OrderItem* with adjacent edges and edge *owns* from *1,2:Customer* to *6,6:Good*, i.e. these elements are deleted by the first transformation and used by the second one. The corresponding conflict reason is given by the conflict graph in Figure 2, its embedding into the deletion graph of rule *returnUnpaidGood*, and its mapping into the LHS of rule *offerGift*. Both embeddings are given by numbers (compare Figures 1 and 2). We focus on the conflict graph of a conflict reason while embeddings are just given by corresponding numbers as explained above.

Conflict concepts. Heading towards a static conflict analysis, we do not investigate each pair of graph transformations but analyze *rule pairs* instead. Rule pair (r_1, r_2) is in *conflict* if there is any pair of conflicting transformations applying rule r_1 and then r_2.

We further concentrate on rule parts that may cause conflicts. The minimal building bricks of conflict causes are called conflict atoms. An atom is derived from an *atom candidate* being a span $a_1 = (C_1 \overset{\supseteq}{\leftarrow} A_1 \overset{e_2}{\rightarrow} L_2)$ for rule pair (r_1, r_2), where A_1 is a single deleted node or edge incident with preserved nodes. It is deleted by the first rule and used by the second one. A deleted edge with at least one incident deleted node is not considered as atom candidate, since the edge is deleted together with the deleted node anyway. If a pair of transformations exists so that their match mappings overlap on the atom candidate, it is called *conflict atom*. Note that, in general, the matches of such a pair of transformations may overlap also in graph elements other than the conflict atom.

A *conflict reason* $s_1 = (C_1 \overset{\supseteq}{\leftarrow} S_1 \overset{e_2}{\rightarrow} L_2)$ for rule pair (r_1, r_2) subsumes all atoms being involved in a conflict. Among the conflict reasons for two rules, there are *minimal reasons* describing minimal compositions of atoms leading to conflict reasons.

In contrast to conflict reasons showing conflict-causing *rule overlaps*, a *critical pair* consists of two conflicting *transformations* applying two rules in a minimal context, where all elements stem from L_1 or L_2 or both. Critical Pair Analysis (CPA) [27, 48] is the state-of-the-art static CDA for graph transformation. The set of critical pairs has the important property that it is *complete*: each possible conflicting pair of transformations is represented by some critical pair. Two important subsets of critical pairs being still complete have been identified in the literature: An *essential critical pair* [42] is one where the rules' LHSs overlap merely deletion graph elements of one rule with LHS elements of the other rule, i.e. only the conflict reason is overlapped. The rationale is that overlapping additional preserved elements (as done in regular CPA) does not contribute to new conflicts. (Essential) Critical pairs can be computed with the state-of-the-art implementation of the CPA in AGG [59] and VeriGraph [5]. An *initial conflict* [40] is an essential critical pair without isolated boundary nodes. The latter arise when a preserved node with incident deletion edges of the first rule is overlapped with a node of the second rule without overlapping

any incident deletion edge of the first rule. Such overlaps do not contribute to new conflicts. Initial conflicts represent currently the most optimal subset of critical pairs fulfilling the completeness property, but their detection has not been implemented yet.

Example [Conflict concepts]. Considering the conflict reason in Figure 2, it is covered by two conflict atom graphs consisting of node *5,5:OrderItem* and edge *owns* from *1,2:Customer* to *6,6:Good*. We can complete conflict atom *5,5:OrderItem* to a minimal conflict reason by adding both adjacent edges and their incident source or target nodes. It is shown as top graph in Figure 1. The conflict atom including the *owns* edge already constitutes a minimal conflict reason shown underneath the top graph in Figure 1. Both together form the conflict reason shown in Figure 2. Overlapping the LHSs of both example rules at this conflict reason yields the graph in Figure 3. The resulting LHS embeddings into this graph are actually rule matches since they fulfill the dangling condition (as shown above). The corresponding transformations form a critical pair, which is actually an initial conflict here.

Dependency and dependency concepts. For reasoning about dependencies, where graph elements being produced by the first transformation are used by the second one, we simply consider the dual concepts by inverting the first transformation of a conflicting pair. Based on this analogy the concepts *dependency between rules* and *(minimal) dependency reason* are defined accordingly.

4.2 Formalizing granularity levels

As suggested by the granularity requirements derived from our literature survey in Sec. 3, various use-cases may require either fine-grained, coarse-grained, and binary analysis results. To support all of these granularity levels in our technique, we formalized each granularity level with appropriate conflict and dependency concepts as outlined in this section. A full account of formal definitions, characterizations and proofs is given in [43].

Overapproximation. Our survey (in Sec. 3) indicates serious performance problems in practice when computing conflict information on a fine-grained level in the form of (essential) critical pairs, as done in the state-of-the-art implementation in AGG [59]. Therefore we propose as first improvement a well-chosen *overapproximation* of results that is easier to compute. It is based on the idea that, if there is a conflict for a pair of rules (r_1, r_2'), then an equivalent conflict exists for the rule pair (r_1, r_2) with r_2 being the *non-deleting variant* of rule r_2'. The other direction does not hold, since the dangling condition of rule r_2' could be violated then.

Mapping conflict concepts. Fig. 4 gives an overview of how we further map conflict concepts to granularity levels.

Since initial conflicts represent currently the most optimal subset of critical pairs being still complete [40], we have chosen to return all *conflict reasons* for (r_1, r_2) corresponding to initial conflicts as new *fine-grained results*. An initial conflict for (r_1, r_2) with r_2 being non-deleting simply corresponds to the overlap of such a conflict reason. As reported in Sec. 6, this choice represents a very good trade-off between precision and performance.

Moving to the *coarse-grained level*, we selected *minimal conflict reasons*, since our considered conflict reasons at the fine-grained level are composed of minimal ones. The conflict graph of minimal conflict reasons for a rule pair (r_1, r_2) with r_2 non-deleting can be characterized as subgraph of a so-called deletion component. The deletion part $L_1 \setminus K_1$ of a given rule r_1 may consist of several disjoint fragments, called *deletion fragments*. Completing a deletion fragment to a graph by adding all incident boundary nodes from B_1 yields a *deletion component*. Each two deletion components overlap in boundary nodes only; the union of all deletion components coincides with the deletion graph C_1 of the rule. Deletion components thus specify maximal parts of C_1 that need to be overlapped in order to find corresponding conflicting transformations. Overlapping more elements than present in a deletion component can never lead to additional minimal conflict reasons since, if the dangling condition of r_1 was not fulfilled before, it will never be fulfilled.

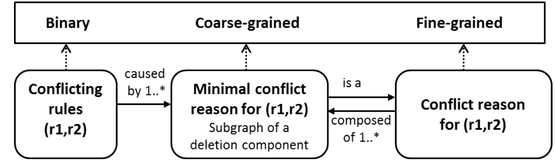

Figure 4: Conflict concepts & mapping to granularity

Example [Deletion components] Rule *returnUnpaidGood* in Figure 1 has three deletion fragments: Node *5:OrderItem* with adjacent edges, node *7:BillItem* with adjacent edges, and edge *owns* from node 1 to node 6. For completing them to deletion components, boundary nodes *1:Customer*, *2:Order*, *4:Bill*, and *6:Good* are needed.

On the *binary level*, we report if a *rule pair is conflicting*, since this means by definition that there is at least one pair of conflicting transformations via these rules. In particular, we can check if there is a minimal conflict reason for the given rule pair (r_1, r_2) with r_2 non-deleting, equivalent with the rule pair being conflicting.

Example [Granularity levels] Two minimal conflict reasons of rule pair (*returnUnpaidGood, offerGift*) were discussed above. Hence, these two rules are in conflict. Two further minimal conflict reasons arise if node 5 of the first rule overlaps with node 3 of the second rule, and if node 6 of the first rule is overlapped with node 4 of the second one. These four minimal reasons form the coarse analysis result of the considered rule pair. Together with all their possible combinations, we get 6 conflict reasons being reported as “New Fine” analysis result in Table 2.

Mapping dependency concepts. As mentioned before, a *dependency* can be understood as dual concept to conflicts. Analogous to the conflict case, we overapproximate produce-use dependencies by produce-read dependencies. Then *dependencies between rules*, *minimal dependency* and *dependency reasons* are mapped to the binary, coarse and fine granularity level accordingly.

5 ALGORITHM SUITE

We present an algorithm suite for computing binary, coarse and fine-grained conflicts. It is implemented in Henshin [3, 57], a model transformation framework based on graph transformation concepts.

Given a pair of rules, we consider a non-deleting variant of the second rule, supporting the overapproximation presented in Section 4.2. The algorithm in Figure 5 computes minimal conflict reasons (*coarse granularity*). The algorithm in Figure 6 uses these results to compute all conflict reasons (*fine granularity*). To support use-cases where *binary granularity* is sufficient, we can stop the computation as soon as one minimal conflict reason is discovered.

```
1:  funct computeMinReasons(r1 : Rule, r2 : Rule)
2:      var reasons ← ∅
3:      for each c ← ComputeAtomCandidates(r1, r2) do
4:          computeMinReasonsRecursively(r1, r2, c, reasons)
5:      return reasons
6:
7:  funct computeAtomCandidates(r1 : Rule, r2 : Rule)
8:      var candidates ← ∅
9:      for each el1 ← r1.conflictInducingElements do
10:         for each el2 ← r2.lhs.occurenceOf(el1) do
11:             var S1 ← new Graph({el1})
12:             var embed_r1 ← new Mapping({el1↦el1})
13:             var embed_r2 ← new Mapping({el1↦el2})
14:             candidates += new Span(S1, embed_r1, embed_r2)
15:     return candidates
16:
17: funct computeMinReasonsRecursively(r1 : Rule, r2 : Rule, s1 :
    Span, reasons : Set<Span>)
18:     var (G, m1, m2) ← constructOverlap(r1, r2, s1)
19:     var isCR_s1 ← findDanglingEdges(r1, m1).isEmpty()
20:     if isCR_s1 then reasons += s1; return; else
21:     for each s2 ← extendSpan(r1, r2, s1, reasons) do
22:         computeMinReasonsRecursively(r1, r2, s2, reasons)
23:
24: funct extendSpan(r1 : Rule, r2 : Rule, s1 : Span, reasons : Set<Span>)
25:     var (G, m1, m2) ← constructOverlap(r1, r2, s1)
26:     var dangling ← findDanglingEdges(r1, m1)
27:     var fixing ← dangling.foreach(findFixingEdges(r1, r2, s1))
28:     return aggegrate(fixing.foreach(enumerateExtensions(s1)))
```

Figure 5: Computing minimal conflict reasons.

Computing minimal conflict reasons. For efficiency, we use the characterization of minimal conflict reasons as introduced in Sect. 4.2. The key idea is to compute first the set of all *conflict atom candidates* (line 3). To this end, all conflict-inducing elements of r_1 are identified in line 9. Each match of such an element to r_2 (line 10) is extended to a span (lines 11–14) yielding an atom candidate. Next, we try to extend each atom candidate to minimal conflict reasons (line 4) recursively and return the results (line 5).

To determine efficiently if a particular candidate can be extended to a minimal conflict reason recursively, we check if it gives rise to a critical pair by computing the overlap graph G of the rules' LHSs along the candidate (line 18) and checking if the corresponding embeddings $m_1 : L_1 \longrightarrow G$ and $m_2 : L_2 \longrightarrow G$ are rule matches (line 19). Since we assume r_2 to be non-deleting, m_2 is automatically a match. We are finished if m_1 is a match as well (i.e. if r_1 can be applied at embedding m_1 without producing dangling edges) meaning that we found a reason (line 20). Otherwise, we extend candidate spans in potentially several ways (line 21), discarding further extension opportunities in case we identified a conflict reason. We computed all extensions by function `extendSpan()` (line 24). A span s_1 has to be extended if the rule' LHS embeddings into a given overlap graph do not fulfill the dangling condition (line 25). An extension is performed stepwise by first identifying all dangling edges (line 26). For each of them a set of fixing edges in r_2 is searched (line 27). Suitable candidates for this purpose are all adjacent edges e in $C_1 \setminus S_1$ of e's adjacent node in S_1 being identified by calling *findFixingEdges()*. For each fixing edge, function *enumerateExtensions()* (line 28) yields a set of spans, each extending s_1 by an edge and its adjacent node if not previously included in s_1. All these sets have to be aggregated to obtain the result (line 28). Since we know that a minimal conflict reason is always part of a deletion component, we have a stopping criterion for the extension process, as we can focus on adjacent edges when computing extensions.

Computing conflict reasons. Starting from a set of minimal conflict reasons, all conflict reasons can be composed of them. Function `ComputeConflictReason` in Figure 6 picks any minimal reason *mr* out of a given set (line 4), adds it as reason (line 5), and composes it with the set *subReasons* of all conflict reasons computed from the remaining set of minimal reasons (lines 7–12). All conflict reasons in *subReasons* are first of all conflict reasons themselves (line 9) and second to be composed with *mr* if composable (lines 11–12).

```
1:  funct computeConflictReasons(minReasons : Set<Span>)
2:      var reasons ← ∅
3:      if minReasons.nonEmpty()
4:      then var mr ← minReasons.pickAny()
5:          reasons.add(mr)
6:          if minReasons.size ≥ 2
7:          then var subReasons = computeConflictReasons(
8:                                      minReasons.remove(mr))
9:              reasons.add(subReasons)
10:             for each s : subReasons do
11:                 if isComposable(mr, s)
12:                 then reasons.add(compose(mr, s))
13:     return reasons
```

Figure 6: Computing conflict reasons.

Computing dependency reasons. We invert rule r_1, compute all conflict reasons, and interpret them as dependency reasons.

Implementation. We implemented the algorithm suite for the Henshin model transformation language, which is based on graph transformation concepts and the Eclipse Modeling Framework (EMF). Our implementation includes a comprehensive test suite, in which the test oracle is provided by computing fine-grained conflicts in the state-of-the-art CDA framework AGG. Section 6 studies the performance of our implementation.

6 IMPLEMENTATION EVALUATION

We evaluated our analysis via comparison to the existing analysis in AGG, focusing on two research questions: ***RQ1**: How fast are our coarse and fine-grained analyses in relation to the existing analysis in AGG?* ***RQ2**: What is the degree of the overapproximation of the multi-granular CDA technique?*

We performed our evaluation on three subject rule sets, all of them representing use-cases identified in Sect. 3: Refac is a set of refactoring rules as used in refactoring recommenders [46]. FMEdit is a set of editing rules that was proposed in [58] as a benchmark for edit operation detection [35]. NanoXML is a set of rules that was reverse-engineered from the Java code for a small XML parser [1] to provide an oracle for test case generation [54]. The implementation and evaluation artifacts are available at `https://github.com/KristopherBorn/multiCDA`.

RQ1. We considered the *essential* CPA in AGG and compared it with our computations of minimal conflict/dependency reasons (MCR/MDR) as coarse-grained analysis and conflict/dependency reasons (CR/DR) as fine-grained one. We analyzed delete-use-conflicts between each rule pair considering a non-deleting version of the “use” rule in each case. Moreover, we preprocessed the rule set and meta-model such that they fit to the features supported by AGG and by our implementation. In the results (Table 4), we observed that our approach achieved a major speed-up from e.g. over 8 minutes to 5 seconds for 1681 rule pairs. In terms of quantity, we see that the mean number of results dropped considerably for the

Rule set	#Rule pairs	Runtime (m:ss.x) MCR MDR	CR DR	ess. CPA	#Results (mean) MCR MDR	CR DR	ess. CPA
Refac	64	**0:00.3**	**0:00.3**	0:08.7	**1.0**	**1.4**	2.6
FMEdit	1681	**0:05.3**	**0:16.3**	8:41.1	**0.5**	**1.3**	2.6
NanoXML	1296	**0:04.6**	**0:04.7**	1:30.4	**0.2**	**0.2**	0.2

Table 4: RQ1 results from our coarse (MCR/MDR) and fine-grained (CR/DR) vs. the existing analysis in AGG (ess. CPA).

Rule set	prec.	recall
Refac	0.88	1.0
FMEdit	0.98	1.0
NanoXML	1.0	1.0

Table 5: Overapproximation results

larger cases, e.g. from 2.6 to 0.5 for FMEdit rule pairs. There were even more striking results for individual rule pairs. In the most extreme case, we had 1588 ess. critical pairs, 644 conflict reasons, and 24 min. conflict reasons. A trend towards excessive individual cases is reflected in a higher standard deviation of results, which in the FmEdit case amounted to 1.4 for coarse-grained, 18.2 for fine-grained, and 49.1 for the essential CPA results.

RQ2. We computed the sets of essential critical pairs (ess. CPs) for an original rule pair and the one for a pair where the second rule is the non-deleting variant of the original one. Both sets of ess. CPs have been filtered w.r.t. initial conflicts (dependencies). The precision is the percentage of rule pairs with equal numbers of initial conflicts (dependencies). Since we do an overapproximation, the recall is always 1.0, i.e., we do not miss a critical pair when switching to the non-deleting rule variant. The precision, however, happens to be smaller than 1.0, i.e., false positives can occur. As the results in Table 5 show, the resulting precision is still acceptable.

Threads to validity. External validity can be questioned since we focus on a limited number of rule sets being preprocessed according to unsupported features such as application conditions [19] and amalgamation [11]. We intend to support more transformation features in the future, thus enabling a more comprehensive study with more expressive subject rule sets.

7 USER STUDY

The goal of our user study is to test the usefulness of our technique's output compared to that of its predecessor, critical pair analysis (CPA). As discussed in Section 3, in many use-cases, coarse-grained results may provide a more suitable level of detail than the fine-grained ones produced by CPA. Focusing on such use-cases, we investigated the following research question: *How useful are our coarse-grained results compared to fine-grained ones?*

We conducted a user study in which comprehension tasks had to be solved based on conflict analysis results. The analysis results were embedded as screenshots into an online questionnaire. With this web-based setup, we aimed to recruit a sufficient number of participants with appropriate expertise in graph transformations. A replication package with all tasks and data from our user study is available at `http://uni-marburg.de/y35ak`.

Focusing on the usefulness concerns of user performance and perception, our null hypotheses were as follows: ($H0_{perf}$) *Users can solve comprehension tasks equally well using the given coarse- and fine-grained results;* ($H0_{perc}$) *Users perceive the usefulness of the given coarse- and fine-grained results as equal.* We used a standard experimental setup involving independent, dependent, and controlled variables. Granularity was the independent variable. User performance and perception were the dependent variables. We controlled the used examples and the chosen visualization by keeping them constant. All analysis results were shown in the result visualization of AGG [59].

Methods, participants, materials. Our study design is a *cross-over study*, a variant of *within-subject design* [34] in which all participants are sequentially exposed to both treatments (here: coarse- and fine-grained analysis results). We selected this design because it minimizes the number of participants necessary to identify statistically significant differences between the result types. The main threat to the validity for this design are learning effects. We later discuss threats and adopted mitigation measures.

The experiment took place in Summer 2017. To recruit subjects with appropriate expertise, we invited participants of recent software engineering conferences focusing on model and graph transformations, and authors of the papers surveyed in Sect. 3. We sent invitations to 131 persons, 33 of which participated in the survey. Our overall sample consisted of 24 academic/scientists, 5 PhD students, 3 practitioners, and 1 MSc student. To survey the shared software-engineering and graph-transformation background of our participants, we collected demographic data based on five-point Likert scales. 30 respondents rated their experience with UML-based modeling 3 or higher (20 of them ≥4), like 32 did for their graph-transformation experience (in 27 cases ≥4), 25 participants rated their experience as CDA users as 3 or higher (14 cases ≥4).

The questionnaire was made up of three parts: an instruction part, an experimental part, and a survey part. The *instruction* part used an example to revisit basics about conflicts of graph transformations and familiarize the participants with the used visualization. The *comprehension* part included question-based comprehension tasks. In the *survey* part, we aggregated the demographic information and asked the participants for their subjective experiences.

The instruction and comprehension parts used a common example domain, namely the detection of conflicting requirements during the development of an online shop. The rationale for choosing this domain was twofold: First, it did not require expertise in a specialized technical environment. Second, it is based on a use-case in which we hypothesize that coarse-grained results are beneficial, namely consistency validation of use-cases [26]. The example in Sect. 2 is representative for this example domain and use-case.

In the comprehension part, each participant solved two tasks, one for each granularity level. Each task included an example rule pair, a number of conflict analysis results, and two questions: First, whether any conflicts existed for the rule pair. Second, if the first question was answered "yes", we asked to name the specific elements causing the conflicts. The rule pairs and their order were the same for each participant. The order of the granularity levels used to present the conflicts was assigned randomly. Based on the questions, the task metrics were measured as follows: A correct answer for the first question was rewarded with 1 point. The second question was more complicated, which was reflected in 2 points for a fully and 1 point for a partially correct answer. The time needed to complete both tasks was measured using the questionnaire.

Table 6: User study results.

Conflict type	Correctness	Completion time	Understandability 5=hard to understand	Simplicity 5=hard to solve tasks	Effort 5=much effort
Fine-grained	93.9 % (± 13.0 %)	7.7 min (± 6.6 min)	2.6 / 5 (± 0.9)	2.5 / 5 (± 1.0)	2.9 / 5 (± 1.1)
Coarse-grained	91.9 % (± 14.5 %)	5.7 min (± 2.9 min)	2.0 / 5 (± 0.9)	2.0 / 5 (± 1.1)	2.0 / 5 (± 0.9)
	p = 0.42	*p = 0.10*	*p = 0.004*	*p = 0.008*	*p ≤ 0.001*

We tested the tasks in a prestudy with 10 participants, in which we experimented with various difficulty levels. We aimed to balance complexity—the drawbacks of fine-grained results are more obvious for complex examples and tasks—and simplicity, to avoid participant exhaustion and to benefit completion rate. To this end, we decided to drop additional tasks concerning dependency analysis and conflict repair. The actual tasks used in our experiment were based on the conflict example shown in Sect. 2 (Task 1) and a comparable one (Task 2). Fine-grained results represent essential critical pairs, as provided by the state-of-the-art CDA tool AGG. Coarse-grained results are those of our multi-granular CDA.

In the survey part, to measure user perception, the participants were asked for their subjective assessment of both granularity levels. Three metrics were collected using five-point Likert scales: ratings of understandability ("*How easy was it to understand the results of type X?*"), difficulty of solving tasks ("'*How difficult was it to answer the questions using the results of type X?*"), and perceived effort ("'*How much effort was required to answer the questions using the results of type X?*"'). Finally, we asked them for their overall preference between both granularity levels on a five-point Likert scale. Additional qualitative information regarding the subjective assessment was collected using free-form text fields.

For statistical hypothesis testing, we used the Wilcoxon signed-ranked test [22] and, where applicable, the paired-samples t-test [63]. Wilcoxon is a standard nonparametrized test that supports high-confidence inferences for paired data. The t-test provides greater statistical power than Wilcoxon for normally distributed data. We checked for normality using the Shapiro-Wilk test [55].

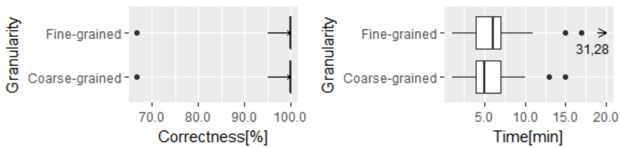

Figure 7: Correctness and completion time.

Results. Table 6 summarizes the results in terms of descriptive and inferential statistics. The task metrics are visualized with boxplots in Fig. 7. Irrespective of whether coarse- or fine-grained detection results were used, the vast majority of all participants answered the tasks with full correctness scores. The mean time required to complete the tasks was shorter by 33 % (2 minutes) when coarse-grained results were used. Yet, the completion times for the fine-grained case included two excessive data points of 28 and 31 minutes, which can be considered as outliers. After removing them, the mean completion time for coarse-grained is still lower by 0.6 minutes. The completion times did not not differ to a statistically significant extent (p=0.10). In summary, since we cannot find a significant effect on correctness and completion time, our first null hypothesis $H_{0_{perf}}$ cannot be rejected.

The perception metrics of understandability, difficulty, and effort are visualized in Fig. 8. In all cases, the participants reported more positive (70–79 % vs. 36–55 %), fewer negative (9–12% vs. 18–36 %) , and fewer neutral scores (12–18% vs. 24–33 %) when working with the coarse-grained detection results: they experienced better understandability, ease of solving tasks, and less effort. The differences were strongly statistically significant (p ≤ 0.01). In summary, our second null hypothesis $H_{0_{perc}}$ is rejected.

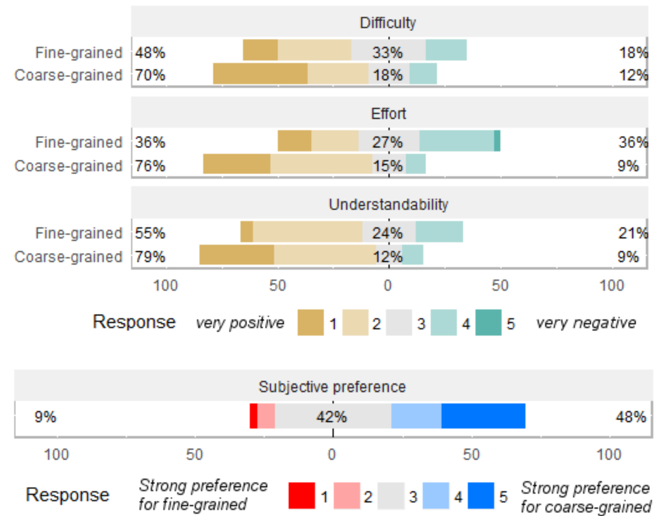

Figure 8: User perception and preference.

Finally, we asked for an overall subjective preference. Fig. 8 gives a distribution overview: coarse-grained results attracted more subjective preferences than fine-grained ones. In fact, the overall sample is mostly divided into two groups: those participants with a preference for the coarse-grained results (48%, 30 % of them with a strong preference), and those without a preference (42 %).

In qualitative information collected using the questionnaire, we found that the reasons for preferring were largely in line with our motivation: Participants found that *"less results provided a quicker overview"*, and that coarse-grained results *"boil down the problem to actual distinguish conflicts"*. Some subjects extrapolated from our tasks to more complex cases: *"Assuming that we have a larger set of rules that conflict with each other, I'd assume that the [fine-grained] results will become very complex and hard to understand*".

We also found reasons why this preference was not shared unanimously. Some participants found the redundancy of fine-grained beneficial as it *"presented conflicts more consistently*" and felt that coarse-grained results did not mirror their understanding of conflicts: *"I believe that [fine-grained] displays all conflicts, whereas [coarse-grained] only displays selected conflicts."* This group of users may still benefit from our faster computation of fine-grained results.

Threats to Validity. *Construct validity.* For our effort and understandability measurements, we rely on subjective ratings by the participants. However, such subjective measures are highly correlated with objective measures of cognitive load [24]. Moreover, we aimed to avoid participant bias in favor of the experimenters by replacing the names of the used tools and concepts with pseudonyms.

Internal validity. The main threat in our within-subject design are learning effects, in particular, since the same questions were asked for each analysis result type (albeit for different tasks). We mitigated this threat by using counterbalancing, i.e., randomizing the order in which the results were shown to the participants.

External validity. We addressed external validity by recruiting a sufficiently large number of participants with relevant software engineering and CDA expertise. However, our web-based setup required some trade-offs: the sample analysis results were relatively small (3–10 entries) and were shown using screenshots instead of in the actual tool. Generalizations to a greater variety of use-cases are threatened since we considered a single domain and use-case. While our discussion in Sect. 3 highlights key similarities to other domains, a definitive verdict on the practical usefulness of coarse-grained results is outside the scope of this paper.

Summary. Having identified use-cases in which the level of detail offered by our coarse-grained results seems sufficient (Sect. 3), we set out to empirically study their usefulness in one such use-case. While we did not detect an effect on participants' performance, the participants perceived coarse-grained results as easier to understand and work with than fine-grained ones. The relative majority of participants preferred coarse-grained over fine-grained results. Our user study complements significant performance benefit achieved by our multi-granular computations and, to the best of our knowledge, provides the first empirical evidence on the usefulness of CDA for graph transformation.

8 RELATED WORK

In this paper, we have presented a generic, multi-granular CDA technique based on graph transformation (GT) which is statically analyzing transformation rules; it is fully automatic and state-independent. We compare with other analysis techniques along these aspects.

Further GT analysis techniques. Another *static analysis* technique for GT checks for invariants [7, 9]. In contrast, *model checking techniques* for GT [21, 51] need an initial start graph and then generate for this graph and a given rule set a corresponding state space in order to analyze more complex temporal logic properties. Whereas both kinds of analysis techniques are fully automatic, there exist also *interactive theorem proving techniques* dedicated to GT that are able to prove partial correctness of graph programs [25, 49].

With respect to designing and performing *analysis techniques with different granularity levels*, we can relate our work in a broader sense to the work on *counter example guided abstraction refinement* (CEGAR) initially described in [14] and applied to the analysis of GT in [37]. Our approach contains a similar idea in the sense that we perform a stepwise analysis technique with different levels of accuracy and terminate with the level of accuracy that is needed. In the CEGAR approach the desired level of accuracy is obtained if either the desired property for the analyzed system is successfully verified or a real counterexample has been detected. In our approach there can be many different use-cases to be satisfied with a certain level of accuracy of the analysis as described in detail in the literature review in Section 3.

Other generic CDA techniques. Besides GT there exist other formal approaches which allow for static and dynamic CDA of specified systems. A Church-Rosser-Checker for equational specifications in Maude [16] looks for critical pairs between conditional *term rewriting* rules and tries to join them. There are also logic-based approaches such as *conditional transformations* [36] providing static CDA techniques. These techniques are generic and therefore language-independent but the analysis techniques provided are not multi-granular. Further logic-based approaches such as *constraint networks* and *model checking* approaches such as Alloy [32] are not state-independent.

Language-specific CDA techniques. There are many sophisticated CDA techniques in the context of sequential and concurrent programming. In the context of concurrent threads, conflicts are called *data races*; they occur if two different threads access the same memory location and at least one of them is a write (see e.g. [28, 56]). Common challenges of these techniques are dynamic object creation, dynamic thread creation, and references to objects [52]. Similarly, there has been a long line of works developing *data flow analyses* to find dependencies between object changes and accesses; for example, static taint analyses have been recently developed for Android applications as e.g. FlowDroid [4]. These CDA techniques differ considerably to ours in several aspects: they are not generic but have been specifically developed to find bugs in implementations. To the best of our knowledge, there are no language-specific CDA techniques for reasoning about software engineering activities. Language-specific CDA techniques are mostly dynamic and therefore state-dependent. This is often not suitable for engineering activities such as refactorings which are specified state-independently. Moreover these CDAs are usually not multi-granular so that gradual analysis time reduction and easy understanding are not especially supported.

9 CONCLUSION

In this paper, we presented a novel static CDA technique for graph transformation which can detect conflicts and dependencies in software engineering on multiple granularity levels. Compared to the state-of-the-art CDA, we were able to significantly speed-up the computation of fine-grained CDA results and to complement them with coarse-grained ones offering usability benefits for numerous use-cases. Our technique is especially advantageous for analyzing interactions on complex and dynamic object structures as, e.g., for feature interaction in software product line engineering.

Our multi-granular CDA technique currently uses transformation rules without *advanced features* such as application conditions [19] and amalgamation [11], for which the state-of-the-art CDA technique has been investigated already. In the future, we aim to support these more complex concepts in our technique as well. Due to the major speed-up we could achieve with our multi-granular approach, this technique has now the potential to be applied in fields where performance plays a larger role.

Acknowledgements. We wish to thank all participants of the user (pre-)study for their participation and constructive feedback.

REFERENCES

[1] A. Alshanqiti and R. Heckel, "Extracting Visual Contracts from Java Programs (T)," in *ASE*, 2015, pp. 104–114.
[2] Z. Altahat, T. Elrad, L. Tahat, and N. Almasri, "Detection of syntactic aspect interaction in UML state diagrams using critical pair analysis in graph transformation," *CoRR*, vol. abs/1312.6939, 2013.
[3] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place EMF model transformations," in *MoDELS*. Springer, 2010, pp. 121–135.
[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014.
[5] G. G. Azzi, J. S. Bezerra, L. Ribeiro, A. Costa, L. M. Rodrigues, and R. Machado, "The Verigraph System for Graph Transformation," in *Graph Transformation, Specifications, and Nets. In Memory of Hartmut Ehrig*. Springer, 2018, pp. 160–178.
[6] L. Baresi, K. Ehrig, and R. Heckel, "Verification of model transformations: A case study with BPEL," in *TGC*, 2007, pp. 183–199.
[7] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling, "Symbolic invariant verification for systems with dynamic structural adaptation," in *ICSE*, 2006, pp. 72–81.
[8] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *ICSE*, 2012, pp. 419–429.
[9] C. Blume, H. J. S. Bruggink, and B. König, "Recognizable graph languages for checking invariants," *ECEASST*, vol. 29, 2010.
[10] K. Born, L. Lambers, D. Strüber, and G. Taentzer, "Granularity of conflicts and dependencies in graph transformation systems," in *ICGT*, 2017, pp. 125–141.
[11] K. Born and G. Taentzer, "An algorithm for the critical pair analysis of amalgamated graph transformations," in *ICGT*, 2016, pp. 118–134.
[12] P. Bottoni, G. Taentzer, and A. Schürr, "Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation," in *VL/HCC*, 2000, pp. 59–60.
[13] A. Bucchiarone, P. Pelliccione, C. Vattani, and O. Runge, "Self-repairing systems modeling and verification using AGG," in *WICSA/ECSA*, 2009, pp. 181–190.
[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, 2000, pp. 154–169.
[15] S. Degrandsart, S. Demeyer, J. Van den Bergh, and T. Mens, "A transformation-based approach to context-aware modelling," *Software & Systems Modeling*, vol. 13, no. 1, pp. 191–208, 2014.
[16] F. Durán and J. Meseguer, "A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications," in *WRLA*, 2010, pp. 69–85.
[17] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Monographs in Theoret. Computer Science. Springer, 2006.
[18] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing, 1999.
[19] H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas, "M-adhesive transformation systems with nested application conditions. part 2: Embedding, critical pairs and local confluence," *Fundam. Inform.*, vol. 118, no. 1-2, pp. 35–63, 2012.
[20] C. Ermel, J. Gall, L. Lambers, and G. Taentzer, "Modeling with plausibility checking: Inspecting favorable and critical signs for consistency between control flow and functional behavior," in *FASE*. Springer, 2011, pp. 156–170.
[21] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using GROOVE," *STTT*, vol. 14, no. 1, pp. 15–40, 2012.
[22] J. D. Gibbons and S. Chakraborti, "Nonparametric statistical inference," in *International encyclopedia of statistical science*. Springer, 2011, pp. 977–979.
[23] H. Giese, S. Hildebrandt, and L. Lambers, "Bridging the gap between formal semantics and implementation of triple graph grammars," *Software & Systems Modeling*, vol. 13, no. 1, pp. 273–299, 2014.
[24] D. Gopher and R. Braune, "On the psychophysics of workload: Why bother with subjective measures?" *Human Factors*, vol. 26, no. 5, pp. 519–532, 1984.
[25] A. Habel and K. Pennemann, "Correctness of high-level transformation systems relative to nested conditions," *Mathematical Structures in Computer Science*, vol. 19, no. 2, pp. 245–296, 2009.
[26] J. H. Hausmann, R. Heckel, and G. Taentzer, "Detection of Conflicting Functional Requirements in a Use Case-Driven Approach: A Static Analysis Technique Based on Graph Transformation," in *ICSE*, 2002, pp. 105–115.
[27] R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems," in *ICGT*, 2002, pp. 161–176.
[28] T. A. Henzinger, R. Jhala, and R. Majumdar, "Race checking by context inference," in *PLDI*, 2004, pp. 1–13.
[29] F. Hermann, H. Ehrig, U. Golas, and F. Orejas, "Efficient analysis and execution of correct and complete model transformations based on triple graph grammars," in *MDI*, 2010, pp. 22–31.
[30] F. Hermann, H. Ehrig, F. Orejas, and U. Golas, "Formal analysis of functional behaviour for model transformations based on triple graph grammars," in *ICGT*, 2010, pp. 155–170.
[31] S. Hildebrandt, L. Lambers, and H. Giese, "Complete specification coverage in automatically generated conformance test cases for tgg implementations," in *ICMT*, K. Duddy and G. Kappel, Eds., 2013, pp. 174–188.
[32] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002.
[33] P. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model composition in product lines and feature interaction detection using critical pair analysis," in *MoDELS*, 2007, pp. 151–165.
[34] B. Jones and M. G. Kenward, *Design and analysis of cross-over trials*. CRC Press, 2014.
[35] T. Kehrer, U. Kelter, and G. Taentzer, "Consistency-Preserving Edit Scripts in Model Versioning," in *ASE*, 2013, pp. 191–201.
[36] G. Kniesel, "Detection and resolution of weaving interactions," *Trans. Aspect-Oriented Software Development*, vol. 5, pp. 135–186, 2009.
[37] B. König and V. Kozioura, "Counterexample-guided abstraction refinement for the analysis of graph transformation systems," in *TACAS*, 2006, pp. 197–211.
[38] C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab, "Modeling dynamic reconfigurations in reo using high-level replacement systems," *Sci. Comput. Program.*, vol. 76, no. 1, pp. 23–36, 2011.
[39] J. M. Küster, C. Gerth, and G. Engels, "Dependent and conflicting change operations of process models," in *ECMDA-FA*, 2009, pp. 158–173.
[40] L. Lambers, K. Born, F. Orejas, D. Strüber, and G. Taentzer, "Initial conflicts and dependencies: Critical pairs revisited," in *Graph Transformation, Specifications, and Nets. In Memory of Hartmut Ehrig*. Springer, 2018, pp. 105–123.
[41] L. Lambers, H. Ehrig, L. Mariani, and M. Pezzè, "Iterative model-driven development of adaptable service-based applications," in *ASE*, 2007, pp. 453–456.
[42] L. Lambers, H. Ehrig, and F. Orejas, "Efficient conflict detection in graph transformation systems by essential critical pairs," *Electr. Notes Theor. Comput. Sci.*, vol. 211, pp. 17–26, 2008.
[43] L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Hübert, "Multi-granular conflict and dependency analysis in software engineering based on graph transformation: Extended version," 2018, https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/forschung/publikationen/.
[44] P. Leenheer and T. Mens, "Using graph transformation to support collaborative ontology evolution," in *AGTIVE*, 2008, pp. 44–58.
[45] K. Mehner-Heindl, M. Monga, and G. Taentzer, "Analysis of Aspect-Oriented Models Using Graph Transformation Systems," in *Aspect-Oriented Requirements Engineering*, A. Moreira, R. Chitchyan, J. Araújo, and A. Rashid, Eds. Springer, 2013, pp. 243–270.
[46] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Soft. and Sys. Modeling*, vol. 6, no. 3, pp. 269–285, 2007.
[47] T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and resolving model inconsistencies using transformation dependency analysis," in *MoDELS*, 2006, pp. 200–214.
[48] D. Plump, "Critical Pairs in Term Graph Rewriting," in *Mathematical Foundations of Computer Science*, vol. 841, 1994, pp. 556–566.
[49] C. M. Poskitt and D. Plump, "Verifying monadic second-order properties of graph programs," in *ICGT*, 2014, pp. 33–48.
[50] F. Qayum and R. Heckel, "Local search-based refactoring as graph transformation," in *SSBSE*, 2009, pp. 43–46.
[51] A. Rensink, Á. Schmidt, and D. Varró, "Model checking graph transformations: A comparison of two approaches," in *ICGT*, 2004, pp. 226–241.
[52] M. C. Rinard, "Analysis of multithreaded programs," in *SAS*, 2001, pp. 1–19.
[53] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations*. World Scientific, 1997.
[54] O. Runge, T. A. Khan, and R. Heckel, "Test case generation using visual contracts," *ECEASST*, vol. 58, 2013.
[55] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.
[56] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," *SIGPLAN Not.*, vol. 47, no. 1, pp. 387–400, Jan. 2012.
[57] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for emf model transformation development," in *ICGT*. Springer, 2017, pp. 196–208.
[58] D. Strüber, T. Kehrer, T. Arendt, C. Pietsch, and D. Reuling, "Scalability of Model Transformations: Position Paper and Benchmark Set," in *BigMDE*, 2016, pp. 21–30.
[59] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," in *AGTIVE*, 2003, pp. 446–453.
[60] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "Conflict detection for model versioning based on graph modifications," in *ICGT*, 2010, pp. 171–186.
[61] J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering @ ICSE*, 2007, pp. 214–225.
[62] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo, "MATA: A unified approach for composing UML aspect models based on graph transformation," *Transactions on Aspect-Oriented Software Development VI: Special Issue on*

Aspects and Model-Driven Engineering, pp. 191–237, 2009.
[63] D. W. Zimmerman, "Teacher's corner: A note on interpretation of the paired-samples t test," *Journal of Educational and Behavioral Statistics*, vol. 22, no. 3, pp. 349–360, 1997.